ПОЛТАВСЬКИЙ ТЕХНІКУМ ХАРЧОВИХ ТЕХНОЛОГІЙ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ХАРЧОВИХ ТЕХНОЛОГІЙ

Алгоритми і методи обчислень

КУРС ЛЕКЦІЙ

для студентів спеціальності 5.05010201 «Обслуговування комп'ютерних систем і мереж»

денної та заочної форми навчання

Полтава
2015

Гак П.В. Алгоритми і методи обчислень. Курс лекцій для студентів спеціальності 5.05010201 «Обслуговування комп'ютерних систем і мереж» напряму 6.050102 „Комп'ютерна інженерія"

Полтава: ПТХТ НУХТ, 2014. – _____ с.

Рецензенти: Павленко М.І. , викладач-методист ПТХТ НУХТ
Селін О.М. к.т.н., доц., доцент кафедри математичних методів системного аналізу ННК «ІПСА» НТУУ «КПІ»

Укладач: П.В.Гак , викладач спецдисциплін циклової комісії «Обслуговування комп'ютерних систем і мереж» ПТХТ НУХТ

СХВАЛЕНО
на засіданні циклової комісії «Обслуговування комп'ютерних систем і мереж»
Протокол № ____
від «___» _____ 2015 р.

ЗМІСТ

ЗМІСТОВИЙ МОДУЛЬ 1. БАЗОВІ ПОНЯТТЯ ТЕОРІЇ АЛГОРИТМІВ. ПОНЯТТЯ СТРУКТУРИ. СТРУКТУРНІ ТА ЛІНІЙНІ ТИПИ ДАНИХ.

Тема 1. Визначення алгоритму. Способи описання та властивості, класи алгоритмів.

1.1. Визначення алгоритму

За визначенням А.П.Єршова, інформатика - це наука про методи подання, накопичення, передавання та опрацювання інформації за допомогою комп'ютера. Що таке інформація? Вважається, що інформація - це поняття, яке передбачає наявність матеріального носія інформації, джерела і передавача інформації, приймача і каналу зв'язку між джерелом і приймачем інформації.

Основними в загальній інформатиці є три поняття: задача, алгоритм, програма. Відповідно, маємо три етапи в розв'язуванні задач (зазначимо, що, з точки зору інформатики, розв'язати задачу - це отримати програму, тобто, забезпечити можливість отримати рішення за допомогою комп'ютера): постановка задачі, побудова і обгрунтування алгоритму, складання і налагодження програми. Оскільки програма - об'єкт гранично формальний, а тому точний (можливо не завжди прозорий, навантажений неістотними із змістовної точки зору деталями, але недвозначний) то, пов'язані з нею об'єкти також мають бути точними. Алгоритм містить чіткий і ясний спосіб побудови результатів за точно вказаною в постановці задачі залежністю їх від наявних аргументів.

Відповідно до етапів маємо три групи засобів інформатики: специфікація, алгоритмізація і програмування.

Для специфікації задач в курсі застосовані засоби типу рекурентних співвідношень, рекурсивних визначень, а також прості інваріантні співвідношення, початкові й кінцеві умови.

Побудова алгоритму за точною постановкою задачі дає можливість його обгрунтування математичними методами. Більше того, існують класи задач, які дозволяють формальне перетворення специфікації в алгоритм. Вивченню деяких таких класів і відповідних методів відводиться важливе місце в нашому курсі.

Істотно, що, порівняно з програмою, алгоритм може перебувати на вищому рівні абстракції, бути вільним від тих або інших деталей реалізації, пов'язаних з особливостями мови програмування та конкретної обчислювальної системи. Засоби, прийняті для зображення алгоритмів, за традицією називають алгоритмічною мовою. До речі, так називалися також перші мови програмування високого рівня, наприклад, Алгол - це просто скорочення ALGOrithmic Language - алгоритмічна мова. Але, загалом, жодна мова програмування не може цілком замінити алгоритмічну мову, оскільки консервативна

Повинні існувати гарантії, що всі програми, складені вчора, в минулому році, десять років тому, не втратять значення ні сьогодні, ні завтра. Модифікація мови програмування призводить до небажаних наслідків: вимагає перероблення системи програмування, знецінює напрацьоване програмне забезпечення. У той же час алгоритмічна мова може створюватися спеціально для певної предметної області, певного класу задач або навіть окремої задачі. Вона може розвиватися навіть при створенні алгоритму, вбираючи в себе новітні результати.

Алгоритм - точне формальне розпорядження, яке однозначно трактує зміст і послідовність операцій, що переводять задану сукупність початкових даних в шуканий результат, або можна також сказати, що алгоритм - це кінцева послідовність загальнозрозумілих розпоряджень, формальне виконання яких дозволяє за скінченний час отримати рішення деякої задачі або будь-якої задачі з деякого класу задач.

Алгоритм — не скінченна послідовність команд, які треба викопати над вхідними даними для отримання результату.

Приклад 1.1. Обчислити *(x+y)/(a-b)*

A=(A1,A2, A_3)

A_1: x+y

A_2: a-b

A_3: A_1 / A_2

Слово алгоритм походить від algorithmi - латинської форми написання імені великого математика IX ст. Аль-Хорезмі, який сформулював правила виконання арифметичних дій.

Приклад 1.2. Розглянемо відому задачу про людину з човном (Л), вовком (В), козою (Кз) і капустою (Кп). Алгоритм її розв'язання можна подати так:

{ Л, В, Кз, Кп —> } - початковий стан,
пливуть Л, Кз, Кп
{ В —> Л, Кз, Кп } - 1-ий крок,
пливуть Л, Кз
{ Л, В, Кз —> Кп } - 2-ий крок,
пливуть Л, В
{ Кз —> Л, В, Кп } - 3-ій крок,
пливуть Л
{ Л, Кз —> В, Кп } - 4-ий крок,
пливуть Л, Кз
{—>Л, В, Кз, Кп } - кінцевий стан

1.2 Способи описання алгоритмів

Існують такі способи описання алгоритмів:

- словесний,
- формульний,
- графічний,
- алгоритмічною мовою.

Перший спосіб — це словесний опис алгоритму. Словесний опис потребує подальшої формалізації.

Другий спосіб - це подавання алгоритму у вигляді таблиць, формул, схем, малюнків тощо. Він є найбільш формалізованим та дозволяє описати алгоритм за допомогою системи умовних позначень.

Третій спосіб — запис алгоритмів за допомогою блок-схеми. Цей метод був запропонований в інформатиці для наочності подання алгоритму за допомогою набору *спеціальних блоків. Основні з цих блоків подані на рис. 1.1.*

Четвертий спосіб - *це мови програмування. Справа в тому, що найчастіше в практиці виконавцем створеного людиною алгоритму являється машина і тому він має бути написаний мовою, зрозумілою для комп 'ютера, тобто мовою програмування.*

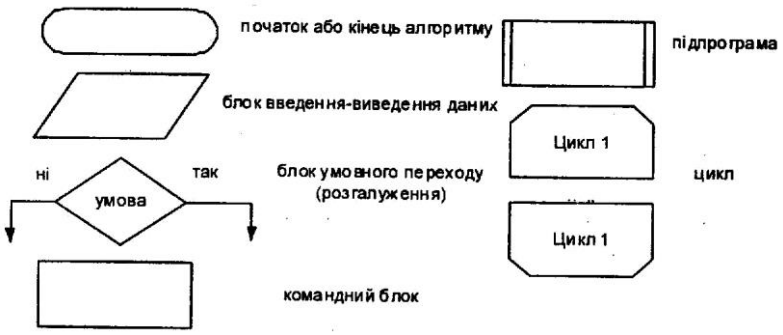


Рис. 1.1. Блоки для подання блок-схем.

1.3. Властивості алгоритмів

Розглянемо такі властивості алгоритмів: визначеність, скінченність, результативність, правильність, формальність, масовість.

Визначеність алгоритму. Алгоритм визначений, якщо він складається з допустимих команд виконавця, які можна виконати для деяких вхідних даних.

Приклад 1.3. Невизначеність виникне в алгоритмі (x+y)/(a-b), якщо в знамен-иику буде

записано, наприклад, 92/92 (ділення на нуль неприпустиме).

Невизначеність виникне, якщо деяка команда буде записана не-правильно, бо така команда не належатиме до набору допустимих команд виконавця, (x+y)/(a-b).

Скінченність алгоритму. Алгоритм повинен бути скінченним - послідовність команд, які треба виконати, мусить бути скінченною. Кожна команда починає виконуватися після закінчення виконання попередньої. Цю властивість ще називають **дискретністю** алгоритму.

Приклад 1.4. Алгоритм (x+y)/(a-b) — скінченний. Він складається з трьох дій. Кожна дія, у свою чергу, реалізується скінченною кількістю елементар-них арифметичних операцій. Нескінченну кількість дій передбачає математичне правило перетворення деяких звичайних дробів, таких як 5/3, у нескінченні десяткові дроби.

Результативність алгоритму. Алгоритм результативний, якщо він дає результати, які можуть виявитися і невірнимим.

Наведені вище алгоритми є результативними. Прикладом нерезультативного алгоритму буде алгоритм для виконання обчислень, в якому пропущена команда виведення результатів на екран тощо.

Правильність алгоритму. Алгоритм правильний, якщо його виконання забезпечує досягнення мети.

Приклад 1.5. Наведені вище алгоритми є правильними. Помінявши місцями в алгоритмі з прикладу 1.2 будь-які дві команди, отримаємо неправильний алгоритм.

Формальність алгоритму. Алгоритм формальний, якщо його можуть виконати не один, а декілька виконавців з однаковими результатами. Ця властивість означає, що коли алгоритм *A* застосовують до двох однакових наборів вхідних даних, то й результати мають бути однакові.

Приклад 1.6. Наведені алгоритми задовольняють цю умову, їх можуть виконати багато виконавців.

Масовість алгоритму. Алгоритм масовий, якщо він придатний для розв'язування не однієї задачі, а задач певного класу.

Приклад 1.6. Алгоритм з прикладу 1.1 не є масовим. Алгоритм Маляр є масовим, оскільки може застосовуватись не тільки для зафарбовування якихось елементів, але й для певних задач на графах. Прикладами масових алгоритмів є загальні правила, якими користуються для множення, додавання, ділення двох багатозначних чисел, бо вони застосовні для будь-яких пар чисел. Масовими с алгоритми розв'язування математичних задач, описаних у загальному вигляді за допомогою формул, їх можна виконати для різних вхідних даних.

1.4. Класи алгоритмів

Основною оцінкою функції складності алгоритму f(n) є оцінка Θ .

Кажуть, що f(n) = Θ *g(n)),* якщо при$g > 0$ при $n > 0$ існують додатні c_1, c_2, n_0, такі, що

$c_1 g(n) \leq f(n) \leq c_2 g(n)$

при $n > n_0$. Тобто, можна знайти такі c_1, та c_2, що при достатньо великих n функція знаходитиметься між $\boldsymbol{c_1 g(n)}$ та $c_2 g(n)$.

У такому випадку функція *g(n)* є асимптотично точною оцінкою функції f(n) , оскільки за визнченням функція f(n) не відрізняється від функціїg *g(n)* з точністю до постійного множника.

Виділяють такі основні класи алгоритмів:

- ✓ логарифмічні*: f(n) = Θ (log$_2$n);*
- ✓ *лінійні f(n)* = Θ (n); якщо $n = 1$, то отримуємо константі алгоритми;
- ✓ *поліноміальні: f(n)* = Θ (n^m); тут *m* - натуральне число, більше від одиниці; при $m = 1$ алгоритм є лінійним;
- ✓ *експоненційні*: *f(n)* = Θ (a^n); *a* - натуральне число, більше від одиниці. Експоненційні алгоритми часто пов'язані з перебором різних варіантів розв'язку.

Для однієї й тієї ж задачі можуть існувати алгоритми різної складності. Часто буває і так, що повільніший алгоритм працює завжди, а швидший - лише за певних умов.

Будемо називати **часовою складністю задачі** часову складність найефективнішого алгортітму для її розв'язання.

Алгоритми без циклів і рекурсивних викликів мають константну складність. Якщо немає

рекурсії та циклів, всі керуючі структури можуть бути зведені до структур константної складності. Отже, і весь алгоритм також характеризується константною складністю.

Визначення складності алгоритму в основному зводиться до аналізу циклів і рекурсивних викликів.

Приклад 1.7. Розглянемо алгоритм опрацювання елементів масиву. Нехай таким опрацюванням буде пошук заданого елемента.

```
For i=1 to N do
Begin
If a[i]=k then
…
End;
```

Складність цього алгоритму (N), оскільки тіло циклу виконується N разів, і складність тіла циклу рівна (1).

Якщо один цикл вкладений у інший і обидва цикли залежать від величини однієї і тієї ж змінної, то вся конструкція характеризується квадратичною складністю.

```
For i:=1 to N do
For j:=1 toN do
Begin
…
End;
```

Складність цієї програми (N^2).

Але, якщо заздалегідь відомо, що послідовність упорядкована за зростанням або за спаданням, можна застосувати інший алгоритм - алгоритм половинного ділення. Послідовність ділиться на дві рівні частини. Оскільки послідовність упорядкована, можна визначити, в якій частині міститься потрібний елемент. Після цього процедура повторюється: потрібна частина знову ділиться навпіл і т.д. Цей алгоритм є логарифмічним.

Дамо тепер визначення **складності класів задач P і NP. Клас P** складається 3S задач, для яких існують поліноміальні алгоритми розв'язання. ***Клас NP*** складають задачі, для яких існують поліноміальні алгоритми перевірки правильності рішення (точніше, якщо є розв'язок задачі, то існує деяка підказка, яка дозволяє за поліноміальний час отримати цю відповідь). Неформально кажучи, клас P складається **із** задач, які можна швидко розв'язати, а класс NP - зі задач, розв'язок яких можна швидко перевірити.

Наведемо приклад задачі класу P. Треба визначити, чи є у масиві дійсних чисел *A[1..n]* елемент зі значенням не меншим, ніж *k.* Очевидний у цьому випадку алгоритм перебирає всі елементи масиву за час (*n*).

Прикладом задачі класу NP є задача комівояжера (є множина міст та відділей між ними, мандрівний торговець мас відвідати усі міста, не заходячи у жодне двічі, з мінімальними витратами на дорору. Дійсно, якщо задано деякий маршрут завдовжки не більше ніж *k,* то за час(n) можна перевірити, що він дійсно має саме таку довжину, і тим самим переконатися у його існуванні. Для цього треба перебрати всі *n* переходів між містами, що містяться у маршруті, і додати їх довжини.

Очевидним є також включення P NP (для перевірки розв'язання задачі класу P досить розв'язати її поліноміальним алгоритмом).

Задача називається ***NP-повною,*** якщо вона належить класу NP і до неї за поліноміальний час можна звести будь-яку іншу задачу цього класу. Якщо якась NP- повна задача має поліноміальний алгоритм розв'язання, то всі NP-повні задачі можуть бути поліноміально розв'язані і, як наслідок, P=NP.

NP-повні задачі є найважчими у класі NP.

Експоненційні алгоритми та перебір

Експоненційні алгоритми часто пов'язані з перебором різних варіантів розв'язання.

Наведемо типовий приклад.

Приклад 1.8. Розглянемо задачу про виконуваність булевого виразу, яка формулюється так: **для** будь-якого булевого виразу від **/7** змінних знайти хоча б один набір значень змінних х,... x_n, при якому цей вираз приймає значення 1.

Типова схема розв'язування цієї задачі може мати такий вигляд: спочатку надаємо одне з двох можливих значень (0 або 1) першій змінній х,, потім другій і т.ін. Коли будуть розставлені значення всіх змінних, ми можемо визначити значення булевого виразу. Якщо він дорівнює 1, задача розв'язана. Якщо ні - треба повернутися назад і змінити значення деяких змінних.

Можна інтерпретувати цю задачу як задачу пошуку на певному дереві перебору. Кожна вершина цього дерева відповідає певному набору встановлених значень х ..., $x_к$. Ми можемо встановити змінну $x_ж$ в 0 або 1, тобто маємо вибір з двох дій. Тоді кожній із цих дій відповідає одна з двох дуг, які йдуть від цієї вершини. Вершина х, ..., $x_к$ має двох синів х, ..., $x_к$ 0 і х, ..., $x_к$ 1.

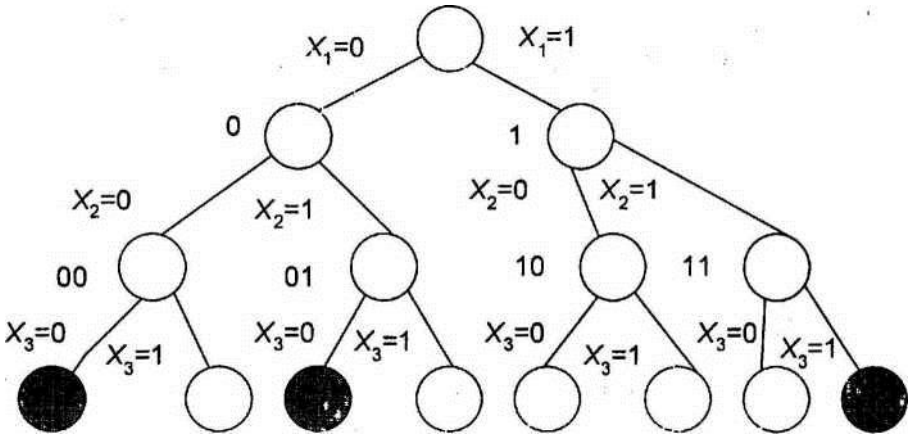


Рис 1.2. Пошук на дереві

Звертаємо увагу на те, що не кожен перевірний алгоритм є експоненційним. (Наприклад, алгоритм пошуку в масиві. Незважаючи на його перевірний характер, він є лінійним, а не експоненційним).

Зі зростанням розмірності будь-який поліпоміальний алгоритм стає ефективнішим, ніж будь-який експоИенційний. Дія лінійного алгоритму зросгання швидкодії комп'ютера в 10 разів дозволяє за той самий час розв'язати задачу, розмір якої в 10 разів більший. Для експоненційного алгоритму з основою 2 цей самий розмір можна збільшити лише на 3 одиниці.

Як правило, якщо для розв'язування якоїсь задачі є деякий пшіноміальний алгоритм, то часова оцінка цього алгоритму значно покращується.

Алгоритм із поверненнями назад

Метод перебору із поверненнями дозволяє розв'язувати практично незліченну множину задач, для багатьох з яких не відомі інші алгоритми. Незважаючи на таке велике різноманіття перебірних задач, в основі їх розв'язування є щось спільне, що дозволяє застосувати цей метод. Таким чином, перебір можна вважати практично універсальним методом розв'язування перебірних завдань. Наведемо загальну схему цього методу.

Розв'язування задачі методом перебору з поверненням будується конструктивно послідовним розширенням часткового розв'язування. Якщо на конкретному кроці таке розширення провести не вдається, то відбувається повернення до коротшого часткового розв'язування, і спроби його розширити продовжуються.

```
{пошук одного вирішення}
    procedure backtracking(k: integer); {k - номер ходу}
      begin
      {запис варіанту}
      if {рішення знайдене} then
            {виведення рішення}
```

```
            else
            {перебір всіх варіантів }
 if { варіант задовольняє умови задачі} then
            backtracking(k+1); { рекурсивний виклик}
               {стирання варіанту}
          end
begin
backtracking(l);
end.
```

Тема 2. Структура даних «масив», «множина», «таблиця», «стек», «черга».

2.1. Поняття структури даних типу «масив»

◆ *Масив* - послідовність елементів одного типу, який називається базовим. Математичною мовою масив - це функція з обмеженою областю визначення. Структура масивів однорідна. Для виділення окремого компонента масиву використається індекс. Індекс —— це значення спеціального типу, визначеного як тип індексу певного масиву. Тому на логічному рівні СД типу «масив» можна записати так: type A = array *[T1]* of T2, де Л - базовий тип масиву, 72 - тип індексу.

Якщо $\mathbf{D_n}$ - множина значень елементів типу 71, $\mathbf{D_n}$ - множина значень елементів типу 72, то **A:** $\mathbf{D_{Tl}}$ **®** $\mathbf{D_n}$ (відображення).

Кардинальне число Car(T) структури типу 7 - це множина значень, які може приймати задана структура типу Т. Кардинальне число характеризує об'єм пам'яті, необхідний такій структурі.

Для масиву А: Car *(A)* = [Car(72)] Car(71).

Масив може бути одновимірнгіч (вектором), та багатовимірним (наприклад, двовимірною таблицею), тобто таким, де індексом є не одне число, а кортеж (сукупність) із декількох чисел, кількість яких співпадає з розмірністю масиву.

У переважній більшості мов програмування масив є стандартною вбудованою структурою даних.

Отже, з вищенаведеного сформулюємо такі властивості масиву:

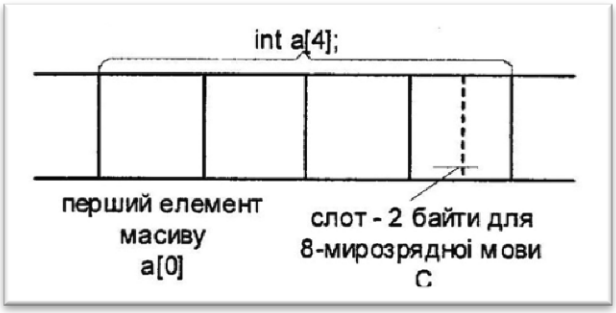
- ✓ *усі елементи масиву мають той самий тин;*
- ✓ *кожний компонент має свій номер у послідовності (індекс) і відрізняється ним від інших елементів (ідентифікується);*
- ✓ *множина індексів (індексова множина) скінченна й зафіксована в означенні масиву і ід час виконання програми не змінюється;*
- ✓ *можливість опрацювання компонента, або його доступність, не залежить від його місця в послідовності (елементи рівнодоступні).*

2.2. Набір допустимих операцій для сд типу «масив»

Над масивом можна виконувати такі операції:

- Операція доступу (доступ до елементів масиву - прямий; від розміру структури операція не залежить).
- Операція присвоювання.
- Операція ініціалізації (визначення початкових умов).

На фізичному рівні СД типу «масив» є неперервною ділянкою пам'яті елементів однакового об'єму. Ділянка пам'яті, необхідна для одного елемента, називається ***слотом.***



Var B: A {визначаємо змінну *B* як змінну типу «масив *A»}*;

p *<i* <g, де ***p*** - індекс першого елемента масиву, ***g*** - індекс останнього елемента масиву, *i* - індекс елемента.

2.3. Дескриптор сд типу «масив»

Нерідко фізичній структурі ставиться у відповідність дескриптор (заголовок), що ,і і /ить загальні відомості про задану фізичну структуру. Дескриптор також

зб рігається, як і структура, в пам'яті. Загалом дескриптор являє собою структуру т .пу «запис».

Стосовно до СД типу «масив», дескриптор містить такі компоненти: ім'я масиву, умовне позначення заданої структури, адресу першого елемента масиву, індекси нижньої й верхньої границь масиву, тип елемента масиву, розмір слота.

Наприклад, для наступного описування масиву: var *A:* array [-5 .. 4] of Char дескриптор буде виглядати так:

Для СД типу «масив» розмір дескриптора не залежить від розмірності масиву. При кожній операції доступу використовується вся інформація дескриптора. Наприклад, поля границі зміни індексу використовуються при обробці виняткових операцій.

2.4. ЕФЕКТИВНІСТЬ МАСИВІВ

Масиви ефективні при звертанні до довільного елемента, яке відбувається за постійний час (0(1)), однак такі операції як додавання та видалення елемента, потребують часу 0 (н), де п - розмір масиву. Тому масиви переважно використовуються для зберігання даних, до елементів яких відбувається довільний доступ без додавання або видалення нових елементів, тоді як для алгоритмів з інтенсивними операціями додавання та видалення, ефективнішими є зв'язані списки.

Інша перевага масивів, яка є досить важливою - це можливість компактного збереження послідовності їх елементів в локальній області пам'яті (що не завжди вдається, наприклад, для зв'язаних списків), що дозволяє ефективно виконувати операції з послідовного обходу елементів таких масивів.

Масиви є дуже економною щодо пам'яті структурою даних. Для збереження 100 цілих чисел у масиві треба рівно у 100 разів більше пам'яті, ніж для збереження одного числа (плюс, можливо, ще декілька байтів). У той же час, усі структури даних, які базуються на вказівниках, потребують додаткової пам'яті для збереження самих вказівників разом із даними. Однак, операції з фіксованими масивами ускладнюються тоді, коли виникає необхідність додавання нових елементів у вже заповнений масив. Тоді його слід розширювати, що не завжди можливо і для таких задач слід використовувати зв'язані списки, або динамічні масиви.

У випадках, коли розмір масиву є досить великий і використання звичайного звертання за індексом стає проблематичним, або великий відсоток його комірок не використовується, треба звертатися до асоціативних масивів, де проблема індексування великих об'ємів інформації

вирішується оптимальніше. В асоціативному масиві замість числових індексів використовуються ключі будь-яких типів. Дані в асоціативному масиві так само можуть бути різнотипними. Така структура також відома як «хеш» або як «словник». Це лише відображення «назва-значепня», як показано нижче:

```
h = {1 =>2, «2» => «4»}
print hash,»\n»
print hash[1],»\n»
print hash[«2»],»\n»
print hash[5],»\n»
```

З тої причини, що масиви мають фіксовану довжину, треба дуже обережно ставитися до процедури звертання до елементів за їхнім індексом, тому що намагання звернутися до елемента, індекс якого перевищує розмір такого масиву (наприклад, до

2.5. Сд типу «множина»

❖ **Множина** — *скінчений набір елементів одного типу, для яких не важливий порядок слідування і жоден з елементів не може бути два рази включений. Така СД визначається конструкцією type T = set of T0, де T0 - вбудований або раніше визначений тип даних (базовий тип). Значеннями змінних типу T є множини елементів типу T0 (зокрема, порожні множини).*

Кардинальне число множини (потужність) рівне кількості її елементів.

Набір допустимих операцій для СД типу «множина»: «*» - перетин множин, «+» - об'єднання множин, «-»-різниця множин, «іп» - перевірка належності до множини елемента базового типу.

Дескриптор СД типу «множина» не відрізняється від дескриптора СД типу «масив». Подамо програму, яка виводить усі цифри, що не входять у десятковий запис числа.

Для цього скористаємося множиною s, що міститиме усі цифри.

```
Vars:set of 0..9;
   n,ost,i:integer; {n - число, яке треба проаналізувати}
begin write('lnput number');
     readln(n);
     s:=[0,1 ,2,3,4,5,6,7,8,9]; {включили у множину всі цифри}
     while n>0 do
      {виділяємо цифри числа методом ділення його на 10}
      begin
         {визначаємо остачу від ділення}
         ost:=n mod 10;
         n:=n div 10;
         if (ost in s) then s:=s-[ost] {здійснюємо операцію різниці}
      end;
{виведемо всі цифри, що не належать до запису числа, за зростанням}
    for i:=0 to 9 do
     if i in s then write (i,',')
    end.
```

Найчастіше множини використовуються для формування набору елементів, що зустрічаються у масивах лише один раз.

2.6. Сд типу «таблиця»

❖ *Таблиця* — послідовність записів, які мають ту саму організацію. Такий окремий запис називається *елементом таблиці.* Найчастіше використовується простий запис. Отже**,** таблиця - це агрегація елементів. Якщо послідовність записів впорядкована щодо певної ознаки, то така таблиця називається *впорядкованою,* інакше - таблиця *невпорядкована.*

Класифікацію СД типу таблиця подано на рис. 1.1.

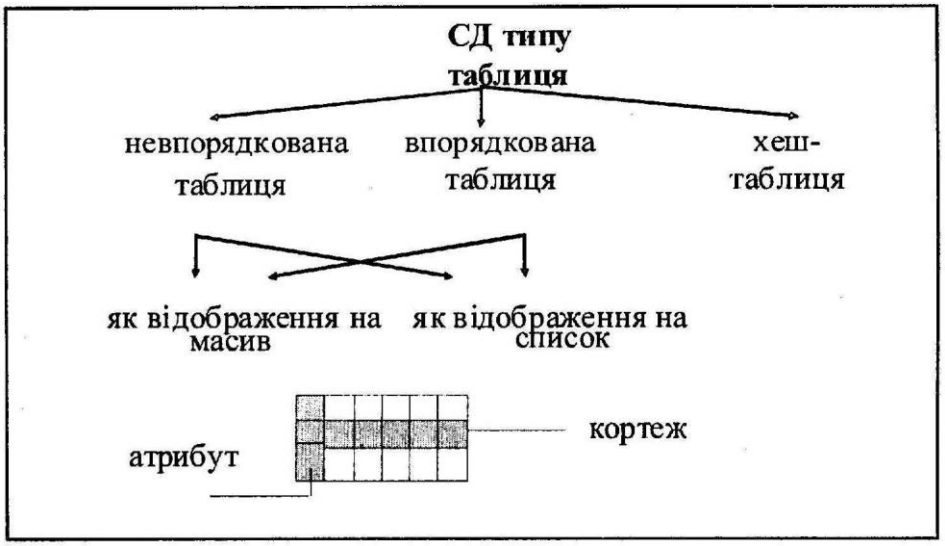


Рис. 1.3. Класифікація СД типу «таблиця»

Якщо один елемент d, то *кортеж*-це <d_v d_v .... d>, причому DT_t O d. Множина значень елементів типу T (множина допустимих значень СД типу «таблиця») буде визначатися за допомогою прямого декартового добутку:

$DT = DT_1$ **r** DT_2 r ... DT_n, *причому* DT_i <d_1 ,d_2 ,d_n>...

Сам елемент таблиці можна подати у вигляді двійки <*K,V*>, де *K* - ключ, а *V*— тіло елемента. Ключем може бути різна кількість полів, які визначають цей елемент. Ключ використовується для операції доступу до елемента, тому що кожний із ключів унікальний для заданого елемента. Отже, таблиця є сукупністю двійок <*K, V*>.

На логічному рівні елемент СД типу «таблиця» описуванняється так (приклад на мові Паскаль):

Type Element = record Key: integer;

{опис інших полів} end;

При реалізації таблиці як відображення на масив її опис виглядає так:

Tabl = array [0 .. N] of Element.

Під час виконання програми кількість елементів може змінюватися. Структура, у якій змінюється кількість елементів під час виконання програми, називається *динамічною.* Якщо розглядати динамічну структуру як відображення на масив, то така структура називається *напівапатичною.*

Перед тим як визначити операції, які можна виконувати над таблицею, розглянемо **класифікацію операцій.**

Конструктори — операції, які створюють об’єкти розглянутої структури. *Деструктори* - операції, які руйнують об’єкти розглянутої структури. Ознакою цієї операції є звільнення пам’яті.

Модифікатори - операції, які модифікують відповідні структури об’єктів. До них належать динамічні й напівстатичні модифікатори.

Спостерігачі — операції, у яких елементом (вхідним параметром) є об’єкти відповідної структури, а повертають ці операції результати іншого типу. Отже, операції- спостерігачі не змінюють структуру, а лише подають інформацію про неї.

Ітератори - оператори доступу до вмісту частини об’єкта у певному порядку. **Набір допустимих операцій для СД типу «таблиця» подаємо нижче**

- Операція ініціалізації (конструктор).

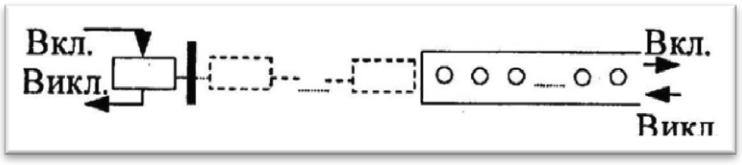
- Операція включення елемента в таблицю (модифікатор).
- Операція виключення елемента з таблиці (модифікатор).

Операції-предикати:

- таблиця порожня / таблиця не порожня (спостерігач),
- таблиця переповнена / таблиця не переповнена (спостерігач).
- читання елемента за ключем (спостерігач).

2.7. Сд типу «стек»

❖ ***Стек*** - це послідовність, у якій включення й виключення елемента здійснюється з однієї сторони послідовності (вершини стека). Так само здійснюється й операція доступу. Структура функціонує за принципом LIFO (останній, що прийшов, обслуговується першим). Умовні позначення стека зображені на рис



а) відображеня на масив б) відображення на список

Рис. 1.4. Організація стека.

При реалізації стека розглядаються стек як відображення на масив і стек як відображення на список.

Відображення на масив передбачає оголошення звичайного масиву та змінної, значення якої дорівнюватиме значенню індексу елемента, що відіграватиме роль «голови» (елемента, на який вказуватиме вказівник):

```
int a[100]; // оголошення стеку
    int n=0; // дійсна кількість елементів у стека
    int current=99; // індекс «голови», діє принцип LIFO
    void pop () // функція видобування (вилучення) елемента зі стека
    {
        if (n!=0)
        {
              current++; // зсунули «голову» на один елемент вперед
            printf('‘%d%, a[current]);
            n—;          // зменшили кількість елементів
        }
    }
    void push 0 // функція додавання елемента до стеку
    {
        if (п<99)
        {
            current—;
            //додали «голову», зсунувши індекс на один елемент вперед
            scanf(“%d%, &a[current]);
            n++;          // збільшили кількість елементів
        }
    }
```

Відображення на список передбачає оголошення динамічної структури. Перевагою такого відображення є відсутність обмежень на максимальну кількість елементів у стеку, але, водночас, передбачає підтримку складнішої вказівникової структури:

```
struct stack {
            int el;          // значення елемента
```

```
struct stack *next;} st // адреса наступного елемента st ‘head, *p1, *p2; //вказівник на «голову», допоміжні вказівники st* push(int a; st *cur)
// функція додавання елемента,
//a - значення, яке треба внести у стек //cur-вершина («голова») стека {st *p;
    // якщо стек порожній if(!cur)
    {
        //          створюємо          вершину
      cur=(st*)malloc(sizeof(st));
        cur->el=a;
        cur->next=NULL;
        return (cur);
    }
    else
    {
        // створюємо новий елемент, який стане вершиною стека
        p=(st*)malloc(sizeof(st));
        p->ei=a;
        p->next=cur;
        return (p);
    }
}
st* pop()          // видалення вершини стека
{st *p;
    if(head) // якщо в стеку є елементи
     { .// вершиною стає наступний елемент
          p=head->next;
          //знищуємо «голову»
          free(head);
          return (p);
     }
 else return (NULL);
 }
```

Сукупність операцій, що визначають структуру типу «стек», подана нижче.

- ✓ Операція ініціалізації.
- ✓ Операція включення елемента в стек.
- ✓ Операція виключення елемента зі стека.
- ✓ Операція перевірки: стек порожній / стек не порожній.
- ✓ Операція перевірки: стек переповнений / стек не переповнений (ця операція характерна для стека як відображення на масив).
- ✓ Операція читання елемента (доступ до елемента).

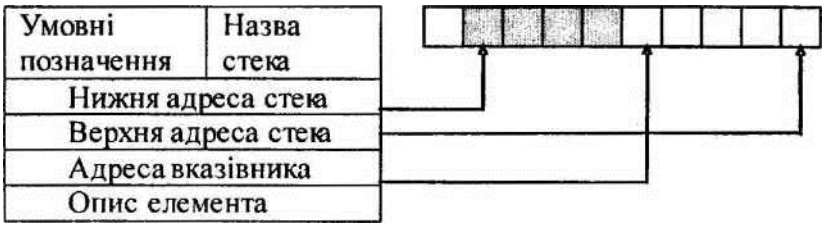
Є дві модифікації стека:

- ➢ вказівник перебуває на вершині стека, показуючи на перший порожній елемент;
- ➢ слот;
- ➢ вказівник вказує на перший заповнений елемент.

2.7.1. Дескриптор СД типу «стек»

Дескриптор СД типу «стек» містить:

- ➢ адреси початку та кінця стека,
- ➢ адресу вказівника,
- ➢ опис елементів



2.7.2. Області застосування СД типу «стек»

Стек використовується при перетворенні рекурсивних алгоритмів у нерекурсивні. Зокрема, за допомогою стека можна модифікувати алгоритм сортування Хоора.

Стек використовується при розробленні компіляторів.

Стеки вплинули й на архітектуру комп'ютера, послужили основою для стекових машин. У такого комп'ютера акумулятор виконаний у вигляді стека, що дозволяє розширити спектр безадресних команд, тобто команд, що не вимагають явного задання адрес операндів. Наслідком використання стека є збільшення швидкості опрацювання.

2.8. Сд типу «черга»

❖ ***Черга*** *- послідовність, у яку включають елементи* ***з*** *одного боку, а виключають - з іншого. Структура функціонує за принципом FIFO (надійшовший першим, обслуговується першим). Умовне позначення черги подане на рис 3.3.*

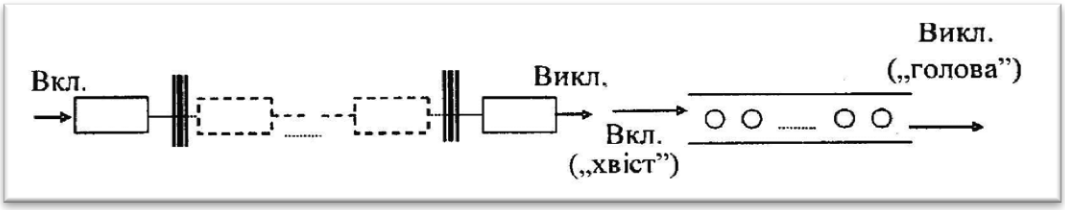


Рис. 1.5. СД типу «черга».

При реалізації черги розглядаються черга як відображення на масив (напівстатична реалізація) і черга як відображення на список.

Відображення на масив:

```
int a[100]; // оголошення черги
int n=0; // індекс останнього елемента у черзі
int current=0; H індекс «голови», діє принцип FIFO
void pop () // функція видобування (вилучення) елемента з черги
{
    if (n!=0)
    {
        current++; // зсунули «голову» на один елемент вперед printf("%d%, a[current]);
    }
}
void push () // функція додавання елемента до черги
{
    if (leurrent) // черга порожня
    {
        scanf("%d%, &a[current]);
    }
    else if (n<99)
    {                   // збільшили кількість елементів
        n++;            // додали елемент у кінець черги
        scanf("%d%, &a[n]);
    }
}
```

Відображення на список: struct cherga {

```
        int el;                 //значення елемента
    struct charga *next;} ch // адреса наступного елемента ch 'head, *p1, *p2;     //вказівник на «голову», допоміжні вказівники
ch* push(int a; ch *cur)
// функція додавання елемента,
//a - значення, яке треба внести у чергу // сиг - останній елемент черги {ch *p;
// якщо черга порожня
if(!cur)
{           // створюємо вершину
cur=(ch*)mal!oc(sizeof(ch));
cur->el=a;
cur->next=NULL;
retum(cur);
}
else
{           // створюємо новий елемент та додаємо його у кінець
p=(ch*)mailoc(sizeof(ch)); p->el=a; cur->next=p; return (p);
}
}
ch* pop() // видалення вершини черги {ch *p;
if(head) // якщо в черзі є елементи
{           .// вершиною стає наступний елемент
p=head->next;
// знищуємо «голову»
free(head);
return(p);
}
else return (NULL);
}
```

Сукупність операцій, що визначають структуру типу «черга» подана нижче.

- ✓ Операція ініціалізації.
- ✓ Операція включення елемента в чергу.
- ✓ Операція виключення елемента із черги.
- ✓ Операція перевірки: черга порожня / черга не порожня.
- ✓ Операція перевірки: черга переповнена / черга не переповнена.

Області застосування СД типу «черга»

Черга використається при передаванні даних з оперативної у вторинну пам'ять (при цьому відбувається процедура буферизації: накопичується блок і передається у вторинну пам'ять). Наявність буфера забезпечує незалежність взаємодії процесів між виробником і споживачем (рангування задач користувача). Задачі розділяються за пріоритетами:

- *задачі, розв 'язувані в режимі реачьного часу (вищий пріоритет) (черга I);*
- *задачі, розв'язувані в режимі розділення часу (черга 2);*
- *задачі, розв 'язувані в пакетному режимі (фонові задачі) (черга 3).*

Доступ до елементів черги здійснюється послідовно.

Контрольні запитання до модуля 1.

1. Опишіть структури даних типу «масив».
2. Перерахуйте набір допустимих операцій для структури даних типу «масив».
3. Поняття дескриптора. Приклад.
4. Дескриптор структури даних типу «масив».
5. Структури даних типу «запис» (прямий декартовий добуток).

6. Структури даних типу «таблиця».
7. Класифікація структур даних типу «таблиця».
8. Класифікація операцій над структурами даних типу «таблиця».
9. Набір допустимих операцій для структури даних типу таблиця».
10. Структури даних типу «стек».
11. Сукупність операцій, що визначають структуру типу «стек».
12. Дескриптор структури даних типу «стек».
13. Області застосування структури даних типу «стек».
14. Структури даних типу «черга».
15. Сукупність операцій, що визначають структуру типу «черга».
16. Дескриптор структури даних типу «черга».
17. Області застосування структури даних типу «черга».
18. Області застосування СД типу «дек».

Тести для закріплення матеріалу

1. Перерахувати допустимі операції над масивами:
а) операція доступу;
б) операція розіменування;
в) операція присвоєння;
г) операція індексування;
д) операція ініціалізації.

2. Перерахувати дані, що містить дескриптор масиву:
а) ім’я;
б) умовне позначення;
в) адреса першого елемента;
г) адреса останнього елемента;
д) індекс першого елемента;
е) індекс останнього елемента.

3. Перерахувати операції над множинами:
а) перетин;
б) транспонування;
в) об'єднання;
г) різниця;
д) сума;
е) перевірка належності.

4. Дати визначення структур даних типу «запис»:
а) послідовність елементів, які, в загальному випадку, можуть бути одного типу;
б) послідовність елементів, які, в загальному випадку, можуть бути різного типу;
в) послідовність декількох множин елементів;
г) послідовність декількох масивів.

5. Перерахувати допустимі операції над записами:
а) операція доступу;
б) операція розіменування;
в) операція присвоєння;
г) операція індексування;
д) операція ініціалізації'.

6. Типи таблиць:
а) невпорядкована таблиця;
б) впорядкована таблиця;
в) умовно-впорядкована таблиця;
г) хеш-таблиця;
д) відображення на множину;
е) відображення на масив; є)

відображення на список.

7. За визначенням вибрати операцію над таблицею: операція, у якої вхідним параметром є об'єкти відповідної структури, вона повертає результати іншого типу:

а) конструктори;
б) деструктори;
в) модифікатори;
г) спостерігачі;
д) ітератори.

8. За визначенням вибрати операцію над таблицею: опера тор доступу до вмісту об'єкту частинами у певному порядку:

а) конструктори;
б) деструктори;
в) модифікатори;
г) спостерігачі;
д) ітератори.

9. За визначенням вибрати операцію над таблицею: операція, яка руйнує об'єкти розглянутої структури:

а) конструктори;
б) деструктори;
в) модифікатори;
г) спостерігачі;
д) ітератори.

10. За визначенням вибрати операцію над таблицею: операція, яка створює об'єкти розглянутої структури:

а) конструктори;
б) деструктори;
в) модифікатори;
г) спостерігачі;
д) ітератори.

11. Перерахувати допустимі операції над таблицями:

а) операція ініціалізації;
б) операція присвоєння;
в) операція включення елемента в таблицю;
г) операція виключення елемента з таблиці;
д) операції-предикати;
е) операція порівняння;
є) читання елемента за ключем.

12. Принцип LIFO діє для:

а) черги;
б) стека;
в) списку;
г) слота.

13. Принцип FIFO діє для:

а) черги;
б) стека;
в) списку;
г) слота.

ЗМІСТОВИЙ МОДУЛЬ 2. ЗВ'ЯЗАНИЙ РОЗПОДІЛ ПАМ'ЯТІ. ХЕШУВАННЯ ДАНИХ. НЕЛІНІЙНІ СТРУКТУРИ ДАНИХ: ДЕРЕВА І ГРАФ.

Тема 1. Зв'язаний розподіл пам'яті. Хешування даних. Хеш-функція, алгоритми хешування, динамічне хешування.

2.1. Сд типу вказівник

Вказівний тип займає проміжне положення між скалярними й структурними типами: з одного боку значення вказівного типу є атомарним (неподільним), а з іншого, ці типи визначаються через інші (у тому числі й структурні) типи.

Type <тип вказівиика> = ^{Л} <тип об'єкту, що вказується>

(цей тип дозволяє використати базовий тип перед описом)

Type PtrType = ^{A}BaseType;

BaseType = record

x, y: real;

end;

Var A: PtrType; {A - змінна статичного типу, значенням якої є адреси розташування в пам'яті конкретних значень заданого типу}

B: BaseType;

C: ^{A}PtrType;

Змінній А можна присвоїти адресу якоїсь змінної, для чого використаємо унарну операцію взяття вказівника: A = B.

На фізичному рівні вказівник займає два слоти: у першому слоті перебуває адреса сегмента, у другому - адреса зсуву.

Операції над вказівним типом:

1) операція порівняння на рівність: = (рівність, якщо співпадають адреси сегмента й зсуву);

2) операція порівняння на нерівність: <>;

3) операція доступу: B.X = B.X **+ C.**

Серед всіх можливих вказівників виділяється один спеціальний вказівник, що нікуди не вказує. Тобто у пам'яті виділяється одна адреса, у яку не записується жодна

На це місце в пам'яті й посилається такий порожній або "нульовий" вказівник, що позначається **nil** на мові Паскаль або NULL на мові С. Вказівник **nil** вважається константою, сумісною з будь-яким вказівним типом, тобто це значення можна присвоювати будь-якому вказівному типу.

2.2. Статичні й динамічні змінні

2.2.1. Відмінності між статичними та динамічними змінними

Дотепер ми розглядали змінні, які розміщаються в пам'яті відповідно до цілком певних правил, а саме, для локальних змінних, описаних у підпрограмах, пам'ять приділяється при виклику підпрограми; при виході з неї ця пам'ять звільняється, а самі змінні припиняють існування. Глобальним змінним програми пам'ять приділяється на початку її виконання; ці змінні існують протягом усього періоду роботи програми. Іншими словами, розподіл пам'яті у всіх цих випадках відбувається повністю автоматично. Змінні, пам'ять під які розподіляється подібним чином, називаються **статичними.**

Рис. 2.1. Розподіл памяті для динамічних змінних

New (<ім'я постання>): point; — процедура призначена для створення динамічних змінних певного типу (інакше кажучи, для відведення пам'яті в купі для зберігання значень динамічної змінної).

Dispose (<ім я посилання>): point; — процедура використовується для звільнення пам'яті, відведеної за допомогою процедури **New.**

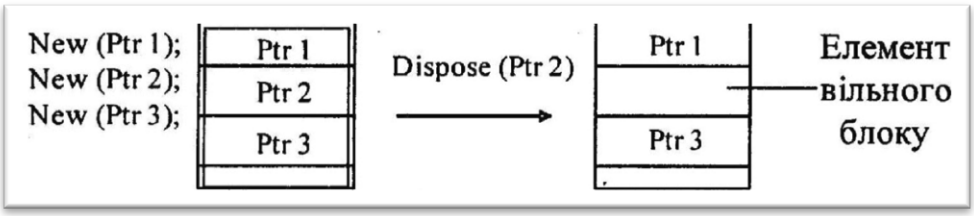
MaxAvail: longint; - функція повертає максимальний розмір (у байтах) безперервної вільної ділянки купи. Застосування цієї процедури необхідне для контролю динамічної пам'яті при реалізації операції включення.

Наприклад: if MaxAvail > SizeOf (BaseType) then {генеруємо об'єкт}

Створимо три об'єкти, які розташуються в пам'яті послідовно (без фрагментарності), а потім знищимо другий об'єкт. У результаті виникне фрагментарність, якої треба уникати.

MemAvail: longint; — функція повертає загальну кількість вільної пам'яті.

Щоб перевірити, є чи фрагментарність, треба порівняти результати застосування функцій ***MaxAvail*** і ***MemAvail*** - вони повинні збігатися.



2.3. Класифікація сд типу «зв'язний список»

Усі вищерозглянуті структури, що реалізувалися як відображення на масив, мають певні недоліки, тобто вони малоефективні при розв'язуванні деяких задач. До таких недоліків можна віднести наступне.

- ✓ Точно невідомо, скільки елементів буде мати певна структура, тобто не можна зробити оцінку.
- ✓ Якщо ми розглядаємо якусь послідовність елементів у послідовній пам'яті

x_1 $\mathbf{x_2.x_n}$ і необхідно включити який-небудь новий елементну цю послідовність, то

ми повинні здійснити масову операцію зсуву всіх елементів, що перебувають за тим елементом послідовності, після якого ми хочемо включити новий елемент. Після цього вставимо цей новий елемент **x** на місце, що звільнилося. Отже, тут проявляється властивість фізичної суміжності

Рис. 2.2 Додавання нового елемента в список

Позбутися фізичної суміжності можна, якщо елементи будуть мати не тільки дані, але й вказівники. У цьому випадку елементи можуть бути хаотично розкидані по оперативній пам'яті, а логічна послідовність буде забезпечуватися одним або декількома вказівниками.

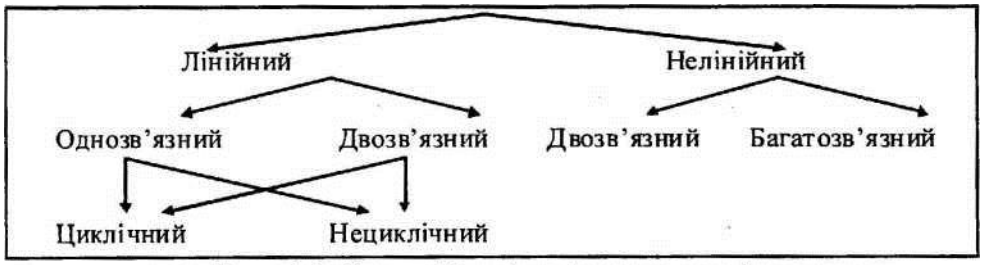


Рис. 4.3. Класифікація зв'язних списків.

2.4. Поняття хеш-функції

Для прискорення доступу до даних у таблицях можна використовувати попереднє впорядковування таблиці відповідно до значень ключів.

При цьому можуть бути використані методи пошуку у впорядкованих структурах даних, наприклад, метод половинного розподілу, що істотно скорочує час пошуку даних за значенням ключа. Проте при додаванні нового запису вимагається перевпорядкувати таблицю. Втрати часу на повторне впорядковування таблиці можуть значно перевищувати виграш від скорочення часу пошуку. Тому для скорочення часу доступу до даних у таблицях використовується так зване **випадкове впорядковування** або **хешування.** При цьому дані організовуються у вигляді таблиці за допомогою хеш- функції А, яка використовується для обчислення адреси за значенням ключа (рис. 5.1).

Ідеальною хеш-функціоо є така хеш-функція, яка для будь-яких двох неоднакових ключів повертає неоднакові адреси.

Підібрати таку функцію можна у випадку, якщо всі можливі значення ключів наперед відомі.

Така організація даних має назву «досконале хешування». У разі наперед невизначеної множини значень ключів і обмеженої довжини таблиці підбір досконалої функції важко здійснити. Тому часто використовують хеш-функції, які не гарантують виконання умови.

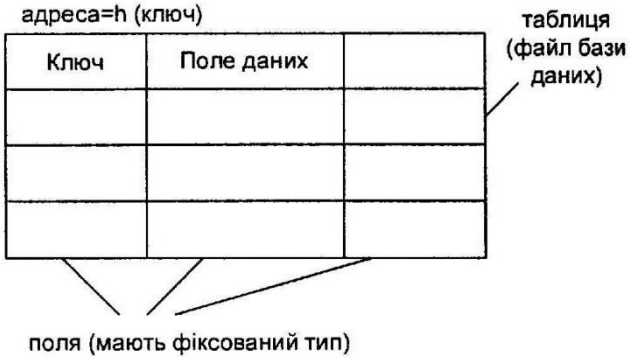


Рис. 2.3. Структура хеш-таблиці

Приклад 5.1. Розглянемо приклад реалізації недосконалої хеш-функції на мові TurboPascal. Припустимо, що ключ складається із чотирьох символів. При цьому таблиця має діапазон адрес від 0 до 10000.

```
Function hash (key: string[4]): integer; varf: longint; begin
f:=ord (key[1 ]) - ord (key[2]) + ord (key[3]) -ord (key[4]);
{обчислення функції за значенням ключа} f:=f+255*2;
{ поєднання початку області значень функції з початковою адресою хеш-таблиці (a=1)}
f:=(f*10000) div (255*4);
{ поєднання кінця області значень функції з кінцевою адресою
хеш-таблиці (a=10 000)}
hash:=f
end;
```

При заповненні таблиці виникають ситуації, коли для двох неоднакових ключів функція обчислює одну і ту саму адресу. Даний випадок має назву **«колізія», а** такі ключі називаються **ключами-синонімами.**

5.2. АЛГОРИТМИ ХЕШУВАННЯ

Для визначеності вважатимемо, що хеш-функція **h{K)** має не більше як **М** різних значень і, що ці значення задовольняють умову

О < *h(K) <M*

для всіх ключів **К.**

Теоретично неможливо так визначити хеш-функцію, щоб вона створювала випадкові дані з невипадкових реальних файлів. Але на практиці неважко зробити достатньо хорошу імітацію випадковості, використовуючи прості арифметичні діїРозглянемо, наприклад, випадок десятизначних ключів на десятковому комп'ютері. Сам собою напрошується наступний спосіб вибору хеш-функції: встановити Мрівним, скажімо, 1000, а як **h(K)** узяти три цифри, вибрані приблизно з середини 20-значного добутку **К*К.** Здавалося б, це повинно давати досить рівномірний розподіл значень між 000 і 999 з незначною ймовірністю часткою колізій. Насправді, експерименти з реальними даними показали, що такий метод **«серединних квадратів»** непоганий за умови, що ключі не містять багато лівих або правих нулів підряд.

З'ясувалося, проте, що існують надійніші йі простіші способи хеш-функцій.

Метод ділення особливо простий: використовується остача частка від ділення

на **М**

h(K)=K mod M

У цьому випадку, очевидно, що деякі значення Мбудуть кращі за інших.Наприклад, якщо М-парне число, то значення h{K) буде парним при парному К, інакше—непарним; часто це приводить до значних зсувів даних. Зовсім погано брати А/рівним розрядності машинного слова, оскільки тоді h(K) дає нам праві значущі цифри К (К mod М не залежить від інших цифр). Аналогічно, Мне має бути кратним 3, бо буквенні ключі, що відрізняються один від одного лише регістром, могли б дати значення функції, різниця між якими кратна 3. (Причина криється в тому, що 10« mod 3= **4п** mod 3= 1.) Взагалі ми хотіли б уникнути значень М, які діляться на **rk** ± **a,** де **k** і **а** —невеликі числа, а **г** - «основа системи числення» для множини літер, що використовуються (зазвичай **г** =64, 256 і 100), оскільки остача від ділення на такі значення Мвиявляється суперпозицією цифр ключа.

Мультиплікативиа схема хешування полягає в заданні послідовність випадкових цілих чисел за формулою

X_{+1} АХ(mod **М).**

Для машинної реалізації найзручнішим є **М= 2^g,** де *g* - розрядність машинного слова. Алгоритм подаємо нижче.

1 .Вибрати $\mathbf{X_O}$ - довільне непарне число.

2. Визначити коефіцієнт **X** = 8/ ± 3, де / - довільне ціле додатне число.

3.Знайти добуток ЯX_0 що містить не більше 2**g** значущих розрядів.

4. Взяти **g** молодших розрядів в якості $\mathbf{X_г}$

5.Знайти дріб х, **=XJ2^g** в інтервалі (0,1).

- Присвоїти А' - $\mathbf{X_{\{}}$.
- Повторити з п. 3.

Хороша хеш-функція повинна задовольняти дві вимоги:

19. її обчислення має бути дуже швидким;

20. вона повинна мінімізувати число колізій.

Властивість (а) частково залежить від особливостей машини, а властивість (Ь)- від характеру даних. Якби ключі були дійсно випадковими, можна було б просто виділити декілька бітів і використовувати їх для хеш-функції, але на практиці, щоб задовольнити (Ь), майже завжди потрібна функція, залежна від усіх бітів.

2.5. Динамічне хешування

2.5.1. Означення динамічного хешування

Описані вище методи хешування є статичними, тобто спочатку виділяється деяка хеш-таблиця, під її розмір підбираються константи для хеш-функції. На жаль, це не надається для завдань, у яких розмір бази даних часто змінюється. У міру зростання бази даних можна

- ✓ *користуватися початковою хеги-функцією, втрачаючи продуктивність через зростання колізій;*
- ✓ *вибрати хеш-функцію «із запасом», що спричинить невиправдані втрати дискового простору;*
- ✓ *періодично змінювати функцію, перераховувати всі адреси; це забирає дуже багато ресурсів і виводить з ладу базу на деякий час.*

Існує техніка, що дозволяє динамічно змінювати розмір хеш-структури. Це - динамічне хешування. Хеш-функція генерує так званий псевдоключ, який використовується лише частково для доступу до елементу. Іншими словами, генерується досить довга бітова послідовність, яка має бути достатня для адресації всіх потенційно можливих елементів. У той час, як при статичному хешуванні було б треба дуже велику таблицю (яка зазвичай зберігається в оперативній пам'яті для прискорення доступу), тут розмір зайнятої пам'яті прямо пропорційний кількості

елементів в базі даних. Кожен запис в таблиці зберігається не окремо, а в якомусь блоці ("bucket"). Ці блоки збігаються з фізичними блоками на пристрої зберігання даних. Якщо в блоці немає більше місця, щоб вміщати запис, то блок ділиться на два, а на його місце ставиться вказівник на два нові блоки.

Завдання полягає в тому, щоб побудувати бінарне дерево, на кінцях гілок якого б)ли б вказівники на блоки, а навігація здійснювалася б на основі псевдоключа. Вузли дерева можуть бути двох видів: вузли, які показують на інші вузли або вузли, які показують на блоки. Наприклад, нехай вузол має такий вигляд, якщо він показує на

Zero	Null
Bucket	Вказівник
One	Null

Якщо ж він вказуватиме на два інші вузли, то він матиме такий вигляд:

Zero	Адреса a
Bucket	Null
One	Адреса b

Спочатку є тільки вказівник на динамічно виділений порожній блок. При додаванні елемента обчислюється псевдоключ, і його біти по черзі використовуються для визначення місця розташування блоку.

2.5.2. Розширюване хешування

Розширюване хешування близьке до динамічного. Цей метод також передбачає зміну розмірів блоків у міру зростання бази даних, але це компенсується оптимальним використанням місця. Оскільки за один раз розбивається не більш як один блок, накладні витрати досить малі.

Замість бінарного дерева розширюване хешування передбачає список, елементи якого посилаються на блоки. Самі ж елементи адресуються за деякою кількостю і бітів псевдоключа. При пошуку береться / бітів псевдоключа і через список (сіігес- Югу) знаходиться адреса шуканого блоку. Додавання елементів відбувається складніше. Спочатку виконується процедура, аналогічна до пошуку. Якщо блок неповний, додається запис у нього і в базу даних. Якщо блок заповнений, він розбивається на два, записи перерозподіляються за описаним вище алгоритмом. У цьому випадку можливе збільшення числа бітів, необхідних для адресації. Тоді розмір списку подвоюється і кожному новому створеному елементу присвоюється вказівник, який містить його батько. Отже, можлива ситуація, коли декілька елементів показують на один і той самий блок. Зауважимо, що за одну операцію додавання перераховуються значення не більш, ніж одного блоку. Видалення здійснюється за таким самим алгоритмом, тільки навпаки. Блоки, відповідно, можуть бути склеєні, а список - зменшений у два рази.

Отже, основною перевагою розширюваного хешування є висока ефективність, яка не знижується при збільшенні розміру бази даних. Окрім цього, розумно витрачається місце на

пристрої зберігання даних, оскільки блоки виділяються тільки під реальні дані, а список вказівників на блоки має розміри, мінімально необхідні для адресації заданої кількості блоків. За ці переваги розробник розплачується додатковим ускладненням програмного коду.

Тема 2. Визначення дерева, «бінарне дерево», алгоритми проходження дерев углиб та вшир.

2.6. Дерево

26.1. Визначення дерева

- ***Дерево*** — скінченна непорожняТ, гцо складається з одного й більше вузлів таких, що виконуються наступні умови:
  - ✓ є один спеціально позначений вузол, який називається коренем дерева;
  - ✓ інші вузли (крім кореня) містяться в т>=0 попарно не пересічних множинах $T_г$ T_v .... Т , кожна з яких у свою чергу, є деревом. Дерева

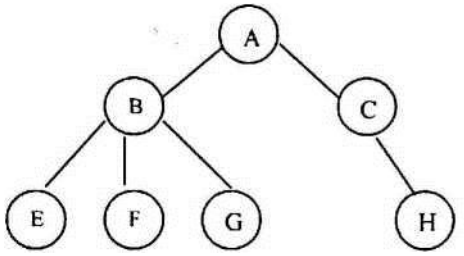
$T_Г$ T_2 . $T_т$ називаються піддеревами такого кореня.

Дерева зображаються такими способами:

• графічно,

• за допомогою множин,

• як модифікація багатозв язних списків.



A: T_1 = {B, E, F, G}
T_2 = {C, H}
B: T_{11} = {E}
T_{12} = {F}
T_{13} = {G}
C: T_{21} = {H}

а) графічний б) за допомогою множин

Рис. 2.4. Способи зображення дерева.

Якщо підмножини **T_x, $T_г$, ..., $T_т$** упорядковані, то дерево називають **упорядкованим.** Якщо два дерева вважаються рівними й тоді, шли вони відрізняються порядком, то такі дерева називаються **орієнтованими деревами.** Кінцева множина непересічних дерев називається **лісом** (рис. 6.2).

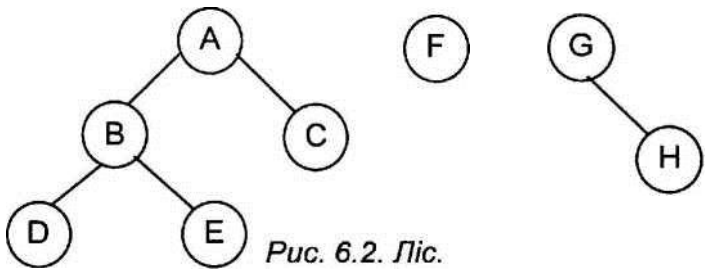


Рис. 6.2. Ліс.

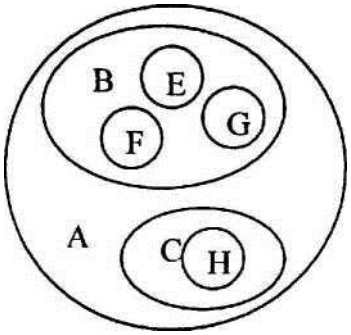
2.6.2. Бінарне дерево

Бінарне дерево - скінченна множина елементів, що може бути порожньою, яка складається з кореня й двох непересічних бінарних дерев, причому піддерева впорядковані: ліве піддерево й праве піддерево.

ь с> * о ь

Кількість підмножин для заданого вузла називається **ступенем вузла.** Якщо така кількість дорівнює нулю, то вузол є листом. Максимальний ступінь вузла в дереві - **ступінь дерева. Рівень вузла** - довжина шляху від кореня до розглянутого вузла. Максимальний рівень дерева - висота дерева.

Структуру дерева можна зображати й за допомогою способів, поданих на рис 6.3.



Вкладені множини
Дужкова форма: (A (B (E) (її) (в)) (C (H)))
Десяткова форма Дьюі: A—I;
B-I.I; C— 1.2;
E-1.1.1; P~1.1.2; C—1.1.3; H-1.2.1

Рис. 2.6. Способи подання дерев.

2.6.3. Подання дерев у зв'язній пам'яті комп'ютера

Розрізняють три основні способи подання дерев у зв'язній пам'яті: стандартний, інверсний, змішаний. Розглянемо ці способи для дерева зображеного на рис. 6.4.

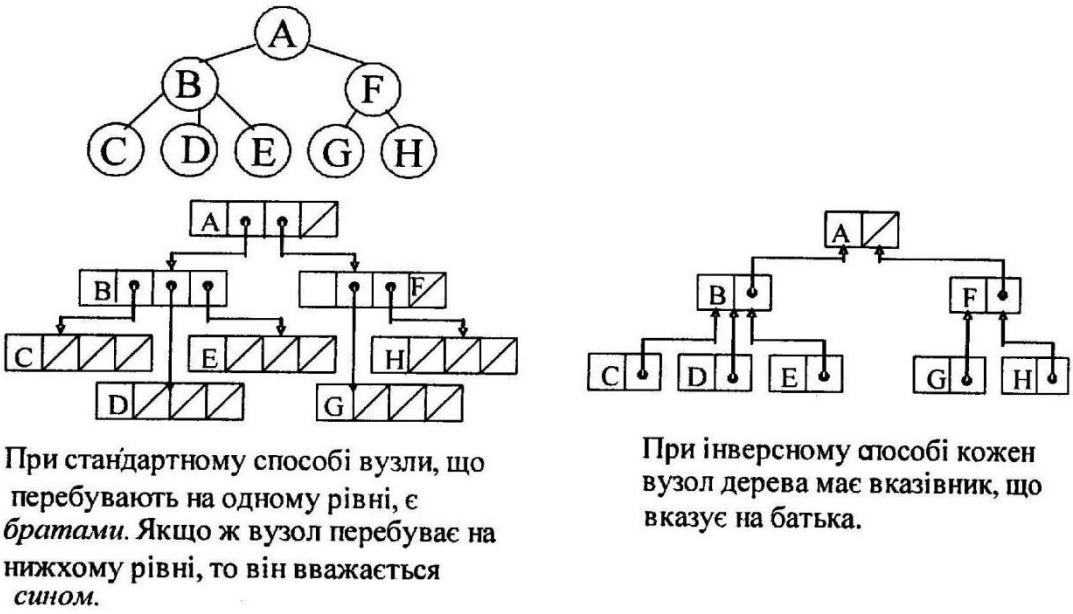


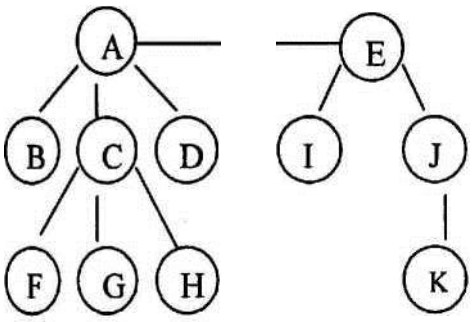
Рис. 2.7. Стандартний та інверсний способи подання дерев

Якщо ж говорити про змішаний спосіб подання дерева у зв'язній пам'яті, то тут, як видно з назви, кожний вузол включає вказівники, що вказують як на синів, так і на батька.

Функція побудови дерева стандартним способом: struct btree

```
// структура дерева {
        int el;
        struct btree *l,*r;
    }; // вказівники на лівого та правого сина //вказівники на корінь
дерева та поточний елемент struct btree *root, *cur;
//функція додавання елемента у дерево
struct btree * insert (struct btree *c, int k, struct btree * new1)
        {
struct btree *p Д int a;
        if(!c) //поточний елемент є порожнім (відсутнім)
        {
            c=(struct btree*) malloc (sizeof(struct btree));
            c->el=k;
            c->l =NULL;
            c->r =NULL;
            //якщо задано значення a, то дерево вже містить вузли // і ми
        здійснюємо прив'язку створенного елемента
            // як лівого (a=1) або правого (a=2) сина if (a==1) new1->l=c; if (a==2)
        new1->r=c;
```

```
            return (c);
        }
        else
        //спускаємося далі по дереву
        {
        new1=c;
            if(c->el>k)
            //переходимо до лівого сина
            {
                c=c->I; a=1; p=insert (c,k,new1);
            }
            else
            //переходимо до правого сина
            {
                c=c->r; a=2; p=insert (c,k,new1);
            }
        }
    }
    void main()
    //виклик функції побудови дерева
    {
    int k,n=5,l; scanf("%d",&k); root=insert(NULL,k,cur);
for(i=2;i<=n;i++)
        {
            scanf("%d",&k);
            cur=insert(root,k,cur);
        }
    }
```



2.6.4. Алгоритми проходження дерев углиб і вшир

При проходженні вглиб зображеного дерева, список його вершин, записаних у порядку їхнього відвідування, буде виглядати так

A, B, C, ¥, *в,* H, *B,* E, I, *5,* K.

Алгоритм проходження дерева вглиб

спорожній стек S>;

спройти корінь і включити його в стек S>;

while <стек S не порожній> do

begin

{нехай P - вузол, що перебуває у вершині стека S} if <не всі сини вузла P пройдені»

then спройти старшого сина й включити його в стек S> else begin

свиключити з вершини стека вузол P>; if <не всі брати вершини P пройдені»

then спройти старшого брата й включити його в стек S>

end;

end;

При проходженні зображеного дерева вшир (по рівнях), список його вершин, записаних у порядку їхнього відвідування, буде таким:

A, B, C, D, E, F, G, H, I, J, K.

Алгоритм проходження дерева вширину: свзяти дві черги 01 і 02»; спомістити корінь у чергу 01»; while <01 або 02 не порожня» do begin

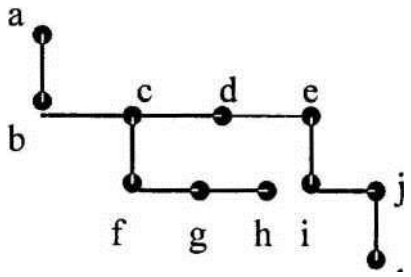
if <01 не порожня» then
{P - вузол, що перебуває в голові черги 01} begin
свиключити вузол із черги 01 і пройти його»; спомістити всі вузли, що відносяться до братів цього вузла P, у чергу 02»; end
else <01=02; *o2=o*
end;

*6.*1.5. Подання дерев у вигляді бінарних

Між деревами загального виду (вузол дерева може мати більше двох синів) і бінарними деревами існує взаємно однозначна відповідність, тому бінарні дерева часто використають для подання дерев загального виду.

Для такого подання використають наступний алгоритм:

- зображуємо корінь дерева;
- по вертикалі зображуємо старшого сина цього кореня;
- по горизонталі вправо від цього вузла подаємо всіх його братів;
- пп. I, 2, 3 повторюємо для всіх його вузлів.

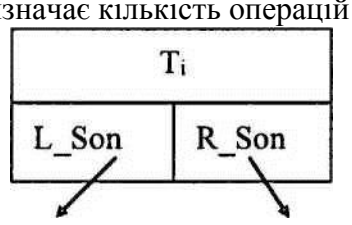


if n=0 then Tree:=nii else begin nl:=n div 2; nr:=n - nl - 1;
read (x); New (NewElement); with NewElement do begin Data:=x;
L_Son:=Tree (nl);
R_Son:=Tree (nr); end;
Tree:=NewElement;
end;
end;

Ефективність рекурсивного визначення полягає в тому, що воно дозволяє за допомогою кінцевого висловлення визначити нескінченну множину об'єктів.

2.6.5. Застосування бінарних дерев в алгоритмах пошуку

В однозв'язному списку неможливо використати бінарні методи, вони можуть використовуватися тільки в послідовній пам'яті. Однак, якщо використати бінарні дерева, то в такій зв'язній структурі можна одержати алгоритм пошуку зі складністю $0(\log_2 N)$. Таке дерево реалізується в такий спосіб; для будь-якого вузла дерева із ключем Г всі ключі в лівому піддереві повинні бути менші від Г, а в правому - більше **T.** У дереві пошуку можна знайти місце кожного ключа, рухаючись, починаючи від кореня й переходячи на ліве або праве піддерево, залежно від значення його ключа. З **п** елементів можна організувати бінарне дерево (ідеально збалансоване) з висотою не більшою ніж $\log_2 N$, що визначає кількість операцій порівняння при пошуку.



< Ti > Ti

Function Search (x: integer; t: ElPtr): EIRtr;
{поле Data замінимо на поле Key} varf: boolean; begin f:=false;
while (t<>nil) and not f do if $x=t^A$. Key then f:=true else
if$x>t^A$. Key then $t:=t^A$. R_Son else
$t:=t^A$. L_Son;
Search:=t;

end;

Якщо одержимо, що значення функції = nil, то ключа зі значенням ху дереві не знайдено.

```
Function Search (x: integer; t: ElPtr): ElPtr; begin
   S^A.key:=x;  while  t^A.key<>x  do  if
x>t^A.key then
          t:=t^A. R_Son else t:=t^A. L_Son;
   Search:=t;
```

2.6.6. Застосування бінарних дерев

1. **Дерево Хафмана** - двійкове дерево, листям якого є символи алфавіту, в кожній вершині якого зберігається частота символу (для листа) або сума частот його двох нащадків (називатимемо це число вагою вершини). При цьому виконується властивість: якщо відстань від кореня до вершини А"більша, ніж до вершини У, то вага **Х**не перевищує вагу У. Один із способів застосування дерев Хафмана - **алгоритми кодування (архівування) інформації (детальніше** див. розділ 10.3.3).

Код змінної довжини символа (змінний код у дереві Хафмана) - послідовність бітів, яку отримуємо при проходженні по ребрах від кореня до вершини із цим символом, якщо зіставити кожному ребру значення І або 0. У дереві Хафмана ребрам, що виходять із вершини до її нащадків присвоюються різні значення (вважатимемо далі, що ребро з І - ліве,

появи символів, будується двійкове дерево Хафмана, з нього отримуються відповідні кожному символу коди змінної довжини. Нарешті, знову здійснюється проходження початковим файлом, при цьому кожен символ замінюється на свій код у дереві. Отже, статичному алгоритму потрібні два проходи по файлу-джерелу, щоб закодувати дані. **Статичний алгоритм:**

```
{Ініціалізуємо двійкове дерево}
InitTree;
For i:=0 to Length(CharCount)
{Ініціалізуємо масив частот елементів алфавіту}
CharCount[i]:=0;
{Запускаємо перше проходження файлу}
While not EOP(Файл-джерело)
Begin
{Читаємо активний символ} i^eacI(Файл-джерело, С);
{Збільшуємо частоту його появи}
CharCount[C]= CharCount[C]+1;
End;
{За отриманими частотами будуємо двійкове дерево}
ReBuildTree;
WriteTree (Файл-приймач) {Записуємо у файл двійкове дерево} {Запускаємо друге проходження файлу}
While not EOP(Файл-джерело)
Begin
Read(Файл-джерело, С); {Читаємо поточний символ}
{Запускаємо другий прохід файлу} code=FindCodelnTree(C);
write( Файл-приймач, code); {Записуємо послідовність бітів у файл}
End
```

Процедура InitTree ініціал ізує двійкове дерево, онулюючи значення, ваги, і вказівники на батька і нащадків активного листка. Далі виконується перший прохід по джерелу даних з метою перевірки частотності символів, результати якого запам’ятовуються у масив CharCount. Розмірність цього масиву повинна дорівнювати кількості елементів алфавіту джерела. У загальному випадку, для стискання заданого комп’ютерного файлу його довжина повинна становити 256 елементів. Потім, відповідно до отриманого набору частот ,будуємо двійкове дерево ReBuildTree. Під час другого проходу переконуємо джерело відповідно до дерева і записуємо одержані послідовності бітів у стиснутий файл. Для того, щоб мати можливість

відновити стиснуті дані, необхідно в отриманий файл зберегти копію двійкового дерева WriteTree.

Динамічний алгоритм:

{Ініціалізуємо двійкове дерево}

InitTree;

For i:=0 to Length(CharCount)

{Ініціалізуємо масив частот елементів алфавіту}

CharCount[i]:=0;

{Запускаємо проходження файлу}

While not EOP(Файл-джерело)

Begin

КеасІ(Файл-джерело, С); {Читаємо активний символ}

If С немає в дереві then {Перевіряємо чи зустрічався символ раніше} Code:=Kofl "порожнього" символу & Asc(C)

{Якщо ні, то запам'ятовуємо код порожнього листка і ASCII код активного символу}

{Інакше запам'ятовуємо код активного символу} else

code:=Kofl C;

{Записуємо послідовність бітів у файл} write(Файл-приймач, code);

{Оновлюємо двійкове дерево символом}

ReBuildTree(C);

End;

Динамічний алгоритм дозволяє реалізувати однопрохідну модель стискання. Не знаючи реальної ймовірності появи символів у початковому файлі, програма поступово змінює двійкове дерево, з кожним символом, що зустрічається, збільшуючи частоту його появи е дереві та перебудовуючи зв'язки у самому дереві. Проте, стає очевидним, що, вигравши в кількості проходжень початковим файлом, ми втрачаємо у стисканні, оскільки у статичному алгоритмі реальні частоти символів були відомі із самого початку і довжини кодів цих символів ближчі до оптимальних, тоді як динамічний метод, вивчаючи джерело, поступово доходить до його реальних частотних характеристик. Але, оскільки динамічне двійкове дерево постійно модифікується новими символами, немає необхідності запам'ятовувати їх частоти заздалегідь - при розархівуванні програма, отримавши з архіву код символа, так само відновить дерево, як вона це робила при стисканні, і збільшить на одиницю частоту символа.

2.Будь-який *алгебраїчний* вираз містить змінні, числа, знаки операцій і дужки можна подавати *у вигляді бінарного дерева* (виходячи з бінарності цих операцій). При цьому знак операції міститься в корені, перший операнд - у лівому піддереві, а другий операнд- у правому. Дужки при цьому опускаються. У результаті всі числа й змінні виявляться в листках, а знаки операцій - у внутрішніх вузлах.

Розглянемо приклад: 3 (x - 2) + 4.

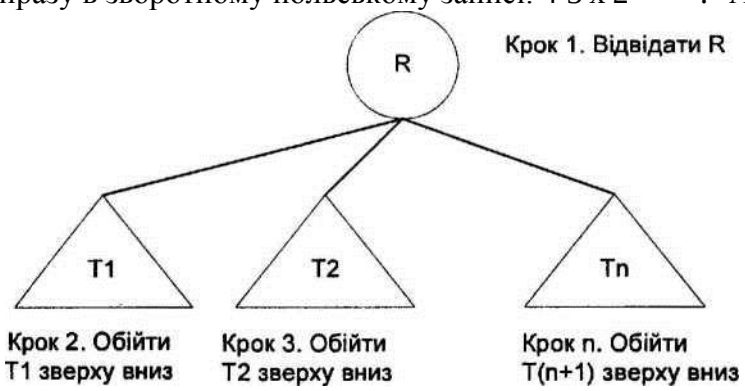
Звичною формою виразів є інфіксна, коли знак бінарної 4 операції записується між позначеннями операндів цієї операції, наприклад, х-2. Розглянемо запис знаків операцій після позначень операндів, тобто постфіксний запис, наприклад, х 2 -. Такий запис має також назву зворотного польського, оскільки його запропонував польський логік Ян Лукасевич.

Сформулюємо правило обчислення значень виразу у інфіксному записі: вираз проглядається справа наліво, виділяються перші 2 операнди перед операцією, ця операція виконується, і її результат — це новий операнд.

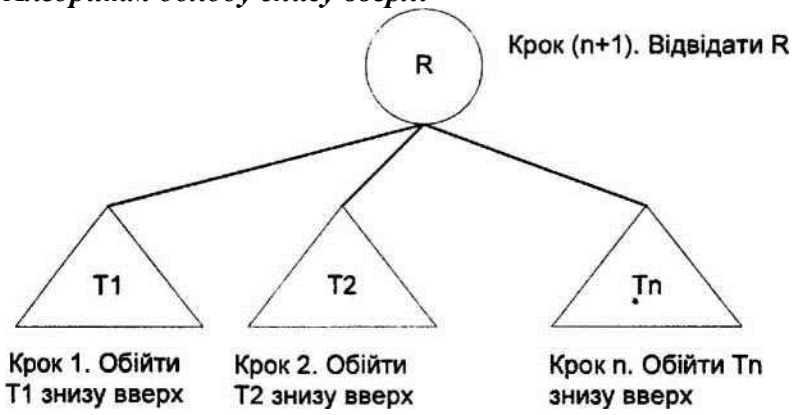
Правило обчислення значення виразу в зворотному польському записі: вираз проглядається зліва направо, виділяються перші 2 операнди перед операцією, ця операція виконується, і її результат - це новий операнд.

Запис виразу в інфіксній формі: 4 + 3 * х - 2.

Запис виразу в зворотному польському записі: 4 3 x 2 - * + . Алгоритм обходу зверху вниз:



Алгоритм обходу знизу вверх:



3. Класична програма із класу інтелектуальних: ***побудова дерев рішень.*** У цьому випадку не листкові (внутрішні) вузли містять предикати - запитання, відповіді на які можуть приймати значення “так” або “ні”. На рівні листків перебувають об’єкти (альтернативи, за допомогою яких розпізнаються програми). Користувач одержує запитання, починаючи з кореня, і, залежно від відповіді, спускається або на ліве піддерево, або на праве. Таким чином він виходить на той об’єкт, що відповідає сукупності відповідей на зппитання.

2.7. ВИДИ БІНАРНИХ ДЕРЕВ

2.7.1. Збалансоване дерево

У програмуванні ***збалансоване дерево*** в загальному розумінні цього слова - ***це*** такий різновид бінарного дерева пошуку, яке автоматично підтримує свою висоту, тобто кількість рівнів вершин під коренем, мінімальною. Ця властивість є важливою тому, ***що*** час виконання більшості алгоритмів на бінарних деревах пошуку пропорційний ***до*** їхньої висоти, і звичайні бінарні дерева пошуку можуть мати досить велику висоту в тривіальних ситуаціях. Процедура зменшення (балансування) висоти дерева виконується за допомогою трансформацій, відомих як обернення дерева, в певні моменти часу (переважно при видаленні або додаванні нових елементів).

Більш строге визначення збалансованих дерев було дане Г.Адельсон-Вельським та Є.Ландісом. *Ідеально збалансованим деревом* за Адельсон-Вельським та Ландісом є таке, у якого для кожної вершини різниця між висотами лівого та правого піддерев не перевищує одиниці. Однак, така умова доволі складна для виконання на практиці і може вимагати значної перебудови дерева при додаванні або видаленні елементів.

Тому було запропоноване менш строге визначення, яке отримало назву умови *АВЛ(AVL)-збалансованості* і говорить, що бінарне дерево є збалансованим, якщо висоти лівого та правого піддерев різняться не більше ніж на одиницю. Дерева, що задовольняють такі умови, називаються AVL-деревами. Зрозуміло, що кожне ідеально збалансоване дерево є також АВЛ-збалансованим, але не навпаки.

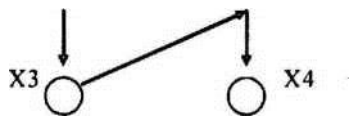
Наведемо програму додавання та знищення елемента у збалансованому дереві. Type node - record {запис для визначення дерева}

```
Key: integer;
Left, right: ref;
Bal: -1 ..1;                    {показує різницю висоти гілок дерева}
End;
procedure search(x: integer; var p: ref; varh: boolean);
var p1, p2: ref; {h = false}
begin
if p = nil then
begin                           {вузла немає у дереві; включити його}
new(p); h := true; with pД do begin
key := x; count := 1; left := nil; right := nil; bal := 0; end end else
ifx<pA.key then begin
search(x, pMeft, h);
if h then                       {виросла ліва гілка}
case pA.bal of
1: begin pA.bal := 0; h := false end ;
0: pA.bal :=-1;
-1: begin {балансування} p1:= pMeft; if p1A.bal = -1 then {однократний правий поворот} begin
//
```

Тема 3. Поняття графу, алгоритми проходження графу вглиб та вшир, топологічне сортування,пошук мостів.

2.3.1. Поняття графу

❖ *Граф — двійка О = (Х,***U),** *де* **X** - *множина елементів (вершин, вузлів), а* **U** - *бінарне відношення на множині* **X (U∩Xx** *X). Якщо* **|X| = п,** *то граф є* **скінченим.** *Елементи* **и** *називають або* **дугами,** *або* **ребрами.**



X = {XI, X2, X3, X4}
XxX = {(X1,X1), ...,(X4,X4)}
U = {(XI,X2), (XI,X3), (X2,X3), (X2,X4)}=> $UcXxX = X^2$

Якщо (X, X) - впорядкована пара, то такий граф називається **орієнтованим (орграфом,),** а елементи *1!* називаються **дугами.**

Якщо (X, X) = (X, X), то граф - **неорієнтований (неорграф),** а **елементи** *и* називаються **ребрами.**

Я: II —> Я - якщо задано таку функцію, то граф є **зваженим,** де *Я* - множина дійсних чисел, тобто відображення дуги на число.

Загалом: 0 *17я X x X*.

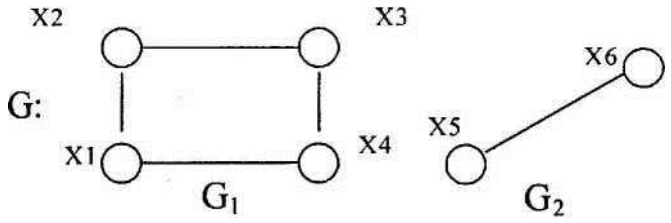
Для орієнтованого графу кількість ребер, що входять у вузол, називається **напівступенем вузла,** кількість ребер, що виходять з вузла - **напівступенем результату.** Кількість вхідних та вихідних ребер може бути довільною, у тому числі і нульовою. Граф без ребер називається **нуль-графом.**

Мультиграфом називається граф, що має паралельні (що сполучають одні і ті ж вершини) ребра, інакше граф називається **простим.**

Компонентами орграфу є: дуга, шлях, контур. **Шлях** - така послідовність дуг, у якій кінець кожної попередньої дуги збігається з початком наступної. Контур - кінцевий шлях, у якому початкова вершина збігається з кінцевої. Граф, у якому є контур, назвається циклічним. Контур

одиничної довжини називають петлею.

Компонентами неорграфу є, відповідно: ребро, ланцюг, цикл. Ланцюг-безперервна послідовність ребер між парою вершин неорієнтованого графу. Неорієнтований граф називають зв'язним, якщо будь-які дві його вершини можна з'єднати ланцюгом. Якщо ж граф - незв'язний, то його можна розбити на підграфи. Наприклад:



Слабка зв'язність - орграф замінюється неорієнтованим графом, який, у свою чергу, є зв'язним. **Однобічна зв'язність** - це така зв'язність, коли між двома вершинами існує шлях в одну або в іншу сторону. **Сильна зв'язність** — це зв'язність, коли між будь-якими двома вершинами існує шлях в одну й в іншу сторону.

Отже, багатозв'язна структура має такі властивості:

- на кожен елемент (вузол, вершину) може бути довільна кількість посилань;
- кожен елемент може мати зв'язок з будь-якою кількістю інших елементів;
- кожна зв'язка (ребро, дута) може мати напрямок і вагу.

Дерево - зв'язний граф без циклів.

Ліс (або ациклічний граф) - неограф без циклів (може бути і незв'язним).

Контур (каркас) зв'язного графу - дерево, що містить всі вершини графу. Визначається неоднозначно.

Редукція графу - контур з найбільшим числом ребер.

Цикломатичне (або циклічний ранг) число графу £=ш-я+с, де *n* - кількість вершин, *m* - кількість ребер, *c* - кількість компонент зв'язності графу.

Коциклічний ранг (або коранг) *E*-n-c.*

Неограф Сг є лісом тоді і тільки тоді, коли *$(C)^2-0.*

Неограф *О* має єдиний цикл тоді і тільки тоді, коли 6ХС)=1.

Контур неографу має 5 ребер.

Ребра графа, що не входять в контур, називаються хордами.

Цикл, що виходить при додаванні до контуру графу його хорди, називається фундаментальним щодо цієї хорди.

2.2. Подання графу в пам'яті комп'ютера

Графічний спосіб подання (якщо граф невеликий).

Використання матриць. Матриця легко описуванняється, й при аналізі характеристик графу можна використати алгоритми лінійної алгебри. Також використається подання графа у зв'язній пам'яті, у тому випадку, якщо значна кількість елементів у матриці дорівнює нулю (матриця не заповнена).

Одним із матричних способів подання графу є **матриця суміжності.** Нехай задано граф *Є = (X, Ц),\X\ = n.* Маємо матрицю *А* розмірності *пхп,* що називається *матрицею суміжності*, якщо елементи її визначаються так:

$$a_{ij} = \begin{cases} 1, (X_i X_j) \in U \\ 0, (X_i X_j) \notin U \end{cases}$$

Приклад 7.1. Нехай маємо граф, поданий рис. 7.1

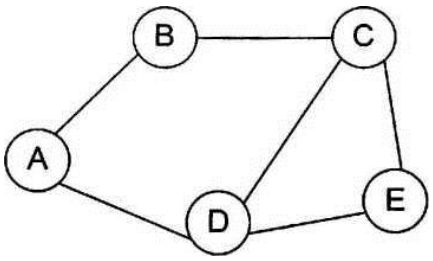


Рис. Приклад графу

Йому відповідає така матриця суміжності:

	A	в	с	D	E
A	0	1	0	1	0
B	0	0	1	1	0
C	0	0	0	1	1
D	I	0	1	0	1
E	0	0	1	1	0

Розглянемо застосування матричної алгебри для визначення характеристик графу. Вираз о $_{ik}L$ a_{t} означає, що між вузлами *і* і у є дві дуги, що проходять через вузол к, якщо значення виразу дорівнює True.

п

Вираз $У^{\wedge a}ік$ $^{Aa}к$ *і* означає, що завжди є шлях між цими вузлами довжиною 2, якщо вираз є істинним.

А Ь *А* - $A^{(7>}$ - логічні операції заміняються арифметичними. Тоді кожний елемент $a_{ч}$ буде подавати інформацію про те, є чи шлях з / в у (ї,у = 1, 2

Вираз $A^{(n)} = A^{(п\sim 11}$ Ь *А* означає, чи є шлях довжиною п між різними вузлами і. По діагоналі буде характеристика, чи є цикли (контури) у матриці.

небудь шлях між вузлами *і* та j. Алгоритм обчислення заданого виразу:

- Р = А;
- повторити 3, 4 (А=1,2
- повторити4 для/= 1,2, ...,*n*\
- повторити $P_u = P_{i\,i}$ V $(P_{ік}$ Ь $P_к),]$ = 1, 2,..., *n*...

У зв'язній пам'яті найчастіше подання графу здійснюється за допомогою **структур**

суміжності. Для кожної вершини множини *X* задається множина *M(X.)* відповідно до дуг її послідовників (якщо це орграф) або сусідів (для неорграфу). Отже, структура суміжності графу *Є* буде списком таких множин: < $M(X_1)$, $M(X_2)$,..., $M(X_n)$> для всіх його вершин.

Приклад Нехай маємо граф, поданий рис. 7.2 (вузли позначаємо у вигляді цифр: 1,2,..., и):

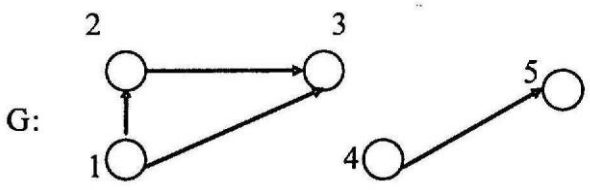


Рис. 2.3. Подання графу

Для неорграфу:
1:2,3;
2:1,3;
3:1,2;
4:5;
5:4.
Для орграфу:
1:2;
2:3 3:1 4:5 5:-.
Структуру суміжності можна реалізувати масивом з *n* лінійно зв'язаних списків:
Подання графу може вплинути на ефективність алгоритму. Часто запис алгоритмів на графах задається в термінах вершин і дуг, незалежно від подання графу. Наприклад, алгоритм визначення кількості послідовників вершин: С (Л0=О, *S* — кількість дуг.
S = 0;
∀ x ∈ X виконати:
початок
C(x)=0;
∀ t ∈ M(x) виконати: C(x) = C(x) + 1;
S = S + C(x);
кінець;

7.3. Алгоритми проходження графу

Алгоритм проходження може бути використаний як алгоритм пошуку, якщо вузлами графу є елементи таблиці.
Маємо граф G = (X, *U), X= {x1,x2,...,xn}...* Кожне проходження можна розглядати як певну послідовність. Максимальна кількість проходжень (перестановок) – *n!*.

7.3.1. Алгоритм проходження графу вглиб

Для пояснення принципу проходження графу вглиб скористаємося графом, поданим на рис. 7.3.

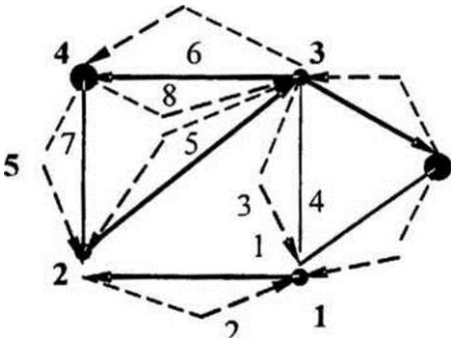


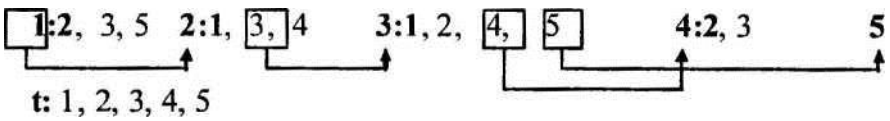
Рис. 2.4. Демонстрація проходження графу вглиб.

Цифри на ребрах графу позначають кроки відвідування. Будемо розрізняти відвідувані (¬) і не відвідувані (¬) вершини. Кожна вершина обходиться два рази. Якщо всі суміжні вершини пройдені, то повертаємося у попередню вершину.

Проходження графу вглиб здійснюється за такими правилами:

- перебуваючи у вершині *x,* треба рухатися до будь-якої іншої, раніше не відвіданої вершини (якщо така знайдеться), одночасно запам'ятовуючи дугу, по якій ми вперше потрапили до цієї вершини;
- якщо з вершини *x* ми не можемо потрапити до раніше не відвіданої вершини або такої взагалі немає, то ми повертаємося у вершину z, з якої вперше потрапили до x, і продовжуємо обхід вглиб з вершини *z.*

Визначимо списки суміжності для кожної вершини графу *G:*



Якщо граф G зв'язний, то описаний процес визначає проходження (обхід) графу *G.* Якщо ж граф G не зв'язний, то проходимо тільки одну з компонентів графу G, що містить початкову вершину. Якщо граф *G* є незв'язним, то для одержання повного обходу необхідно досягати такого результату у кожному зв'язному компоненті. За допомогою цього методу можна визначити кількість компонентів.

Для кожного вибору початкової вершини у зв'язному графі може бути отримане єдине проходження.

Алгоритм проходження графу вглиб буде виглядати так:

```
procedure ОБХІД-ВГЛИБ(р: вершина);
begin
відвідати вершину р;
for all q from множини вершин, суміжних до р, do if q ще не відвідана then
ОБХІД-ВГЛИБ(д) end end end; begin
        for all p from множини вершин G do
    if p ще не відвідувалась then ОБХІД-ВГЛИБ(р) end end end.
```

На мові Паскаль рекурсивна процедура проходження вглиб виглядає так:

```
procedure dfs(v:integer);
var
i: integer; begin
    used[v]:=true; {відзначити вершину як відвідану} for i:=1 to n do
        {якщо між вершинами є зв'язок та вершина не відвідана} if (a[v,i]=1)and(not
     used[i]) then
    dfs(i); {викликаємо процедуру з цією вершиною}
end;
```

Нерекурсивна функція проходження графу вглиб виглядає так:

```
procedure dfs(v: integer);
var
i: integer; found: boolean; begin
    used[v]:=true; {відзначили вершину як відвідану} inc(c); {збільшуємо
 кількість занесених у стек вершин}“ st[c] := v; {занесли у стек}
{доки стек не порожній} while c > 0 do begin
                v := st[c]; {беремо вершину з голови стеку} found := false; {шляху не знайдено}
                {проходимо по усіх вершинах з метою пошуку шляху з обраної
                вершини}
                for i := 1 to n do
                      if a[v, i] and not used [i] then begin
                          found ;= true; {знайшли шлях} break;
                            end;
                    {якщо шлях знайдений} if found then
                  begin
                        used [i] := true; inc(c); {додається у стек}
                    st[c] := i; p[i]:=v; end
                 else {вилучаємо вершину зі стека} dec(c); end;
end;
```

Змістовий модуль 3. Алгоритми пошуку. Загальна класифікація та принципи роботи.

Тема 1. Загальна класифікація алгоритмів пошуку. Лінійний пошук, двійковий (бінарний) пошук елементів в масиві, пошук методом Фібоначчі. М-блоковий пошук.

3.1. Алгоритми пошуку

Одна з тих дій, які найбільш часто зустрічаються в програмуванні – пошук. Існує декілька основних варіантів пошуку, і для них створено багато різноманітних алгоритмів.

Задача пошуку – відшукати елемент, ключ якого рівний заданому „аргументу пошуку". Отриманий в результаті цього індекс забезпечує доступ до усіх полів виявленого елемента.

3.1.1. Послідовний (лінійний) пошук

Найпростішим методом пошуку елемента, який знаходиться в неврегульованому наборі даних, за значенням його ключа є послідовний перегляд кожного елемента набору, який продовжується до тих пір, поки не буде знайдений потрібний елемент. Якщо переглянуто весь набір, і елемент не знайдений – значить, шуканий ключ відсутній в наборі. Цей метод ще називають методом повного перебору.

Для послідовного пошуку в середньому потрібно *N/2* порівнянь. Таким чином, порядок алгоритму – лінійний – *O(N)*. Якщо елемент знайдено, то він знайдений разом з мінімально можливим індексом, тобто це перший з таких елементів. Рівність *i=N* засвідчує, що елемент відсутній.

Єдина модифікація цього алгоритму, яку можна зробити, – позбавитися перевірки номера елементу масиву в заголовку циклу (*i<N*) за рахунок збільшення масиву на один елемент у кінці, значення якого перед пошуком встановлюють рівним шуканому ключу – *key* – так званий „бар'єр".

Якщо нема додаткових вказівок про розташування необхідного елемента, то природнім є послідовний перегляд масиву із збільшенням тієї його частини, де бажаного елемента не знайдено. Такий метод називається лінійним пошуком. Умови закінчення пошуку наступні.

1. Елемент знайдений, тобто a = x.
2. Весь масив проглянувши і збігу не знайдено.
3. Це дає нам лінійний алгоритм:
4. Звертаємо увагу, що якщо елемент знайдений, то він знайдений разом Із мінімально можливим індексом, тобто це перший з таких елементів. Очевидно, що закінчення циклу здійсниться, оскільки на кожному кроці значення *i* збільшується, і, отже, воно досягне за скінченну кількість кроків межі *N;* фактично ж, якщо збігу не було, це відбудеться після *N* кроків.

```
Int LinearSearch (int[] a, int N, int L, int R, int Key)
{
    for (int I = L;i<= R; i++)
  if (a[i] = = Key)
  return (i);
    return (-1 ); // елемент не знайдений
}
```

8.3. ДВІЙКОВИЙ (БІНАРНИЙ) ПОШУК ЕЛЕМЕНТА В МАСИВІ

Якщо у нас є масив, що містить впорядковану послідовність даних, то дуже ефективний двійковий пошук.

Змінні *Lb* і *Ub* містять, відповідно, ліву і праву межі відрізка масиву, де міститься потрібний елемент. Починаємо завжди з дослідження середнього елементу відрізка. Якщо шукане значення менше від середнього елемента, ми переходимо до пошуку у верхній половині відрізка, де всі елементи менші від тільки що перевіреного. Іншими словами, значенням *Ub* стає *(M-* 1) і на наступній ітерації ми працюємо з половиною масиву. Отже, в результаті кожної перевірки ми удвічі звужуємо область пошуку.

Блок-схема бінарного пошуку подана на рис. 8.1.

Функція, що реалізує бінарний пошук, має такий вигляд:

```
int BinarySearch (int a, int Lb, int Ub, int Key)
{
int M;
do
M = Lb + (Ub-Lb)/2;
//шукаємо середину
//відрізку
if (Key < a[M])
Ub = —M; //переходимо у ліву частину else if (Key > a[M])
Lb = ++M; //переходимо у парву частину else
return (M); if (Lb > Ub)
return (-1); //не знайдено while (1);
}
```

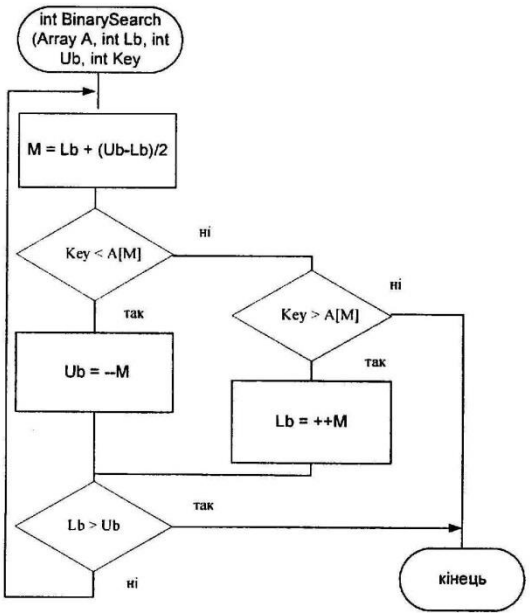


Рис. 3.1. Блок-схема функції бінарного пошуку за параметром

3.1.2. Пошук методом Фібоначчі

Він працює швидше, ніж бінарний пошук, оскільки замість операцій ділення, що застосовуються у попередньому методі, використовує операції додавання та віднімання. Суть методу полягає у визначенні наступного елемента для порівняння з числами Фібоначчі (звідси і назва). Зменшення індекса означає перехід по попереднього числа, збільшення - перехід до наступного.

Числа Фібоначчі формуються на основі додавання двох попередніх чисел, де перше і друге число дорівнюють 1. Тобто, можна скласти таку послідовність чисел:

1,1,2, 3, 5, 8,13,21,34...

Блок-схема пошуку методом Фібоначчі подана на рис. *int find_fibo(int strPar)*

```
int resFind, a[10];
int q=Fibonachi(1), p=Fibonachi(2), i=Fibonachi(3);
//функція, що визначає число Фібоначчі за номером
 int FindResult = -1;
do {
if(a[i]= strPar)
```

```
{
            FindResult = i; curSelRow = i+1; resFind = a[FindResult];
         return (resFind);
}
    else
        if                                          (if(a[i]>                                   strPar)
     if(q==0)
            {
                resFind = NULL; return (resFind);
            }
            else
                {i = i-q; p=q; q=p-q;} //перерахунок наступного числа
                else
                  if(p==1)
                {
                resFind = NULL;
                return (resFind);
                }
                else
                {
                i=i+q; p=p-q; q=q-p;
                }
             }
             while(resFind);
                if(FindResult == -1)
                    resFind = NULL;
             return (resFind);
       }
```

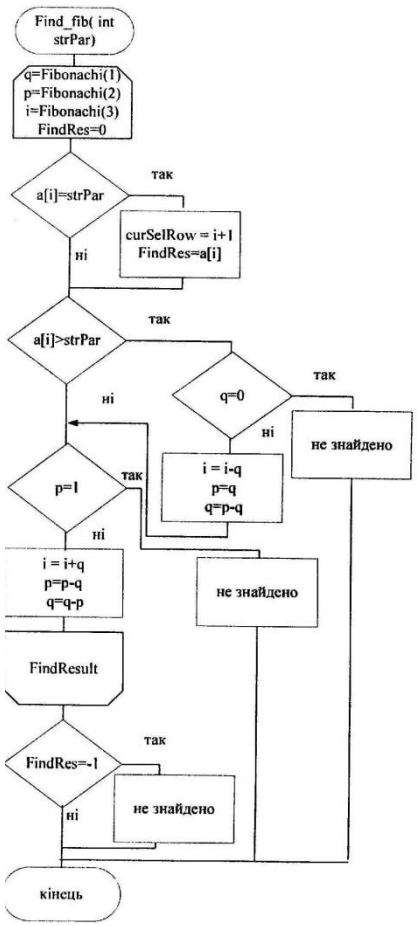


Рис. 3.2. Блок-схема функції пошуку Фібоначчі за параметром.

Алгоритм пошук може бути значно ефективнішим, якщо дані будуть впорядковані.

Іншим, відносно простим, методом доступу до елемента є метод бінарного (дихотомічного) пошуку, який виконується в явно впорядкованій послідовності елементів.

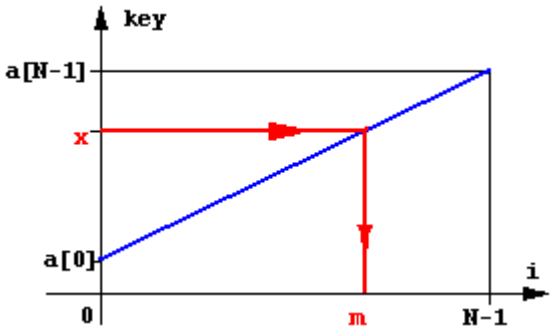
Оскільки шуканий елемент швидше за все знаходиться „десь в середині", перевіряють саме середній елемент: *a[N/2]==key*? Якщо це так, то знайдено те, що потрібно. Якщо *a[N/2]<key*, то значення *i=N/2* є замалим і шуканий елемент знаходиться „праворуч", а якщо *a[N/2]>key*, то „ліворуч", тобто на позиціях *0…i*.

Для того, щоб знайти потрібний запис в таблиці, у гіршому випадку потрібно *log2(N)* порівнянь. Це значно краще, ніж при послідовному пошуку.

Максимальна кількість порівнянь для цього алгоритму рівна *log2(N)*. Таким чином, приведений алгоритм суттєво виграє у порівнянні з лінійним пошуком.

Відомо декілька модифікацій алгоритму бінарного пошуку, які виконуються на деревах.

3.2. Метод інтерполяції



Якщо немає ніякої додаткової інформації про значення ключів, крім факту їхнього впорядкування, то можна припустити, що значення *key* збільшуються від *a[0]* до *a[N-1]* більш-менш „рівномірно". Це означає, що значення середнього елементу *a[N/2]* буде близьким до середнього арифметичного між найбільшим та найменшим значенням. Але, якщо шукане значення *key* відрізняється від вказаного, то є деякий сенс для перевірки брати не середній елемент, а „середньо-пропорційний".

Вираз для поточного значення *i* одержано з пропорційності відрізків:

$$\frac{a[e]-key}{key-a[b]}=\frac{e-i}{i-b}$$

В середньому цей алгоритм має працювати швидше за бінарний пошук, але у найгіршому випадку буде працювати набагато довше.

3.3. Метод „золотого перерізу"

Деякий ефект дає використання так званого „золотого перерізу". Це число φ, що має властивість:

$$\varphi-1=\frac{1}{\varphi};\quad \varphi^2-\varphi-1=0;\quad \varphi_{1,2}=\frac{1\pm\sqrt{5}}{2}.$$

Доданий корінь $\varphi=\frac{1+\sqrt{5}}{2}=1,61803398875\ldots$ і є золотим перерізом.

Згідно цього алгоритму відрізок слід ділити не навпіл, як у бінарному алгоритмі, а на відрізки, пропорційні φ та 1, в залежності від того, до якого краю ближче *key*.

3.4. Алгоритми пошуку послідовностей

Даний клас задача відноситься до задачі пошуку слів у тексті. Одним з найпростіших методів пошуку є послідовне порівняння першого символу з символами масиву. Якщо наявний збіг, тоді порівнюються другі, треті,... символи аж до повного збігу рядка *s* з частиною вектору такої ж довжини, або до незбігу у деякому символі. Тоді пошук продовжується з наступного символу масиву та першого символу рядку.

Існує варіант удосконалення цього алгоритму – це починати пошук після часткового збігу не з наступного елементу масиву, а з символу, наступного після тих, що переглядалися, якщо у рядку немає фрагментів, що повторюються.

Д. Кнут, Д. Моріс і В. Пратт винайшли алгоритм, який фактично потребує лише N порівнянь навіть в самому поганому випадку. Новий алгоритм базується на тому, що після часткового збігу початкової частини слова з відповідними символами тексту фактично відома пройдена частина тексту і можна „обчислити" деякі відомості (на основі самого слова), за допомогою яких потім можна швидко переміститися текстом.

Основною відмінністю КМП-алгоритму від алгоритму прямого пошуку є здійснення зсуву слова не на один символ на кожному кроці алгоритму, а на деяку змінну кількість символів. Таким чином, перед тим як виконувати черговий зсув, потрібно визначити величину зсуву. Для підвищення ефективності алгоритму необхідно, щоб зсув на кожному кроці був би якомога більшим.

Якщо j визначає позицію в слові, в якій міститься перший символ, який не збігається (як в алгоритмі прямого пошуку), то величина зсуву визначається як j-D. Значення D визначається як розмір самої довшої послідовності символів слова, які безпосередньо передують позиції j, яка повністю збігається з початком слова. D залежить тільки від слова і не залежить від тексту. Для кожного j буде своя величина D, яку позначимо d_j.

Так як величини d_j залежать лише від слова, то перед початком фактичного пошуку можна обчислити допоміжну таблицю d; ці обчислення зводяться до деякої попередньої трансляції слова. Відповідні зусилля будуть оправдані, якщо розмір тексту значно перевищує розмір слова ($M<<N$). Якщо потрібно шукати багатократні входження одного й того ж слова, то можна користуватися одними й тими ж d.

КМП-пошук дає справжній виграш тільки тоді, коли невдачі передувала деяка кількість збігів. Лише у цьому випадку слово зсовується більше ніж на одиницю. На жаль, це швидше виняток, ніж правило: збіги зустрічаються значно рідше, ніж незбіги. Тому виграш від практичного використання КМП-стратегії в більшості випадків пошуку в звичайних текстах досить незначний. Метод, який запропонували Р. Боуєр і Д. Мур в 1975 р., не тільки покращує обробку самого поганого випадку, але й дає виграш в проміжних ситуаціях.

БМ-пошук базується на незвичних міркуваннях – порівняння символів починається з кінця слова, а не з початку. Як і у випадку КМП-пошуку, слово перед фактичним пошуком трансформується в деяку таблицю. Нехай для кожного символу x із алфавіту величина dx – відстань від самого правого в слові входження x до правого кінця слова. Уявимо, що виявлена розбіжність між словом і текстом. У цьому випадку слово відразу ж можна зсунути праворуч на $dpM-1$ позицій, тобто на кількість позицій, швидше за все більше одиниці. Якщо символ, який не збігся, тексту в слові взагалі не зустрічається, то зсув стає навіть більшим, а саме зсовувати можна на довжину всього слова.

Варто сказати, що запропоновані методи пошуку послідовностей можна модифікувати таким чином, щоб у кожному рядку пошук йшов не до кінця кожного рядка, а на кількість шуканих символів менше, бо слово s не може бути розташоване у кінці одного рядка та на початку наступного.

Тема 2. Метод обчислення адреси, інтерполяційний пошук в масиві, бінарний пошук, пошук в таблиці, прямий пошук рядка.

3.5. Методи обчислення адреси

Нехай у кожному з M елементів масиву T міститься елемент списку (наприклад, ціле позитивне число). Якщо є деяка функція $H(V)$, що обчислює однозначно по елементі Vйого адресу - ціле позитивне число з інтервалу $[0,M-1]$,то V можна зберігати в масиві T з номером $H(V)$ тобто $V=T(H(V))$. При такому збереженні пошук будь-якого елемента відбувається за постійний час, не залежний від M.

Масив T **називається масивом хешування, а функція** H—функцією хешування (див. розділ 2).

При конкретному застосуванні хешування зазичай є визначена область можливих значень

елементів списку *V* і деяка інформація про них. На основі цього вибирається розмір масиву хешування *M* і будується функція хешування. Критерієм для вибору *M* і *H є* можливість їхнього ефективного використання.

Нехай треба зберігати лінійний список з елементів $K_1, K_2, .., K_n$, таких, що при $K_i = K_j$ mod(K_i,26)= mod(K_j,26). Для збереження списку виберемо масив хешування T(26) із простором адрес 0-25 і функцію хешування *H(V)*= =mod(V,26). Масив *T* заповнюється елементами $T(H(K_i)) = K_i$ і *T(j)*=0, якщо *j є* у ($H(K_1)$, $H\{K_2),..,H(K_n)$).

Пошук елемента *V* у масиві *T* із присвоюванням Z його індексу, якщо *V* міститься в *T,* чи -1, якщо V не міститься в *T,* здійснюється так:

```
int t[26],v,z,i;
        i=(int)fmod((double)v,26.0);
        if(t[i]==v) z=i;
         else z=-1;
```

Додавання нового елемента *V* у список з поверненням у Z індексу елемента, де він буде зберігатися, реалізується фрагментом

```
z=(int)fmod((doule)v,26.0);
t[zj=v;
```

а виключення елемента *V* зі списку присвоєнням

```
t[(int)fmod((double)v,26)3=0;
```

Тепер розглянемо складніший випадок, коли умова $K_i=K_j$ $H(K_i)=H(K_j)$ не виконується. Нехай *V*— множина можливих елементів списку (цілі позитивні числа), у якому максимальна кількість елементів дорівнює 6. Візьмемо *M*= 8 і як функцію хешування виберемо функцію H(V)=mod(V,8).

Припустимо, що B=8, причому $H(K_1)=5$, $H(K_2)=5$, $H(K_3)=5$, $H(K_4)=5$, $H(K_5)=5$,тобто $H(K_2)=H(K_4)$ хоча $K=K_4$. Така ситуація, як показано у попередніх розділах, називається колізією, і в цьому випадку при заповненні масиву хешування треба метод для її дозволу. Зазвичай вибирається перша вільна комірка за власною адресою. Для нашого випадку масив Г[8] може мати вигляд

$$T=<0,K_5,0,K_2,K_4, K_1, K_3,0>$$

При наявності колізій ускладнюються всі алгоритми роботи з масивом хешування. Розглянемо роботу з масивом T[100], тобто з простором адрес від 0 до 99. Нехай кількість елементів N не більш 99, тоді в **T** завжди буде хоча б один вільний елемент дорівнює нулю. Для оголошення масиву використовуємо оператор

```
int static t[ 100];
```

Додавання в масив T нового елемента Z із занесенням його адреси в **I** і числа елементів у **N** виконується так:

```
i=h(z);
        while (t[i]!=0 && t[i]!=z) if
    (i==99) i=0; else i++;
        if (t[i]!=z) t[i]=z, n++;
```

Пошук у масиві T елемента Z із присвоєнням **I** індекса Z, якщо Z є в **T,** чи =-1, якщо такого елемента немає, реалізується в такий спосіб:

```
i=h(z);
        while (t[i]!=0 && t[i]!=z) if
    (i==99) i=0; else i++; if (t[i]==0) i=-1;
```

При наявності колізій виключення елемента зі списку шляхом позначення його як порожнього, тобто t[i]=0, може привести до помилки. Наприклад, якщо зі списку ***B*** виключити елемент K_2, то одержимо масив хешування $T=<0,K_5,0,K_2,K_4,K_5,K_3,0>$, у якому неможливо знайти елемент K_4, оскільки $H(K_4)=3$, а T(3)=0. У таких випадках при виключенні елемента зі списку можна записувати в масив хешування деяке значення, що не належить області значень елементів списку і не рівне нулю. При роботі з таким масивом це значення буде вказувати на те, що треба переглядати із середини комірок.

Перевага методів обчислення адреси полягає в тому, що вони найшвидші, а недолік у том}' що

порядок елементів у масиві T не збігається з їх порядком у списку, крім того, досить складно здійснити динамічне розширення масиву **T.**

3.6. Інтерполяційний пошук елемента в масиві

Якщо відомо, що ключ лежить між K_1 і K_u то наступний пошук доцільно здійснювати не всередині впорядкованого масиву, а на відстані $(u-1)(K-K_1)/(K_u-K_1)$ від 1, припускаючи, що ключі є числами, що зростають приблизно в арифметичній прогресії.

Інтерполяційний пошук працює за log log ***N*** операцій, якщо дані розподілені рівномірно. Як правило, він використовується лише на дуже великих таблицях, причому робиться декілька кроків інтерполяційного пошуку, а потім на малому відрізку використовується бінарний або послідовний варіант.

```
Int interpolationSearch(int[] a, int toFind, int high)
{
// повертає індекс елемента зі значенням toFind або -1, якщо такого // елемента нема int low =
0;
      //high - кількість елементів масиву int mid;
while (a[low] < toFind && a[high] >= toFind)
{
          mid = low + ((toFind - a[low]) * (high - low)) / (afhigh] - a[low]); if (a[mid] <
      toFind) low = mid + 1 ; else if (a[mid] > toFind) high = mid - 1; else
              return mid;
}
if (a[low] == toFind) return low; else
          return -1 ; // Not found
}
```

3.7. Бінарний пошук із визначенням найближчих вузлів

У ряді випадків (зокрема, в завданнях інтерполяції) доводиться з’ясовувати, де по відношенню до заданого впорядкованого масиву дійсних чисел розташовується задане дійсне число. На відміну від пошуку в масиві цілих чисел, задане число в цьому випадку найчастіше не співпадає ні з одним із чисел масиву, і вимагається знайти номери елементів, між якими це число може бути розміщене.

Одним з найшвидших способів цього є бінарний пошук, характерна кількість операцій якого має порядок $\log_2(n)$, де n - кількість елементів масиву. При численних зверненнях до цієї процедури кількість операцій буде рівна m $\log_2(n)$ **(*m*** - кількість обходів). Прискорення цієї процедури можна добитися за рахунок збереження попереднього результату операції і спроб пошуку при новому обігу в найближчих вузлах масиву з подальшим розширенням області пошуку у разі неуспіху.

При цьому в найгірших випадках кількість операцій буде більшою (приблизно у 2 рази) в порівнянні з бінарним пошуком, але, зазвичай, при ***m,*** значно більшою, ніж $\log_2(n)$, вдається довести порядок кількості операцій до **т,** тобто зробити її майже незалежною від розміру масиву.

Завдання ставиться так. Заданий впорядкований масив дійсних чисел ***array*** розмірності ***n,*** значення ***value,*** що перевіряється, і початкове наближення вузла ***old.*** Вимагається знайти номер вузла ***res*** масиву ***array***, такий, що ***array*** **[*res*]** **<=*value*<*array*[*res* +1]**.

Алгоритм працює таким чином.

1. Визначається, чи лежить значення value за межами масиву array. У разі *value<array[0]* повертається -1, у разі *value>array[n-]]* повертається п-1.
2. Інакше перевіряється: якщо значення old лежить за межами індексів масиву (тобто *old*<0 або *old>0*, то переходимо до звичного бінарного пошуку, встановивши ліву межу *left*=0, праву *right*=*n*-1.
3. Інакше переходимо до з’ясування меж пошуку. Встановлюється *leß=right=old, inc*=\ - інкремент пошуку.

4. Перевіряється нерівність *vahie>=array[old]*. При його виконанні переходимо до наступного пункту (5), інакше до пункту (7).

5. Права межа пошуку відводиться далі: *right~right+inc.* Якщо *right*>=n-1, то встановлюється *right*= n-1 і переходимо до бінарного пошуку.

6. Перевіряється *value>=array[righf*. Якщо ця нерівність виконується, то ліва межа переміщається на місце правої: *left=right, inc* множиться на 2, і переходимо назад на (5). Інакше переходимо до бінарного пошуку.

7. Ліва межа відводиться: *left=left-inc.* Якщо *left*<=0, то встановлюємо *left*=0 і переходимо до бінарного пошуку.

8. Перевіряється *value<array[leß*]. При виконанні права межа переміщується на місце лівої: *right=left,* inc множиться на 2, переходимо до пункту (7). Інакше до бінарного пошук.

9. Проводиться бінарний пошук у масиві з обмеженням індексів *left* і *right.* При цьому кожного разу інтервал скорочується приблизно в 2 рази (у залежності від парності різниці), поки різниця між *left* і *right* не досягне 1. Після цього повертаємо *left* як результат, одночасно присвоюючи *old=left.*

```
int fbin_search(float value,int *old,float ‘array,int n)
{
register int left, right;
/* перевірка позиції за межами масиву 7 if(value < array[0]) retum(-1);
    if(value >= array[n-1]) return(n-l);
    процес розширення області пошуку. Перевіряємо валідність початкового
наближення 7 if(*o!d>=0 && *old<n-1)
        {
            register int inc=1; left = right = *old;
         if(value < arrayfold])
            {
                /* область розширюють вліво 7 while(1)
                {
                    left -= inc; if(left <= 0)
                    {
                        left=0;break;

                        }
                    if(array[left] <= value) break; right=left;
                 inc«=1;
                    }
                }
                else
                {
                    /* область розширюють вправо */ while(1)
                    {
                        right += inc; if(right >= n-1)
                        {
                            right=n-1 ;break;
                        }
                    }
                    if(array[right] > value) break; left=right; inc«=1;
                }
            /* початкове наближення погане- за область пошуку
          приймається весь масив */ else {
                left=0;right=n-1 ;
```

```
        }
        /* це алгоритм бінарного пошуку необхідного інтервалу */ while(left<right-1 )
        {
            register int node=(left+right)»1 ;
         if(value>=array[node]) left=node; eise right=node;
        }
    /* повертаємо знайдену ліву межу, обновивши старе
  значення результату */ retum(*old=left);
}
```

3.8. Пошук у таблиці

Пошук у масиві іноді називають пошуком у таблиці, особливо, якщо ключ сам є складовим об'єктом, таким, як масив чисел або символів. Часто зустрічається саме останній випадок, коли масиви символів називають рядками або словами. Рядковий тип визначається так:

String =array[0..Mv1] of char

відповідно визначається і відношення порядку для рядків **x** і y:

x = *y,* якщо xj = yj для *O* face=«Symbol» =< j < *M*

x < *y* якщо xi < **yi** для 0 face=«Symbol» =< i < *M i* xj = yj

для 0 face=«Symbol» =< j < **i**

Щоб встановити факт збігу, необхідно встановити, що всі символи порівнюваних пядків відповідно рівні один іншому. Тому порівняння складових операндів зводиться .іошуку частин, що неспівпадають, тобто до пошуку на нерівність. Якщо нерівних частин не існує, то можна говорити про рівність. Припустимо, що розмір слів достатньо малий, скажімо, менше ніж ЗО. В цьому випадку можна використовувати лінійний пошук і діяти так.

Для більшості практичних застосувань бажано виходити з того, що рядки мають змінний розмір. Це припускає, що розмір вказується в кожному окремому рядку. Якщо виходити з раніше описаного типу, то розмір не повинен перевершувати максимального розміру **M.** Така схема достатньо гнучка і придатна для багатьох випадків, в той самий час вона дозволяє уникнути складнощів динамічного розподілу пам'яті. Найчастіше використовуються два такі подання розміру рядків.

Розмір неявно вказується шляхом додавання кінцевого символа, більше цей символ ніде не вживається. Зазвичай, для цієї мети використовується недрукований символ із значенням 00h. (Для подальшого важливо, що це мінімальний символ зі всієї множини символів.)

Такий прийом має ту перевагу, що розмір явно доступний, а недолік - тому, що цей розмір обмежений розміром множини символів (256).

У подальшому алгоритмі пошуку віддається перевага першій схемі. У цьому випадку порівняння рядків виконується так: i:=0;

while (x[i]=y[ij) and (x[i]<>00h) do i:=i+1;

Кінцевий символ працює тут як бар'єр.

Тепер повернемося до завдання пошуку в таблиці. Він вимагає вкладених пошуків, а саме: пошуку по рядках таблиці, а для кожного рядка послідовних порівнянь між компонентами.

Тема 3. Алгоритми: Ахо-Корасика, Моріса-Прата, Кнута, рабіна-Карпа, Боуера-Мура, Хорспула. Порівняння методів пошуку.

3.9. Алгоритм Ахо-Корасик

Алгоритм Ахо-Корасик - алгоритм пошуку підрядків в рядку, створений Альфредом Ахо і Маргарет Корасик. Алгоритм реалізує пошук множини підрядків із словника в даному рядку. Час роботи пропорційний Θ ***(M+N+K) ,*** де N-довжина рядка - зразка, ***M —*** сумарна довжина рядків словника, а K - довжина відповіді, тобто сумарна довжина входжень слів із словника в рядок-зразок. Тому сумарний час роботи може бути квадратичним (наприклад, якщо в рядку 'ааааааа' ми шукаємо слова 'а', 'аа', 'ааа' ...).

q:=0;

```
for i := 1 to m do begin
        while g (q, T [i]) = -1 do q: = f(q);
q := g (q. т [i]);
if out (q) * 0 then write(i), out (q); end;
```

8.12. Алгоритм Моріса-Прата

Цей алгоритм модифікує прямий пошук, збільшуючи розмір зсуву, запам'ятовуючи одночасно частини тексту, що співпадають зі зраз ком.це дозволяє уникнути непотрібних порівнянь і збільшує швидкість пошуку.

Розглянемо порівняння на позиції ***i,*** де зразокх[0, ***m*** – 1] порівнюється зі частиною тексту ***y[i, i + m–*** 1]. Припустимо, що перша розбіжність відбулася між у[i + **j]** і х**[j]** де 1 <*j* < ***m.***

Введемо поняття префікс-функції.

- **Префікс-функцією** називається функція, що повертає найбільший префікс рядка. Префіксом рядка називається підрядок, який одночасно є і суфіксом (закінченням рядка). Так, для рядка «аавпа» префік-функція поверне символ «а», а для рядка «ааввсваа» - підрядок «аав».

При зсуві можна очікувати, що префікс зразка «співпаде з якимсь суфіксом підслова тексту *и.* Найдовший такий префікс - межа *и* (він зустрічається на обох кінцях и). Це приводить нас до наступного алгоритму: нехай *mp_next*[*j*] - довжина межі **x[0,j** - 1]. Тоді після зсуву ми можемо відновити порівняння з місця **y[i** + у] і *x*[*j-mp*_next[j]] без втрати можливого місцезнаходження зразка. Таблиця *mp next* може бути обчислена за (*m)* перед самим пошуком. Максимальна кількість порівнянь на один символ - *m.*

```
void PRE_MP( char *x, int m, int mp_next[])
{
        inti.j;
        i=0;
        j=mp_next[0]=-1; while (i < m )
        {
                while (j > -1 && x[i] != x[j]) j=mp_next[j];
            mp_next[++i]=++j;
        }
}
void MP( char *x, char *y, int n, int m)
{
        int i, j, mp_next[XSIZE];
        PRE_MP(x, m, mp_next); i=j=0;
        while (i < n )
        {
            while (j > -1 && x[j] != y[i]) j=mp_next[j]; i++; j++;
            if (j >= m)
            {
                ouTPUT(i-j);
                j = mp_next[j];
            }
        }
}
```

8.13. Алгоритм Кнута, Моріса і Пратта

Приблизно в 1970 р. Д. Кнут, Д. Моріс і В. Пратт винайшли алгоритм (КМП), що фактично вимагає тільки У порівнянь навіть в найгіршому випадку. Новий алгоритм грунтується на тому міркуванні, що після часткового співпадіння початкової частини слова з відповідними символами тексту фактично відома пройдена частина т" **-у** і можна визначити деякі відомості (на основі самого слова), за допомогою яких потім можна швидко просунутися по тексту. Приведений приклад пошуку слова ABCABD показує принцип роботи такого алгоритму.

Символи, що порівнювалися, тут підкреслені. Зверніть увагу: при кожному неспівпадінні пари символів слово зсувається на усю пройдену відстань, оскільки менші зсуви не можуть привести до повного співпадіння.

ABCABCABAABCABD
ABCABD
ABCABD
ABCABD
ABCABD
ABCABD

Основною відмінністю КМП-алгоритму від алгоритму прямого пошуку є здійснення зсуву слова не на один символ на кожному кроці алгоритму, а на деяку змінну кількість символів. Отже, перш ніж здійснювати черговий зсуву, необхідно визначити величину зсуву. Для підвищення ефективності алгоритму необхідно, щоб зсув на кожному кроці був якомога більшим.

Якщо **j** визначає позицію в слові, що містить перший символ, що не співпала (як в алгоритмі прямого пошуку), то величина зсуву визначається як ***j-D.*** Значення ***D*** визначається як розмір найдовшої послідовності символів слова, які безпосередньо передують позиції у, і яка повністю співпадає з початком слова. ***D*** залежить тільки від слова і не залежить від тексту. Для кожного j буде своя величина ***D,*** яку позначимо ***d.***

Оскільки величини d_j залежать тільки від слова, то перед початком фактичного пошуку можна обчислити допоміжну таблицю ***d.*** Відповідні зусилля будуть виправданими, якщо розмір тексту значно перевищує розмір слова **(*M*N*).** Якщо треба шукати багато входжень одного і того ж слова, то можна користуватися одними і тими самим **d.** Точний аналіз КМП-пошуку як і сам його алгоритм, вельми складний. Його винах -.ьаодять, що треба порядку **M+N** порівнянь символів, що значно краще,ніж **M*N** порівнянь із прямого пошуку. Вони так само відзначають таку позитивну властивість, як показник сканування, який ніколи не повертається назад, тоді як при прямому пошуку після неспівпадіння перегляд завжди починається з першого символа слова і тому може включати символи, які раніше вже були видимими. Це може привести до негативних наслідків, якщо текст читається з вторинної пам'яті, адже в цьому випадку повернення обходиться дорого. Навіть при введенні буфера може зустрітися таке велике слово, що повернення перевищить місткість буфера.

```
Varn: longint;
T: array[1 ..40000] of char;
S : array[1..10000] of char;
P: array[1 ..10000] of word; {масив, в якому зберігається значення префікс-
функції}
    i,k: longint; m : longint;
Procedure Prefix; {процедура, яка визначає префікс-функцію}
Begin
        P[1]:=0; {префікс рівний нулеві} к:=0;
          for i:=2 to m do begin
        while (k>0) and (S[k+1]<>S[i]) do
              k:=P[k]; {отримаємо значення функції з попередніх розрахунків} if S[k+1]=S[i]
          then k:=k+1;
          P[i]:=k; {присвоєння префікс-функції} end;
End;
```

8.14. Алгоритм Рабіна-Карпа

Ідея, запропонована **Рабіном** і **Карпом,** має на увазі поставити у відповідність кожному рядку деяке унікальне число і замість того, щоб порівнювати самі рядки, порівнювати числа, що набагато швидше. Проблема в тому, що шуканий рядок може бути довгим, рядків у тексті теж вистачає. А оскільки кожному рядку треба поставити у відповідність число, то і чисел має бути багато, а отже, числа будуть великими (порядку $\mathbf{D_m}$**, це D -** кількість різних символів), і працювати з ними буде так само незручно.

```
VarT : array[1 ..40000] of 0..9;
        S : array[1..8]of 0..9; i j : longint; n,m: longint;
     v,w: longint; {v - число, яке характеризує рядок, що шукається, w характеризує
  рядок довжини m у тексті} k ; longint;
const D : longint = 10; {кількість різних символів}
Begin
            v:=0;
            w:=0;
            for i:=1 to m do begin
        v:=v*D+S[i]; {визначення v, рядок поданий як число} w:=w*D+T[i]; {
     визначення початкового значення w} end; k:=1;
            for i:=1 to m-1 do
{k необхідне для багатократного визначення w і рівне Dm-1}
         k:=k*D;
             for i:=m+1 to n+1 do begin
                 if w=v then {якщо числа рівні, то рядки співпадають} write In
              (‘УРА’); if i<=n then
                          w:=d*(w-k*T[i-m])+T[i]; { визначення нового значення w}
         end;
End.
```

Цей алгоритм виконує лінійне проходження по рядку **(т** кроків) і лінійне проходження по всьому тексту **(п** кроків), отже, спільний час роботи є 0 (**п+т**). Цей час лінійно залежить від розміру рядка і тексту, отже, програма працює швидко. Але який інтерес працювати тільки з короткими рядками і цифрами? Розробники алгоритму придумали, як поліпшити цей алгоритм без особливих втрат у швидкості роботи. Як ви помітили, ми ставили у відповідність рядку його числове подання, але виникала проблема великих чисел. Її можна уникнути, якщо проводити всі арифметичні дії з модулями якогось простого числа (постійно брати залишок від ділення на це число). Таким чином знаходиться не само число, що характеризує рядок, а його залишок від ділення на якесь просте число. Тепер ми ставимо число у відповідність не одному рядку, а цілому класу, але оскільки класів буде досить багато (стільки, скільки різних залишків від ділення на це просте число), то додаткову перевірку доведеться виконувати рідко.

```
V:=0;
w:=0;
         for i:=1 to m do { визначення v та w} begin
         v:=(v*D+ord(S[i])) mod P; {ord перетворює символ у число} w:=(w*D+ord(T[i])) mod P;
       end;
k:=1;
for i:=1 to m-1 do
             k:=k*D mod P; {k маємо значення Dm-1 mod P}
         for i:=m+1 to n+1 do begin
         if w=v then {якщо числа рівні, то рядки належать до одного класу, перевірка на
       співпадіння} begin J>0;
                          while (j<m) and (S[j+1]=T[i-m+j]) do j:=j+1;
                          if j=m then {кінцева перевірка}
                          writeln(‘yPA’);
                 end;
         if i<=n then
                 w:=(d*(w+P-(ord(T[i-m])*k mod P))+ord(T[i])) mod P;
             end.
```

Змістовий модуль 4. Алгоритми сортування. Загальна класифікація та принципи роботи. Жадібні алгоритми.

Тема 1. Алгоритми сортування основні поняття. Методи внутрішнього сортування. Метод простого включення, сортування шляхом підрахунку, метод Шелла, обмінне сортування, сортування вибором.

9.1. Методи внутрішнього сортування

У загальній постановці завдання сортування ставиться таким чином. Є послідовність однотипних записів, одне з полів яких вибране як ключове (далі ми називатимемо його ключем сортування). Тип даних ключа повинен включати операції порівняння («=», «>», «<», «>=» і «<=»). Завданням сортування є перетворення початкової послідовності в послідовність, що містить ті самі записи, але у порядку зростання (або спадання) значень ключа. Метод сортування називається стійким, якщо при його застосуванні не змінюється відносне положення записів із рівними значеннями ключа.

Розрізняють сортування масивів записів, розташованих в основній пам'яті (внутрішнє сортування), і сортування файлів, що зберігаються в зовнішній пам'яті і не вміщуються повністю в основній пам'яті (зовнішнє сортування). Для внутрішнього і зовнішнього сортування потрібні істотно різні методи.

Природною умовою, що висувається до будь-якого методу внутрішнього сортування є те, що ці методи не повинні вимагати додаткової пам'яті: всі перестановки

з метою впорядкування елементів масиву мають вироблятися в межах того ж масиву. Мірою ефективності алгоритму внутрішнього сортування є кількість необхідних порівнянь значень ключа (С) і кількість перестановок елементів **(М).**

Зазначимо, що оскільки сортування засноване тільки на значеннях ключа і ніяк не зачіпає поля записів, що залишилися, можна говорити про сортування масивів ключів.

9.1.1. Метод простого включення

Нехай є масив ключів **a[** 1], **a[**2],..., a[n]. Для кожного елемента масиву, починаючи з другого, здійснюється порівняння з елементами з меншим індексом (елемент **a[i]** послідовно порівнюється з елементами **a[i-**1], a[i-2]...) і до тих пір, поки для чергового елемента **a[j]** виконується співвідношення **a[j]> a[i], a[i]** і **a[j]** міняються місцями. Якщо вдається зустріти такий елемент **a[j],** що **a[j]<=** *a[i],* або якщо досягнута нижня межа масиву, здійснюється перехід до опрацювання елемента a[/+1] (поки не буде досягнута верхня межа масиву) (рис.).

Можна побачити, що в кращому разі (коли масив вже впорядкований) для виконання алгоритму з масивом із п елементів буде треба **n-**1 порівняння і 0 пересилань. У гіршому разі (коли масив впорядкований у зворотному порядку) буде треба **n*(n-**1)/2 порівнянь і стільки ж пересилань.

Отже, можна оцінювати складність методу простих включень як $\Theta (n^2)$.

Можна скоротити кількість порівнянь, що використовуються в методі простих включень, якщо скористатися тим фактом, що при опрацюванні елемента масиву a[i] елементи a[1], **a[**2], ..., **a[i-**1] вже впорядковані, і скористатися для пошуку елементу, з яким має виконуватись перестановка, методом двійкового розподілу. У цьому випадку оцінка кількості необхідних порівнянь стає Θ (nlog n).

Зазначимо, що оскільки при виконанні перестановки потрібне зсування на один елемент декількох елементів, то оцінка кількості пересилань

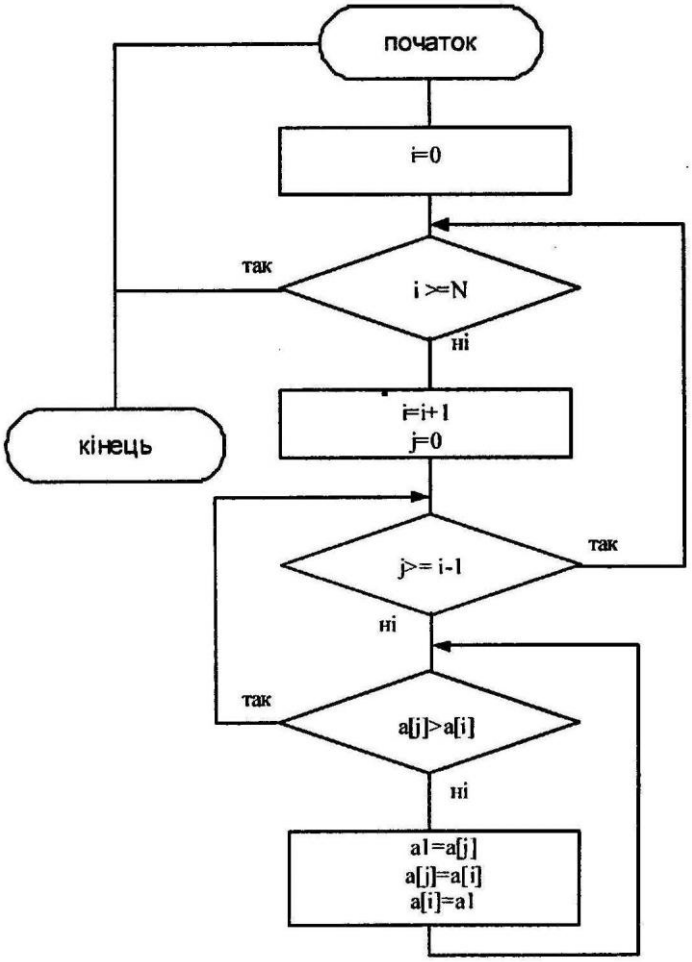


Рис. 4.1. Блок-схема сортування методом включення

```
Void sort_vstavka()
{
    int;  int a[50], a1;
    int k[50];
    i=0;
    while(i<50)
    {
        for  (j=i-1; j>=0; i—)
        {
            if(a[j]>a[i])
            {
                a1=a[j];
                a[j]=a[i];
                    a[i]=a1; i—;
            }
        }
        i++;
    }
}
```

Таблиця 4.1 Приклад сортування методом простого включення

Початковий стан масиву	8 23 5 65 44 33 1 6
Крок 1	8 23 5 65 44 33 1 6
Крок 2	8 5 23 65 44 33 1 6
	5 8 23 65 44 33 1 6
Крок 3	5 8 23 65 44 33 1 6
Крок 4	5 8 23 44 65 33 1 6
Крок 5	5 8 23 44 33 65 1 6
	5 8 23 33 44 65 1 6
Крок 6	5 8 23 33 44 1 65 6
	5 8 23 33 1 44 65 6
	5 8 23 1 33 44 65 6
	5 8 1 23 33 44 65 6
	5 18 23 33 44 65 6
	1 5 8 23 33 44 65 6
Крок 7	1 5 8 23 33 44 6 65
	1 5 8 23 33 6 44 65
	1 5 8 23 6 33 44 65
	15 8 6 23 33 44 65
	15 6 8 23 33 44 65

9.1.3. Сортування шляхом підрахунку

Кожний елемент порівнюється зі всіма іншими; остаточно положення елементів визначаються після підрахунку кількості менших ключів.

Треба знайти перестановку p(1)***p(2)...p(n),*** таку, що

Кр(*I*)<=Кр(2)<=... <Кр(п).

Метод заснований на тому щоу'-кпюч впорядкованої послідовності перевищує рівно j-1 інших ключів. Ідея полягає в тому, щоб зрівняти попарно всі ключі і підрахувати, скільки з них менші від кожного окремого ключа.

Спосіб виконання поставленого завдання такий:

((порівняти ***Kj*** з ***Ki)*** при 1 <=j<i) при ***1<i<=N.***

```
Void sort_pidr()
{
        inti.j; int a[50], a1[50]; int k[50];
            for (i=0; i<50; i++) k[i]=O;
        for (i=0; i<50-1; I++) for(j=i+1; j<50;
    j++) if(a[i]<aO])
                    ВД++;
        else k[j]++; for (i=0; i<50; i++)
    a1[k[i]]=a[i]; for (i=0; i<50; i++)
    a[i]=a1[i];
```

9.1.2. Метод Шелла

Подальшим розвитком методу сортування з включеннями є сортування методом Шелла, яке інакше називається сортуванням включеннями з відстанню, що зменшується.

Метод полягає в тому, що таблиця, яка впорядковується, розділяється на групи елементів, кожна з яких упорядковується методом простих включень. У процесі впорядкування розміри таких груп збільшуються доти, поки всі елементи таблиці не ввійдуть у впорядковану групу. Вибір чергової групи для сортування і її розташування всередині таблиці здійснюється так, щоб можна було використовувати попередню впорядкованість. Групою таблиці називають послідовність елементів, номери яких утворять арифметичну прогресію з різницею А (А називають кроком групи). На початку процесу впорядкування вибирається перший крок групи $A_{|3}$ що залежить від розміру таблиці. Шелл запропонував брати

h_j=[n/2], а h_i=h((i-1)/2).

У більш пізніх роботах Хіббард показав, що для прискорення процесу доцільно визначити крок h_1, за формулою:

h_1=2**k+1 , де $2^{**}k < n <= 2^{**}(k+1)$.

Після вибору h_1 методом простих включень впорядковуються групи, що містять елементи з номерами позицій $i, i+h_1, i+2*h_1, \ldots, i+m_i, *h_1$.

При цьому i= 1,2,...h_1; m[i] - найбільше ціле, що задовольняє нерівність $m[i]*h <= n$.

Потім вибирається крок $h_2<h_1$, і впорядковуються групи, що містять елементи з номерами позицій. Ця процедура з усе зменшуваними кроками продовжується доти, поки черговий крок h[l] стане рівним одиниці ($h_1>h_2>.. .>h_n$,). Цей останній етап являє собою впорядкування всієї таблиці методом включень. Але оскільки вихідна таблиця впорядковувалася окремими групами з послідовним об'єднанням цих груп, то загальна кількість порівнянь значно менша, ніж *n* /4, необхідна при методі включень. Кількість операцій порівняння пропорційна n*($\log_2$(n))**2 .

Застосування методу Шелла до масиву, використаного в наших прикладах, показано в таблиці

Таблиця 4.2 Приклад сортування методом Шелла

Початковий стан масиву	8 23 5 65 44 33 1 6
Фаза 1 (сортуються елементи, відстань між якими чотирьом)	8 23 5 65 44 33 1 6
	8 23 5 65 44 33 1 6
	8 23 1 65 44 33 5 6
	8 23 1 6 44 33 5 65
Фаза 2 (сортуються елементи, відстань між якими двом)	1 23 8 6 44 33 5 65
	1 23 8 6 44 33 5 65
	1 23 8 6 5 33 44 65
	1 23 5 6 8 33 44 65
	1 6 5 23 8 33 44 65
	1 6 5 23 8 33 44 65
	1 6 5 23 8 33 44 65
Фаза 3 (сортуються елементи, відстань між якими одному)	1 6 5 23 8 33 44 65
	1 5 6 23 8 33 44 65
	1 5 6 23 8 33 44 65
	1 5 6 8 23 33 44 65
	1 5 6 8 23 33 44 65
	1 5 6 8 23 33 44 65
	1 5 6 8 23 33 44 65

У загальному випадку алгоритм Шелла природно переформулюється для заданої послідовності з ї відстаней між елементами A,, A,, ..., A(, для яких виконуються умови A, = 1 і A(/+1)< A. При правильно підібраних складність алгоритму Шелла є 0 («(1.2)), що істотно менша за складність простих алгоритмів сортування. Блок-схема методу Шелла подана на рис. У ньому використано змінні: a[] - массив, який необхідно відсортувати, t - допоміжна змінна того ж типу, що і масив, n - розмірність масиву.

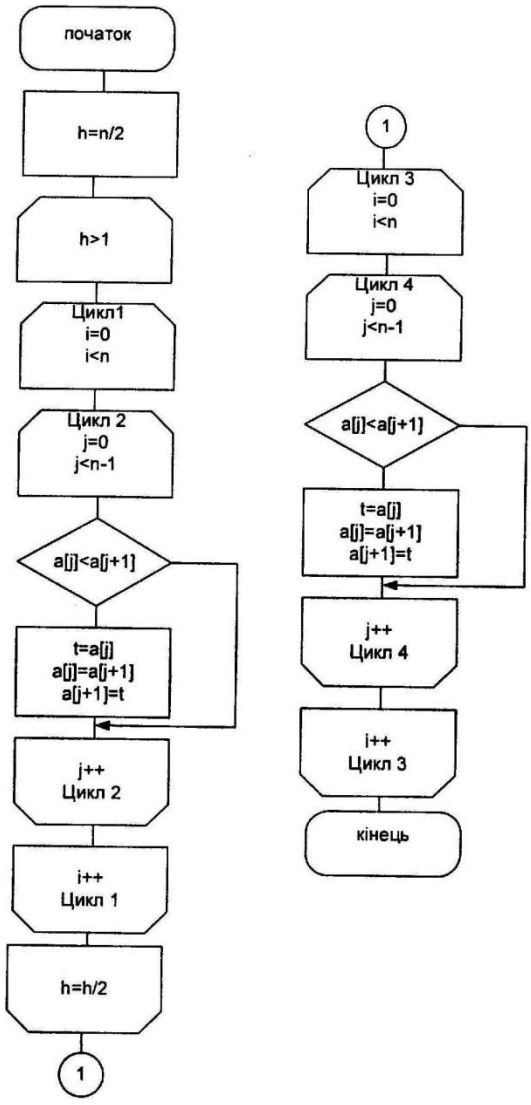


Рис. 4.1 Блок-схема методу Шелла.

Тема 2. Сортування поділом (Хоара), за допомогою дерева, пірамідальне сортування, сортування злиттям, методи порозрядного сортування.

1.1. Сортування поділом (Хоара)

Метод сортування поділом був запропонований Чарльзом Хоаром 1962 року. Цей метод є розвитком методу простого обміну і настільки ефективний, що його почали називати «методом швидкого сортування - Quicksort».

Основна ідея алгоритму полягає в тому, що випадковим чином вибирається деякий елемент масиву x, після чого масив є видимим зліва, поки не зустрінеться елемент *a[i]* такий, що ***a[i]>x,*** а потім масив є видимим справа, поки не зустрінеться елемент *a[j]* такий, що *a[j]*<x. Ці два елементи міняються місцями, і процес перегляду, порівняння і обміну продовжується, поки ми не дійдемо до елемента **x.** У результаті масив виявиться розбитим на дві частини - ліву, в якій значення ключів будуть менші від **x,** і праву із значеннями ключів, більшими від x. Далі процес рекурсивно продовжується для лівої і правої частин масиву до тих пір, поки кожна частина не міститиме лише один елемент. Зрозуміло, що, як завжди, рекурсію можна замінити ітераціями, якщо запам'ятовувати відповідні індекси масиву. Прослідкуємо цей процес на прикладі нашого стандартного масиву

Таблиця 4.3. Приклад швидкого сортування

Початковий стан масиву	8 23 5 65 |44| 33 1

Крок 1 (як x вибирається a[5])	8 23 5 6 4433 1 65
	8 23 5 6 1 33 44
Крок 2 (у підмасиві a[1], a[5] як x вибирається a[3])	8 23 |5| 6 1 33 44
	1 23 5 6 8 33 44
	1 5 23 6 8 33 44
Крок 3 (у підмасиві a[3], a[5] як x вибирається a[4])	1 5 23 |6| 8 33 44
	1 5 8 6 23 33 44
Крок 4 (у підмасивах a[3], a[4] вибирається a[4])	1 5 8 |6| 23 33 44
	1 5 6 8 23 33 44

Алгоритм недаремно називається швидким сортуванням, оскільки для нього оцінкою кількості порівнянь і обмінів є Θ (*n* log ***n*).** Насправді, в більшості утиліт, що виконують сортування масивів, використовується саме цей алгоритм.

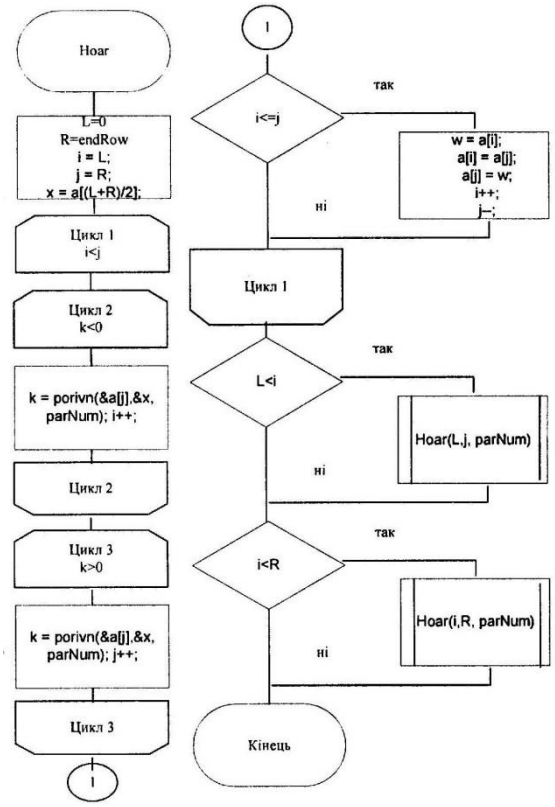


Рис. 4.4 Блок-схема рекурсивного методу Хоара

9.1.7. Сортування за допомогою дерева

Почнемо з простого методу сортування за допомогою дерева, при використанні якого будується двійкове дерево порівняння ключів, Побудова дерева починається з листя, яке містить всі елементи масиву. З кожної сусідньої пари вибирається найменший елемент, і ці елементи утворюють наступний (ближчий до кореня рівень дерева). Із кожної сусідньої пари вибирається найменший елемент і т.ін., поки не буде побудований корінь, тобто найменший елемент масиву.

Приклад двійкового дерева показано на рис. 9.5.

Отже, ми вже маємо якнайменше значення елементів масиву. Щоб одержати наступний по величині елемент, спустимося від коріння по шляху; що веде до листка з якнайменшим значенням.

У цій листковій вершині ставиться фіктивний ключ з «нескінченно великим» значенням, а у всі проміжні вузли, що займалися якнайменшим елементом, вноситься якнайменше значення з вузлів - безпосередніх нащадків (рис. 9.6). Процес продовжується доти, поки всі вузли дерева не будуть заповнені фіктивними ключами.

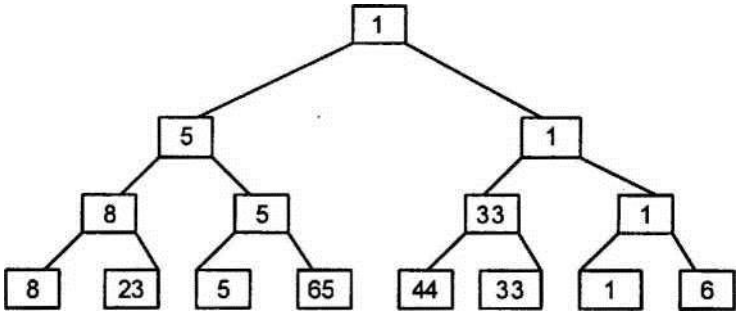


Рис. 4.5. Перший крок

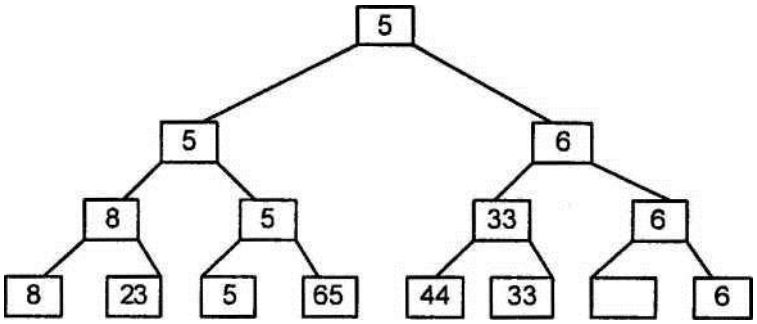


Рис. 4.6. Другий крок

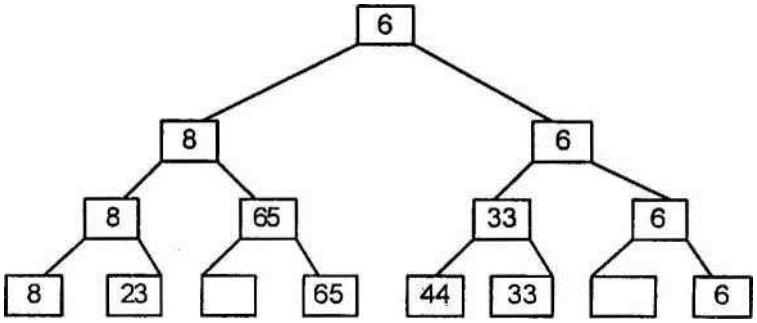


Рис. 4.7. Третій крок

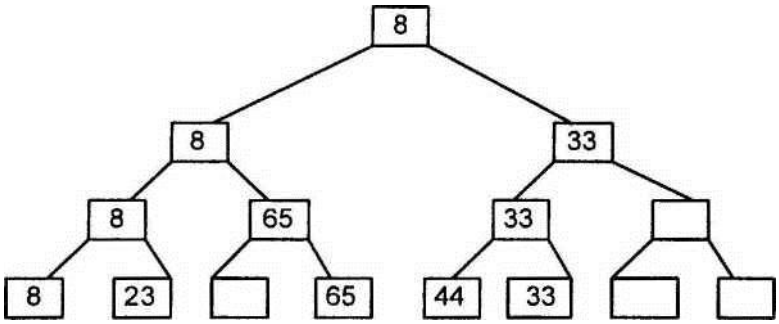


Рис. 4.8. Четвертий крок

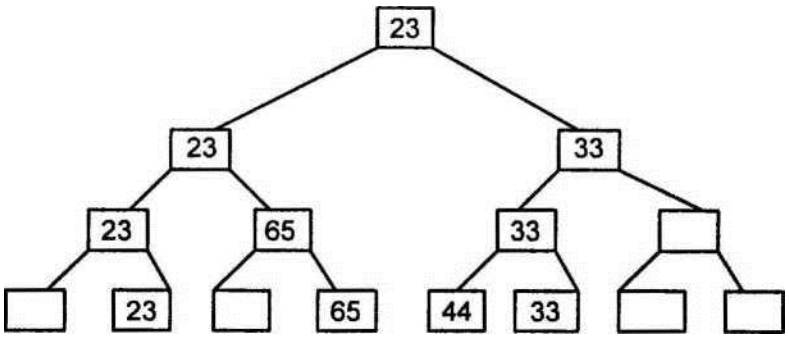


Рис. 4.9. П'ятий крок

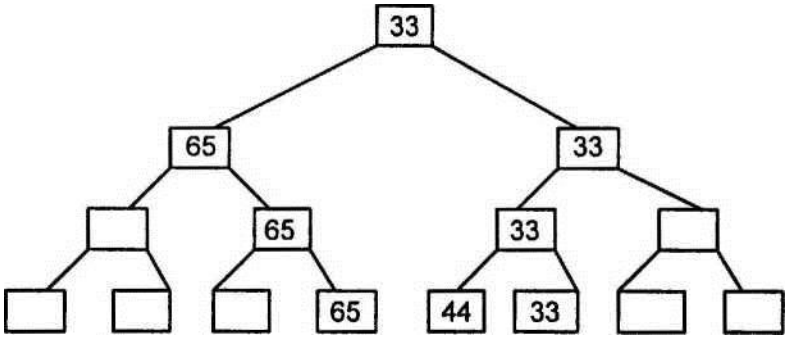


Рис. 4.10. Шостий крок

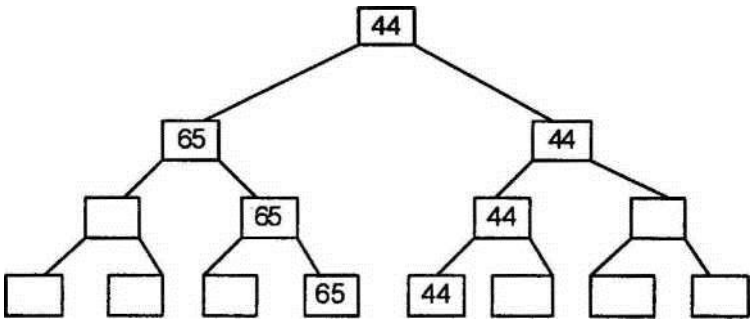


Рис. 4.11. Сьомий крок

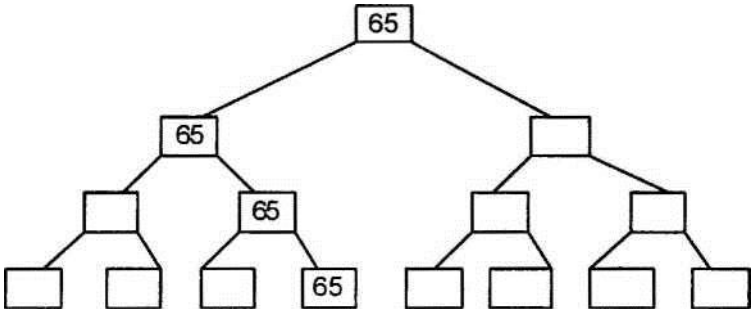


Рис. 4.12. Восьмий крок

9.1.8. Пірамідальне сортування

Є досконаліший алгоритм, який прийнято називати пірамідальним сортуванням (НеарБоїТ). Його ідея полягає у тому, що замість повного дерева порівняння початковий масив а[1], *a[*2i], ..., *a[n]* перетвориться на піраміду, що має таку властивість, що для кожного *a[i*] виконуються умови *a[i*]<= *a[2i]* і а[i]<= а[2i+1]. Потім піраміда використовується для сортування.

Найнаочніше метод побудови піраміди виглядає при деревовидному зображенні масиву, показаному на рис. 9. ІЗ. Масив подається у вигляді двійкового дерева, корінь якого відповідає елементу масиву л[1]. На другому ярусі знаходяться елементи *a[2]* і а[3]. На третьому - [4], *a[*5], *a[*6], а[7] і т.ін. Як видно, для масиву з непарною кількістю елементів відповідне дерево буде збалансованим, а для масиву з парною кількістю елементів *n* елемент *a[n]* буде єдиним (найлівішим) листком «майже» збалансованого дерева.

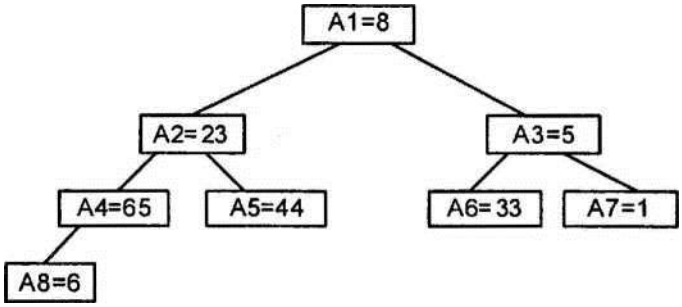


Рис. 4.13. Приклад пірамідального сортування

Очевидно, що при побудові піраміди нас цікавитимуть елементи *a[n/2]*, *a[n/*2-1], ..., a[1] для масивів з парною кількістю елементів і елементи a[(n-1)/2], a[(n-1)/2-1], ..., a[1] для масивів з непарною кількістю елементів (оскільки тільки для таких елементів істотні обмеження піраміди). Нехай *i* - найбільший індекс з індексів елементів, для яких існують обмеження піраміди. Тоді береться елемент **a[i]** у побудованому дереві і для нього виконується процедура просівання, що полягає у тому, що вибирається гілка дерева, яка відповідає min(a[2i], a[2i+1]), і значення **a[i]** міняється місцями із значенням відповідного елементу'. Якщо цей елемент не є листком дерева, для нього виконується аналогічна процедура і т.ін. Такі дії виконуються послідовно для a[j], **a[i-**1], ..., **a[**1]. Як бачимо, в результаті ми одержимо деревовидне подання піраміди для початкового масиву (послідовність кроків для використовуваного в наших прикладах масиву показана на рис.

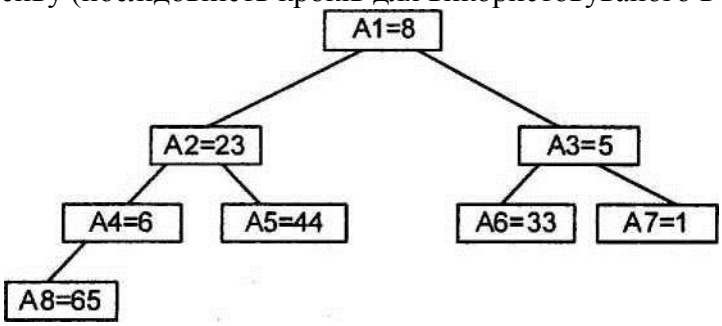


Рис. 4.14. Пірамідальне сортування. Крок 1.

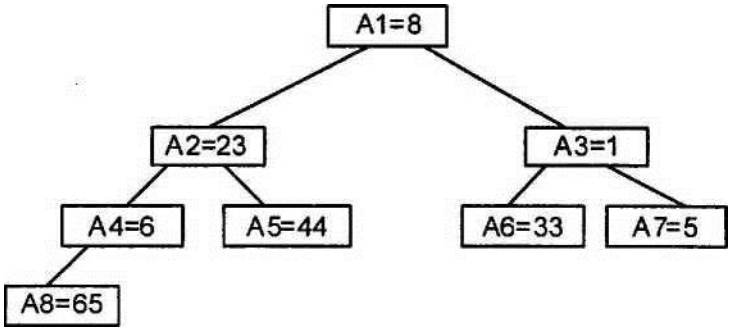


Рис. 4.15. Пірамідальне сортування. Крок 2.

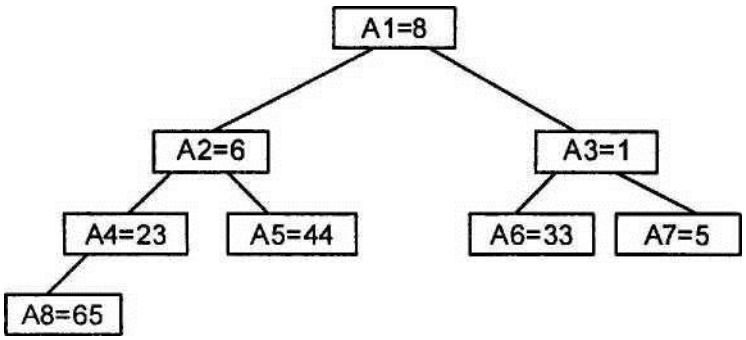


Рис. 4.16. Пірамідальне сортування. Крок 3.

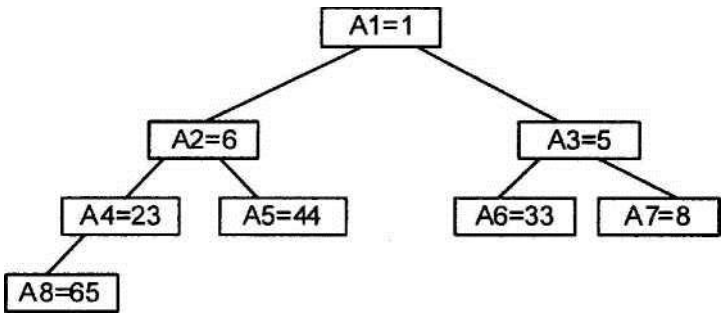


Рис. 4.17. Пірамідальне сортування. Крок 4.

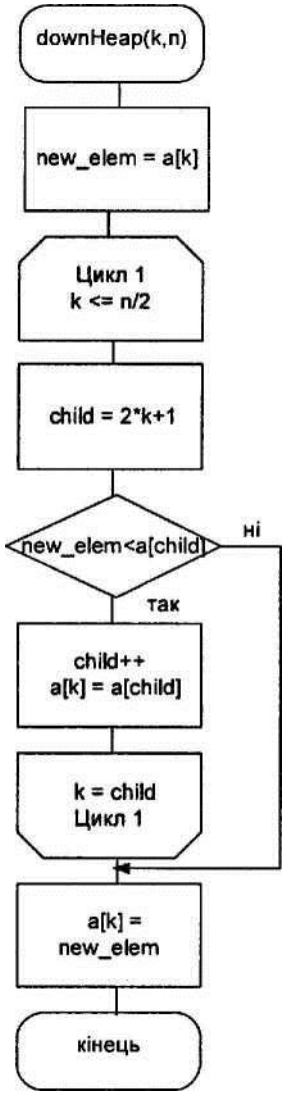


Рис. 4.18. Блок-схема методу впорядкування піраміди

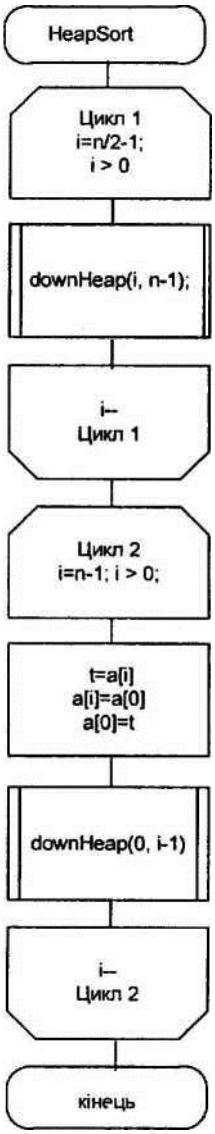


Рис. 4.19. Блок-схема побудови піраміди та її перевірки

9.1.9. Побудова піраміди методом Флойда

1964 року Флойд запропонував метод побудови піраміди без явної побудови дерева (хоча метод заснований на тих самих ідеях). Побудова піраміди методом Флойда для нашого стандартного масиву показана в таблиці 9.7.

Таблиця 4.4 Приклад побудови піраміди.

Початковий стан масиву	8 23 5 |65| 44 33 1 6
Крок 1	8 23 |5| 6 44 33 1 65
Крок 2	8 |23| 1 6 44 33 5 65
Крок 3	|8| 6 1 23 44 33 5 65
Крок 4	1 6 8 23 44 33 5 65
	1 6 5 23 44 33 8 65

У таблиці 9.8 показано, як здійснюється сортування з використанням побудованої піраміди. Суть алгоритму полягає в подальшому. Нехай /'-найбільший індекс масиву, для якого вказані умови піраміди. Тоді, починаючи з я[1] до *a[i]* виконуються такі дії.

Таблиця 4.5. Сортування за допомогою піраміди.

Початкова піраміда	1 6 5 23 44 33 8 65
Крок 1	65 6 5 23 44 33 8 1
	5 6 65 23 44 33 8 1
	5 6 8 23 44 33 65 1
Крок 2	65 6 8 23 44 33 5 1 !
	6 65 8 23 44 33 5 1
	6 23 8 65 44 33 5 1
КрокЗ	33 23 8 65 44 6 5 1
	8 23 33 65 44 6 5 1
Крок 4	44 23 33 65 8 6 5 1
	23 44 33 65 8 6 5 1
Крок 5	65 44 33 23 8 6 5 1
	33 44 65 23 8 6 5 1
Крок 6	65 44 33 23 8 6 5 1
	44 65 33 23 8 6 5 1
Крок 7	65 44 33 23 8 6 5 1

На кожному кроці вибирається останній елемент піраміди (у нашому випадку першим буде вибраний елемент а[8]. Його значення міняється зі значенням а[1], після чого для *a[* 1] виконується просіювання. При цьому на кожному кроці кількість елементів в піраміді зменшується на 1 (після першого кроку як елементи піраміди розглядаються а[1], *a[*2], ..., *a[n*-1]; після другого - а[1], *a[*2], ..., *a[n*-2] і т.ін., поки в піраміді не залишиться один елемент). Легко побачити (це ілюструється в таблиці), що як результат ми одержимо масив, впорядкований у порядку спадання. Можна модифікувати метод побудови піраміди і сортування, щоб одержати впорядкування у порядку зростання, якщо змінити умову піраміди *a[i]>a[2i]* і а[1]>=а[2i+1] для всіх значень індекса.

Тема 3. Методи зовнішнього сортування. Пряме злиття, природне злиття, збалансоване багатошляхове злиття, багатофазне сортування.

1.2. Методи зовнішнього сортування

Прийнято називати «зовнішнім» сортування послідовних файлів, розташованих у зовнішній пам’яті і дуже великих, щоб можна було повністю перемістити їх в основну пам’ять і застосувати один з розглянутих у попередньому розділі методів внутрішнього сортування. Найчастіше зовнішнє сортування застосовується в системах керування базами даних при виконанні запитів, і від ефективності вживаних методів істотно залежить продуктивність СУБД.

Мусимо пояснити, чому йдеться саме про послідовні файли, тобто про файли, які можна читати запис за записом в послідовному режимі, а писати можна тільки після останнього запису. Методи зовнішнього сортування з’явилися, коли найбільш поширеними пристроями зовнішньої пам’яті були магнітні стрічки. Для стрічок послідовний доступ був абсолютно природним. Коли відбувся перехід до пристроїв, що запам’ятовують, з магнітними дисками, що забезпечують «прямий» доступ до буць- якого блоку інформації, здавалося, що послідовні файли втратили свою актуальність. Проте це припущення було помилковим.

Вся річ у тому, що практично всі використовувані на сьогодні дискові пристрої забезпечені рухомими магнітними головками. При виконанні обміну з дисковим накопичувачем виконується підведення головок до потрібного циліндра, вибір потрібної головки (доріжки), прокручування

дискового пакету до початку необхідного блоку і, нарешті, читання або запис блоку. Серед всіх цих дій найбільший час займає підведення головок. Саме цей час визначає загальний час виконання операції. Єдиним доступним прийомом оптимізації доступу до магнітних дисків є якомога «ближче» розташу вання файла на накопичувані блоків, що послідовно адресуються. Але і в цьому випадку рух головок буде мінімізованим тільки у тому випадку, коли файл читається або пишеться в послідовному режимі. Саме з такими файлами при потребі сортування працюють сучасні СУБД.

Зазначимо, що насправді швидкість виконання зовнішнього сортування залежить від розміру буфера (або буферів) основної пам'яті, яка може бути використана для цих цілей.

9.2.1. Пряме злиття

Припустимо, що є послідовний файл ***A,*** що складається із записів ***a***\, я2,..., ***an*** (знову для простоти припустимо, що ***n*** є ступенем числа 2). Вважатимемо, що кожен запис складається лише з одного елементу, що є ключем сортування. Для сортування використовуються два допоміжні файли В і С (розмір кожного з них буде ***n/***2).

Сортування складається з послідовності кроків, в кожному з яких виконується розподіл стану файлу ***A*** у файли ***B*** і ***C,*** а потім злиття файлів ***B*** і ***C*** у файл ***A.*** (Помітимо, що процедура злиття для файлів повністю ілюструється рисунком 9.20.) На першому кроці для розподілу послідовно читається файл Л, і записи ***a\, a3***,..., ***a{n***–1) пишуться у файл ***B,*** а записи ***a2, a4,*** ..., ***an*** - у файл ***C*** (початковий розподіл). Початкове злиття здійснюється над парами ***(a***І, ***a2), (a3,*** <я4), ..., (я(л-І), ***an),*** і результат записується у файл ***A.*** На другому кроці знову послідовно читається файл ***A,*** і у файл ***B*** записуються послідовні пари з непарними номерами, а у файл ***C***– з парними. При злитті утворюються і пишуться у файл ***A*** впорядковані четвірки записів. І так далі. Перед виконанням останнього кроку файл ***A*** міститиме дві впорядковані підпослідовності розміром «/2 кожна. При розподілі перша з них потрапить у файл ***B***, а друга - у файл ***C.*** Після злиття файл ***A*** міститиме повністю впорядковану послідовність записів. У таблиці 9.10 показаний приклад зовнішнього сортування простим злиттям. Зазначимо, що для виконання зовнішнього сортування методом прямого злиття в основній пам'яті вимагається розташувати всього дві змінні - для розміщення чергових записів з файлів ***B*** і С.

Таблиця 4.6 Приклад зовнішнього сортування прямим злиттям

Початковий стан	8 23 5 65 44 33 1 6
Перший крок Розподіл	
Файл В	8 5 44 1
Файл С	23 65 33 6
Злиття: файл А	8 23 5 65 33 44 1 6
Другий крок Розподіл	
Файл В	8 23 33 44
Файл С	5 65 1 6
Злиття: файл А	5 8 23 65 1 6 33 44
Третій крок Розподіл	
Файл В	5 8 23 65
Файл С	1 6 33 44
Злиття: файл А	1 5 6 8 23 33 44 65

9.2.2. Природне злиття

При використанні методу прямого злиття не береться до уваги те, що початковий файл може бути частково відсортованим, тобто містити впорядковані підпослідовності записів. **Серією** називається підпослідовність записів **ai, a(i**+1), ..., **a]** така, що **ак** <= **а(к**+1) для всіх / <= **к** <у, **ai <**

a(i-1) і о/>я(/+]). Метод природного злиття ґрунтується на розпізнаванні серій при розподілі і їх використанні при подальшому злитті.

Як і у разі прямого злиття, сортування виконується за декілька кроків, в кожному з яких спочатку виконується розподіл файла **А** по файлам **В** і С, а потім злиття **В** і **С у** файл **А.** При розподілі розпізнається перша серія записів і переписується у файл **В,** друга — у файл **С** і т.ін. При злитті перша серія записів файлу **В** зливається з першою серією файлу С, друга серія **В** з другою серією **С** і т.ін. Якщо перегляд одного файлу закінчується раніше, ніж перегляд іншого (внаслідок різної кількості серій), то залишок не до кінця переглянутого файла повністю копіюється в кінець файла **А.** Процес завершується, коли у файл і Я залишається тільки одна серія. Приклад сортування файла показаний на рис

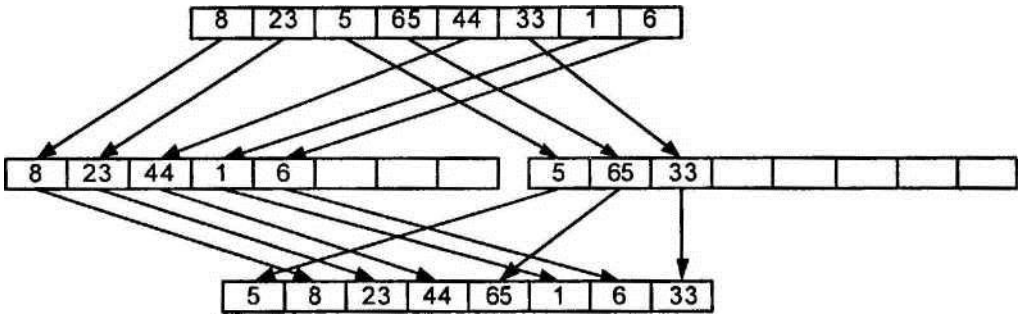


Рис. 4.20. ***Зовнішнє пряме сортування злиттям. Перший крок.***

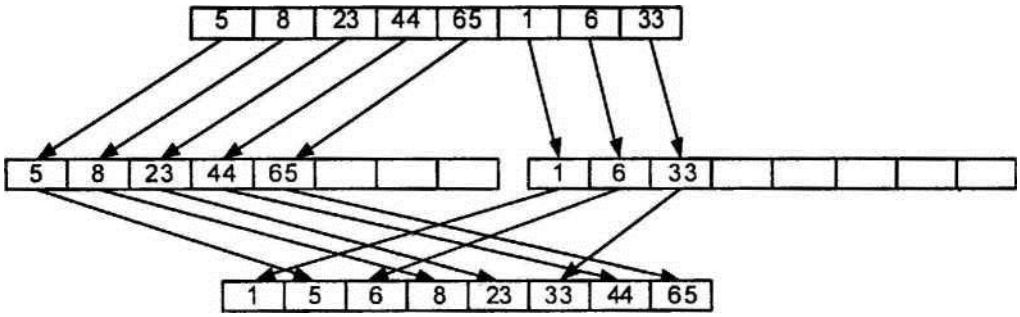


Рис. 4.21. ***Зовнішнє пряме сортування злиттям. Другий крок.***

Очевидно, що кількість зчтувань перезаписів файлів при використанні цього методу буде не більша, ніж при застосуванні методу прямого злиття, а в середньому - менша. З другого боку, збільшується кількість порівнянь за рахунок тих, які потрібні для розпізнавання кінців серій. Крім того, оскільки довжина серій може бути довільною, то максимальний розмір файлів ***В*** і ***С*** може бути близький до розміру файла ***А*.**

9.2.3. Збалансоване багатошляхове злиття

В основі методу зовнішнього сортування збалансованим багатошляховим злиттям є розподіл серій початкового файлу по *т* допоміжних файлів 51, 52, ..., *Вт* і їх злиття в *т* допоміжних файлів CI, С2, ..., *Ст.* На наступному кроці здійснюється злиття файлів CI, С2, ..., *Ст* у файли 51,52, ..*Вт* і т.ін., поки в 51 або CI не утворюється одна серія.

Багатошляхове злиття є природним розвитком ідеї звичного (двошляхового) злиття, ілюстрованого рис. . Приклад тришляхового злиття показаний на рис. .

На рис. показаний простий приклад застосування сортування багато шляховим (багатофазним) злиттям.

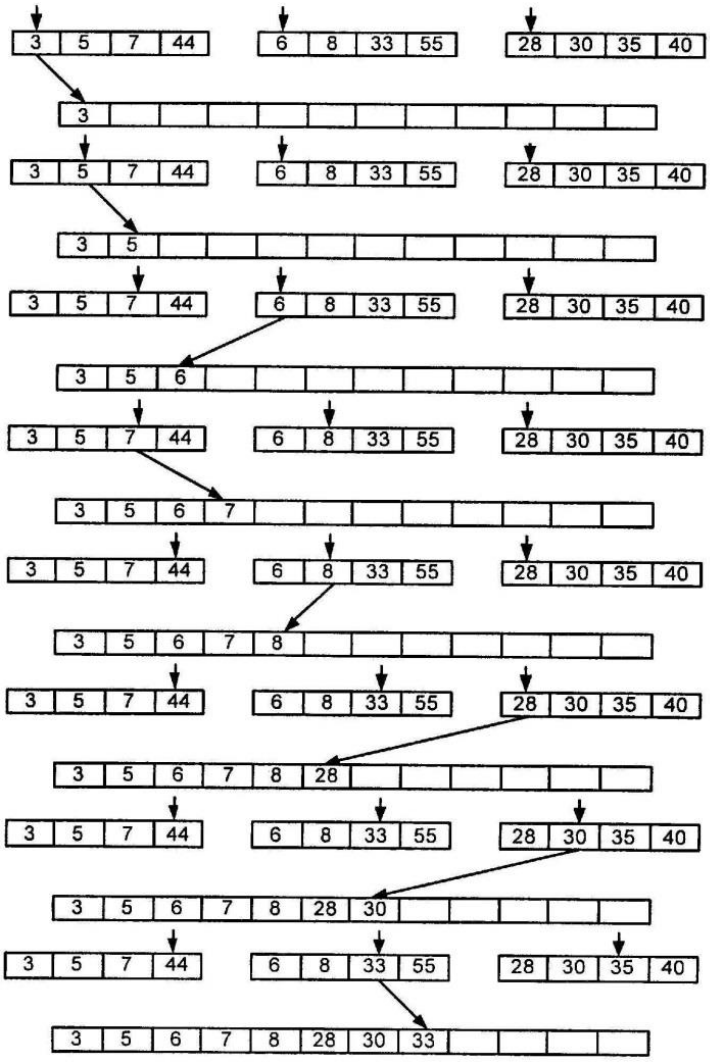


Рис. 4.22. а) Тришляхове злиття. Початок

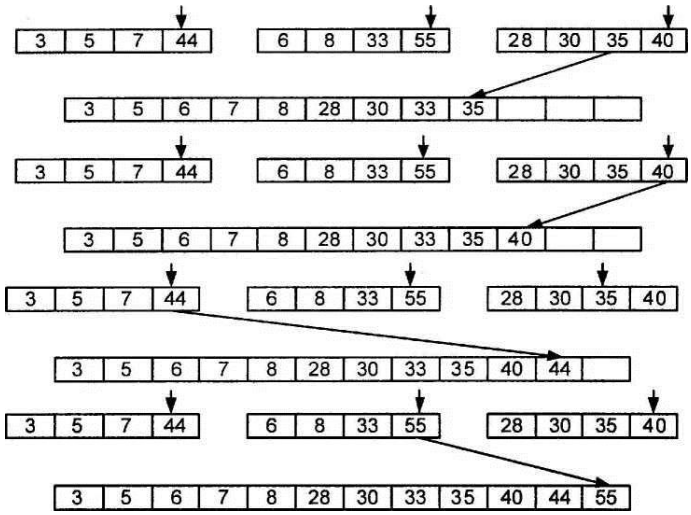


Рис. 4.23. б) Тришляхове злиття. Закінчення.

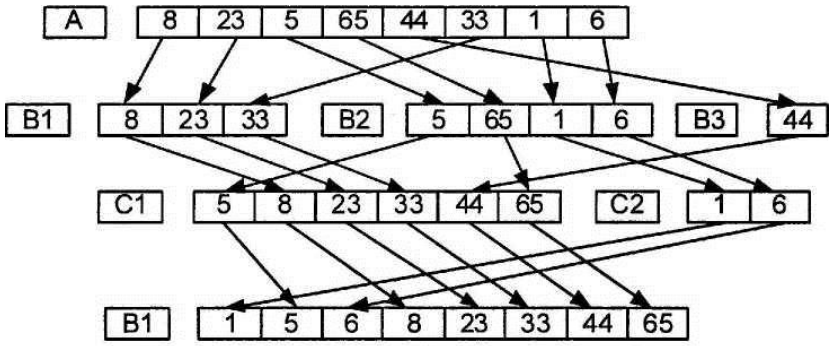


Рис. 9.25. Багатофазне злиття.

Він, зазвичай, дуже тривіальний, щоб продемонструвати декілька кроків виконання алгоритму, проте достатній як ілюстрація загальної ідеї методу. Як показує цей приклад, у міру збільшення довжини серій допоміжні файли з великими номерами (починаючи з номера я) перестають використовуватися, оскільки їм «не дістається» жодної серії. Перевагою сортування збалансованим багатофазним злиттям є те, що кількість проходжень алгоритму оцінюється як 0 (log я) (я - кількість записів у початковому файлі), де логарифм береться по я. Порядок кількості копіювань записів - (log я). Зазвичай, кількість порівнянь не буде меншою, ніж при застосуванні методу простого злиття.

9.2.4. Багатофазне сортування

При використанні розглянутого вище методу збалансованого багатошляхового зовнішнього сортування на кожному кроці приблизно половина допоміжних файлів використовується для введення даних і приблизно стільки ж для виведення злитих серій. Ідея багатофазного сортування полягає у тому, що з наявних т допоміжних файлів (т-1) файл служить для введення злитих послідовностей, а один - для виведення утворюваних серій. Як тільки один з файлів введення стає порожнім, його починають використовувати для виведення серій, одержуваних при злитті серій нового набору (ти-1) файлів. Отже, є перший крок, при якому серії початкового файлу розподіляються по ти-1 допоміжному файлу, а потім виконується багатошляхове злиття серій з (ти-1) файлу, поки в одному з них не утворюється одна серія.

Очевидно, що при довільному початковому розподілі серій по допоміжним файлам алгоритм може не зійтися, оскільки в єдиному непорожньому файлі існуватиме більше, ніж одна серія. Припустимо, наприклад, що використовується три файли 51,52 і 53, і при початковому розподілі у файл 51 вміщені 10 серій, а у файл 52 - 6. При злитті 51 і 52 до моменту, коли ми дійдемо до кінця 52, в 51 залишаться 4 серії, а у 53 потраплять 6 серій. Продовжиться злиття 51 і 53, і при завершенні перегляду 51 у 52 міститимуться 4 серії, а у 53 залишаться 2 серії. Після злиття 52 і 53 в кожному із файлів 51 і 52 міститиметься по 2 серії, яка злитиметься і утворюють 2 серії в 53 при тому, що 51 і 52 - порожні. Отже, алгоритм не зійшовся (таблиця).

Таблиця 4.7 Приклад початкового розподілу серій, при якому трифазне зовнішнє сортування не приводить до требаго результату.

К-сть серій у файлі	К-сть серій у файлі В2	К-сть серій у файлі
10	**6**	**0**
4	**0**	**6**
0	**4**	**2**
2	**2**	**0**
0	**0**	**2**

Оскільки кількість серій у початковому файлі може не забезпечувати можливість такого розподілу серій, застосовується метод додавання порожніх серій, які надалі якомога

рівномірніше розподіляються між проміжними файлами і пізнаються при подальшому злитті. Зрозуміло, що чим менше таких порожніх серій, тобто чим ближча кількість початкових серій за вимогами Фібоначчі, тим ефективніше працює алгоритм.

Тема 4. Поняття жадібного алгоритму. Відмінність між динамічним програмуванням і жадібним алгоритмом. Алгоритми: Краскала, Шеннона-Фано, Хафмана, Пріма.

Поняття жадібного алгоритму

❖ *Жадібний алгоритм - метод оптимізації задач, заснований на тому, що процес ухвалення рішення можна розбити на елементарні кроки, на кожному з яких приймається окреме рішення.*

Рішення, яке приймається на кожному кроці, має бути оптимальним тільки на поточному кроці і прийматися без урахування попередніх або подальших рішень.

Ознаки того, що задачу можна розв'язати за допомогою жадібного алгоритму:

1. задачу можна розбити на підзадачі;
2. величини, що розглядаються в задачі, можна дробити так само, як і задачу, на елементарніші;
3. сума оптимальних рішень для двох підзадач дасть оптимальне рішення для всієї задачі.

Приклад жадібного алгоритму:

```
к:=1;
v[k]:=1;
kol_ver[1 ]:=kol_ver[1 ]+1; s:=0;
while (k<n) do begin
    min:=maxint; for i:=1 to к do for
 j:=1 to n do
                if a[v[i]j]<min then begin
                    min:=a[v[i],]];
                    vi:=v[i];
                    vj:=j;
end;
        f:=true;
        for i:=1 to к do if vj=v[i] then
     f:=false; iff then begin k:=k+1;
     v[k]:=vj;
              kol_ver[vj]:=kol_ver[vj]+1;
          kol_ver[vi]:=kol_ver[vi]+1;        s:=s+a[vi,vj];
          writeln(v[k],'',vi>''',vj); readln; end;
       a[vi,vj]:=maxint;
       a[vj,vi]:=maxint;
end;
```

Загалом неможливо сказати, чи можна отримати оптимальне рішення за допомогою жадібного алгоритму стосовно конкретного завдання. Але є дві особливості, характерні для завдань, які вирішуються за допомогою жадібних алгоритмів: принцип жадібного вибору і властивість оптимальності для підзадач.

Говорять, що до завдання оптимізації застосуємо принцип ***жадібного вибору,*** якщо послідовність локально оптимальних виборів дає глобально оптимальне рішення. У цьому полягає головна відмінність жадібних алгоритмів від динамічного програмування: у другому прораховуються відразу наслідки всіх варіантів.

Щоб довести, що жадібний алгоритм дає оптимум, треба спробувати виконати доведення, яке аналогічне до доведення алгоритму задачі про вибір заявок. Спочатку ми показуємо, що

жадібний вибір на першому кроці не закриває шлях до оптимального розв'язання: для будь-якого розв'язку є інший варіант, узгоджений з жадібним вибором і не гірший від першого. Потім ми показуємо, що розв'язок, який виник після жадібного вибору на першому кроці, аналогічний до початкового. За індукцією витікатиме, що така послідовність жадібних виборів дає оптимальний розв'язок.

Оптимальність для підзадач

Ця властивість говорить про те, що оптимальний розв'язок всієї задачі містить у собі оптимальні розв'язки підзадач. Наприклад, у задачі про вибір заявок можна помітити, що, якщоЛ — оптимальний набір заявок, що містить заявку номер **1, то *A1=A\{1}*** — оптимальний набір заявок для меншої множини заявок 51, що складається з тих заявок, для яких **si** є ***f1.***

Оптимальне розв'язування будується покроково. На кожному кроці часткове розв'язування доповнюється новим елементом, який обирається з допустимої підмножини. Для цього необхідні три процедури: процедура вибору кандидата для розширення часткового розв'язку, процедура визначення допустимості цього кандидата, процедура визначення, чи є цей розширений частковий розв'язок повним розв'язком задачі.

Відмінність між динамічним програмуванням і жадібним алгоритмом

Відмінність між динамічним програмуванням і жадібним алгоритмом можна проілюструвати на прикладі завдання про рюкзак, а точніше, на його дискретному і неперервному формулюванні. Далі ми покажемо, що неперервне завдання вирішується жадібним методом, дискретне ж вимагає тоншого, динамічнішого рішення.

Дискретне завдання про рюкзак. Злодій заліз на склад, на якому зберігається **п** речей. Кожна річ коштує 5 доларів і важить **10** кілограмів. Злодій хоче понести товару на максимальну суму, проте він не може підняти більш **4** кілограмів (всі числа цілі). Що він може покласти в рюкзак?

Неперервне завдання про рюкзак. Тепер злодій уміє дробити товари і укладати в рюкзак тільки їх частки, а не обов'язково ціле. Зазвичай в дискретному завданні йде мова про золоті злитки різної проби, а в неперервному — про золотий пісок.

1. ПРИКЛАДИ ЖАДІБНИХ АЛГОРИТМІВ 10.3.1. Алгоритм Краскала

Алгоритм Краскала знаходить контурний ліс мінімальної ваги у заданому графі.

Спочатку опрацьована множина ребер встановлюється порожньою. Потім, доки це можливо, виконується така операція: зі всіх ребер, додавання яких до вже наявної множини не викличе появу в ньому циклу, вибирається ребро мінімальної ваги і додається до множини опрацьованих ребер. Коли таких ребер більше немає, алгоритм завершений. Підграф графу, що містить усі його вершини і знайдену множину ребер, є його контурним лісом мінімальної ваги.

До початку роботи алгоритму необхідно відсортувати ребра за вагою, це вимагає 0 **(E \ogiE))** часу. Після цього компоненти зв'язності зручно зберігати у вигляді системи непересічної множини. Всі операції у такому разі триватимуть **(E (E, V)).**

10.3.2. Алгоритм Шеннона-Фано

Алгоритм Шеннона-Фано — один з перших алгоритмів стискання, який вперше сформулювали американські вчені Шеннон і Фано. Алгоритм використовує коди змінної довжини: символ, що часто зустрічається, позначається кодом меншої довжини, а символ, що рідко зустрічається — кодом більшої довжини. Коди Шеннона-Фано префіксні, тобто, ніяке кодове слово не є префіксом будь-якого іншого. Ця властивість дозволяє однозначно декодувати будь-яку послідовність кодових слів.

Алгоритм обчислення кодів Шеннона-Фано

Код Шеннона-Фано будується за допомогою дерева. Побудова цього дерева починається з кореня. Вся множина кодованих елементів відповідає кореню дерева (вершині першого рівня). Вона розбивається на дві підмножини з приблизно однаковими сумарними ймовірностями. Ці підмножини відповідають двом вершинам другого рівня, які з'єднуються з коренем. Далі кожна з цих підмножин розбивається на дві підмножини з приблизно однаковою сумарною ймовірністю. їм відповідають вершини третього рівня. Якщо підмножина містить єдиний елемент, то йому відповідає кінцева вершина кодового дерева; така підмножина розбиттю не підлягає. Так само

чинимо доти, поки не отримаємо всі кінцеві вершини. Гілки кодового дерева позначаємо символами 1 і 0. При побудові коду Шеннона-Фано розбиття множини елементів може бути виконане декількома способами. Вибір розбиття на рівні п може погіршити варіанти розбиття на наступному рівні (н+1) і привести до погіршення коду загалом. Іншими словами, оптимальна поведінка на кожному кроці шляху ще не гарантує оптимальності всієї сукупності дій. Тому код Шеннона-Фано не є оптимальним у загальному сенсі, хоч і дає оптимальні результати при деяких розподілах ймовірності. Для одного і того ж розподілу ймовірності можна побудувати, взагалі кажучи, декілька кодів Шеннона-Фано, і всі вони можуть дати різні результати.

Якщо побудувати всі можливі коди Шеннона-Фано для заданого розподілу ймовірності, то серед них будуть і всі оптимальні коди.

10.3.3. Алгоритм Хафмана

Алгоритм Хафмана (англ. Нийтпап) — адаптивний жадібний алгоритм оптимального префіксного кодування алфавіту з мінімальною надмірністю. Був розроблений 1952 року доктором Массачусетського технологічного інституту Девідом Хафманом. На сьогодні використовується в багатьох програмах стискання даних.

На відміну від алгоритму Шеннона-Фано, алгоритм Хафмана залишається завжди оптимальним і для вторинних алфавітів $\mathbf{т_2}$ з більше ніж двома символами.

Цей метод кодування складається з двох основних етапів:

1) побудова оптимального кодового дерева.

2) побудова відображення код->символ на основі побудованого дерева. Алгоритм

1. Визначається ймовірність появи символів первинного алфавіту в початковому тексті (якщо вони не задані заздалегідь).

2. Символи первинного алфавіту /лі виписують у порядку зменшення ймовірності.

3.Останні $\mathbf{п_0}$ символів об’єднують у новий символ, ймовірність якого дорівнює

сумарній ймовірності цих символів, видаляють ці символи і вставляють новий символ у список останніх на відповідне місце (за ймовірністю). визначається із системи:

д**е а**—ціле число, **т**\ і /и, — потужність первинного і вторинного алфавіту відповідно.

4. Останні $\mathbf{т_2}$ символів знову об’єднують в один і вставляють його на відповідну позицію, заздалегідь видаливши символи, що увійшли до об’єднання.

5. Попередній крок повторюють доти, доки сума всіх $\mathbf{т_2}$ символів не стане рівною 1.

Цей процес можна подати як побудову дерева, корінь якого — символ із ймовірністю

1, який отримано при об’єднанні символів з останнього кроку, його $\mathbf{т_2}$ нащадків — символи з попереднього кроку і так далі.

Кожні $\mathbf{т_2}$ елементів, що розташовані на одному рівні, нумеруються від 0 до $\mathbf{т_2}$ -1. Коди отримують зі шляхів (від першого нащадка кореня і до листка). При декодуванні можна використовувати те ж саме дерево, прочитується по одній цифрі і робиться крок по дереві, доки не досягається листка — годі виводиться символ, що стоїть у листку і відбувається повернення в корінь.

Кодування Шеннона-Фано є досить старим методом стискання, і на сьогодні воно не має особливого практичного застосування. У більшості випадків довжина стисненої послідовності за заданим методом дорівнює довжині стисненої послідовності з використанням кодування Хафмана. Але на деяких послідовностях все ж формуються неоптимальні коди Шеннона-Фано, тому стискання методом Хафмана прийнято вважати ефективнішим.

Приклад реалізації

Приклад реалізації алгоритму Хафмана на мові C++ (замість впорядковування піддерев кожного разу шукаємо в масиві дерево з мінімальною вагою) (текст програми взято з Вікіпедії).

```
// вага цього символу

this.ieaf = leaf; this.character = character; this.weight = weight;

Class Tree {
```

```
public Tree child0;       // нащадки «0» і «1»
public Tree childl;
public boolean leaf;      // ознака листка дерева
public int character;     // вхідний символ
public int weight;        // вага цього символу

public Tree() {}
public Tree(int character, int weight, boolean leaf)
{
   Обхід дерева з ґенерацією кодів
•        «Роздрукувати» листове дерево і записати код Хафмана в масив
•        Рекурсивно обійти ліве піддерево (з ґенеруванням коду).
•        Рекурсивно обійти праве піддерево.
*/
public void traverse(String code, Huffman h)
{
if (leaf)
{
System.out.println((char)character +» «+ weight +» «+ code); h.code[character] = code;
}
if (childO != null) child0.traverse(code + «0», h); if (childl != null) child 1.traverse(code + «1», h);
}
}
class Huffman
{
public static final intALPHABETSIZE = 256;
Tree[] tree = new Tree[ALPHABETSIZE]; // робочий масив дерев int weightsf] = new int[ALPHABETSIZE]; II ваги символів
public String[] code = new String[ALPHABETSIZE]; // коди Хафмана private intgetLowestTree(int used)
    { // шукаємо «найлегше» дерево int min=0;
    for (int i=1; i<used; i++)
    if (tree[i].weight < tree[min].weight) min = i;
    return min;
    }
    public void growTree( int[] data)
    { // нарощуємо дерево
        for (int i=0; i<data.length; I++) // шукаємо ваги символів weights[data[i]]++;
    // заповнюємо масив з «листкових» дерев
    int used = 0;                          //з використаними символами
    for (int c=0; з < ALPHABETSIZE; C++)
    {
        int w = weights[c];
        if (w != 0) tree[used++]= new Tree(з, w, true);
    }
    while (used > 1)
    {                                      // парами зливаємо легкі гілки
        int min = getLowestTree( used );   // шукаємо першу гілку
        int weightO = tree[min].weight;
        Tree temp = new Tree();            // створюємо нове дерево
        temp.childO = tree[min];           // і прищеплюємо першу гілку
        tree[min]= tree[—used];            // на місце першої гілки поміщаємо
        // останнє дерево в списку
```

```
            min = getLowestTree( used );          // шукаємо 2 гілку і
            temp.childl = treefmin];          // прищеплюємо їїдо нового дерева
            temp.weight = weightO + tree[min].weight; // рахуємо вагу нового
                                                        //дерева
            tree[min]= temp;           // нове дерево поміщаємо на місце 2 гілки
        }                              // отримали 1 дерево Хафмана
        }
    public void makeCodeQ { // запускаємо обчислення кодів Хафмана tree[0].traverse(«», this);
        }
        public String coder( int[] d a t a )
        { // кодує дані рядка з 1 і 0 String str = «»;
        for (int i=0; i<data.length; i++) str += code[data[i]]; return str;
             public String decoder(String data)
         {
                String str=»»;       // перевіряємо в циклі дані на входження
                int 1 = 0;           // коду, якщо так, то відкидаємо його ...
                while(data.length() > 0)
                {
                    for (int c=0; c < ALPHABETSIZE; c++)
                    {
                        if (weights[c]>0 && data.startsWith(code[c]))
                        {
                            data = data.substring(code[c].length(), data.IengthQ); str += (char)c;
                        }
                    }
                }
                return str;
         }
    }
         public class HuffmanTest { // тест і
      демонстрація
             public static void main(String[] args)
         {
                Huffman h = new HuffmanQ;
                String str = «to be or not to be?»; int data[] = new
             int[str.length()]; for(int i=0; i<str.length(); i++)
             data[i]= str.charAt(i); h.growTree( data);
             h.makeCodeQ; str = h.coder(data);
                System.out.println(str);
                System.out.println(h.decoder( str));
         }
    }
```

Змістовий модуль 5. Методи обчислень. Основні проблеми чисельного розв'язання задач. Системи лінійних алгебраїчних рівнянь.

Тема 1. Основні поняття чисельних методів. Класифікація похибок. Абсолютна і відносна похибки, середня квадратична похибка, поширення похибок. Підвищення точності обчислень.

Серед інформаційних технологій, які лежать в основі всіх напрямів підготовки спеціалістів з комп'ютерних технологій, особливе місце займає математичне моделювання. При цьому під *математичною моделлю* фізичної системи, об'єкта або процесу звичайно розуміють сукупність математичних співвідношень (формул, рівнянь, логічних виразів), які визначають характеристики стану і властивості системи, об'єкта і процесу та їх функціонування залежно від параметрів їх компонентів, початкових умов, вхідних збуджень і часу. Загалом математична модель описує функціональну залежність між вихідними залежними змінними, через які відображається функціонування системи, незалежними (такими, як час) і змінюваними змінними (такими, як параметри компонентів, геометричні розміри та ін.), а також вхідними збудженнями, прикладеними до системи.

Згадана функціональна залежність, що відображається математичною моделлю, може бути явною чи неявною, тобто може бути зображена або як просте алгебраїчне співвідношення, або ж як велика за розміром сумісна система диференціально-алгебраїчних рівнянь. До того, як обчислювальна техніка набула широкого розповсюдження, переважали явні функціональні моделі низьких порядків, пристосовані до можливостей розрахунків ручним способом або розрахунків з малим ступенем механізації (логарифмічна лінійка, арифмометр та ін.).

Саме вони і є сьогодні теоретичною основою багатьох інженерних та природничих дисциплін, яка дозволяє під час проектування проводити наближені розрахунки з точністю до кількох десятих відсотка з подальшим обов'язковим макетуванням проектованого об'єкта та його експериментальним доведенням до потрібних параметрів, внаслідок чого розробка нового виробу розтягується на багато років.

Сучасні комп'ютери дозволяють у багатьох випадках відмовитися від натурного макетування проектованих виробів, замінивши його математичним моделюванням (обчислювальним експериментом), що дуже важливо, коли натурне макетування складне або практично неможливе (наприклад, моделювання прориву дамби, переміщення всюдиходу поверхнею Марса та ін.). Але при цьому повинна бути істотно підвищена точність математичних моделей об'єктів та систем, що враховують багато фізичних ефектів та дестабілізуючих чинників, якими раніше нехтували. В результаті розмірність і складність математичних моделей істотно зростають, а їх розв'язання в аналітичному вигляді стає неможливим. Це звичайний для сучасної науки і техніки компроміс, що полягає в отриманні нової якості одного параметра (висока точність обчислювального експерименту і відмова від натурного макетування) за рахунок зменшення чи ускладнення іншого параметра (відмова від звичних для вищої математики аналітичних рішень).

Для кожної математичної моделі звичайно формулюється математична задача. У загальному випадку, коли функціональна залежності для множини вхідних даних (значення незалежних та змінюваних змінних і вхідних збуджень), що виступають як множина аргументів, задана неявно, за допомогою математичної моделі необхідно визначити множину вихідних залежних змінних, що виступають як множина значень функцій. При цьому відповідно до виду математичної моделі розрізняють такі базові типи математичних задач:

- розв'язання системи лінійних (в загальному випадку лінеаризованих) рівнянь;
- розв'язання нелінійних алгебраїчних рівнянь;
- апроксимація масиву даних або складної функції набором стандартних, більш простих функцій;
- чисельне інтегрування і диференціювання;
- розв'язання систем звичайних диференціальних рівнянь;
- розв'язання диференціальних рівнянь в частинних похідних;
- розв'язання інтегральних рівнянь.

Прості математичні задачі малої розмірності, що вивчаються в курсі вищої математики, допускають можливість отримання аналітичних рішень. Складні математичні моделі великої розмірності вимагають застосування чисельних методів, що вивчаються в даному курсі.

Чисельні методи — це математичний інструментарій, за допомогою якого математична задача формулюється у вигляді, зручному для розв'язання на комп'ютері. У такому разі говорять про перетворення математичної задачі в *обчислювальну* задачу. При цьому послідовність виконання необхідних арифметичних і логічних операцій визначається *алгоритмом* її розв'язання. Алгоритм повинен бути рекурсивним і складатися з відносно невеликих блоків, які багаторазово виконуються для різних вхідних даних.

Слід зазначити, що з появою швидких та потужних цифрових комп'ютерів роль чисельних методів для розв'язання наукових та інженерних задач значно зросла. І хоча аналітичні методи розв'язання математичних задач, як і раніше, дуже важливі, чисельні методи істотно розширюють можливості розв'язання наукових та інженерних задач, не дивлячись на те, що самі рівняння математичних моделей з ускладненням структури сучасних виробів стають погано обумовленими та жорсткими, що істотно ускладнює їх розв'язування. Узявши виконання рутинних обчислень на себе, комп'ютери звільняють час вченого або інженера для творчості: формулювання задач і генерування гіпотез, аналізу та інтерпретації результатів розрахунку тощо.

Чисельні методи забезпечують системний формалізований підхід до розв'язання математичних задач. Проте за умов їх ефективного використання окрім уміння присутня і деяка частка мистецтва, що залежить від здібностей користувача, оскільки для розв'язання кожної математичної задачі існує декілька можливих чисельних методів і їх програмних реалізацій для різних типів комп'ютерів. На жаль, для обрання ефективного способу розв'язання поставленої задачі лише інтуїції замало, потрібні глибокі знання і певні навички. Існує декілька переконливих причин, що мотивують необхідність глибокого вивчення чисельних методів майбутніми фахівцями у галузі комп'ютерно-системної інженерії та прикладної математики.

Чисельні методи є надзвичайно потужним інструментарієм для розв'язання проблемних задач, що описуються довільними нелінійними диференціально-алгебраїчними рівняннями великої розмірності, для яких в даний час не існує аналітичних рішень. Освоївши такі методи, майбутній фахівець набуває здібностей до системного аналізу через математичне моделювання найскладніших задач сучасної науки і техніки.

У своїй майбутній професійній діяльності такий фахівець у першу чергу орієнтуватиметься на використання пакетів сучасних обчислювальних програм, причому те, наскільки правильно він буде їх застосовувати, безпосередньо залежатиме від знання і розуміння ним особливостей і обмежень, властивих чисельним методам, що реалізовані в пакеті. Може трапитися, що одна й та сама математична задача за допомогою певного програмно-технічного комплексу буде одним фахівцем успішно розв'язана, а іншим — ні, оскільки в сучасних пакетах передбачено їх налагоджування для конкретної задачі.

Може з'ясуватися, що низку задач неможливо розв'язати з використанням наявних пакетів програм. Якщо майбутній фахівець знає чисельні методи і володіє навичками програмування, він буде в змозі самостійно провести розробку необхідного алгоритму і програмно його реалізувати, вбудувавши в обчислювальний комплекс.

Вивчення чисельних методів стимулює освоєння самих комп'ютерів, оскільки найкращим способом навчитися програмувати є написання комп'ютерних програм власноруч. Правильно застосувавши чисельні методи, майбутній фахівець зможе пересвідчитися у тому, що комп'ютери успішно розв'язують його професійні задачі. При цьому він сам відчує вплив похибок обчислень на результат і навчиться контролювати ці похибки.

Вивчення чисельних методів сприяє також переосмисленню і більш глибокому розумінню математики в цілому, оскільки однією із задач чисельних методів є зведення методів вищої математики до виконання простих арифметичних операцій.

Хоча існує безліч чисельних методів, усі вони (як і алгоритми, що їм відповідають) мають багато спільних властивостей і характеристик. Чисельні методи:

- передбачають проведення великої кількості рутинних арифметичних обчислень за допомогою рекурсивних співвідношень, що використовуються для організації *ітерацій,* тобто повторюваних циклів обчислень зі зміненими початковими умовами для поліпшення результату;
- направлені на локальне спрощення задачі, коли, наприклад, використовувані нелінійні залежності лінеаризуються за допомогою своїх обчислених похідних або похідні замінюються різницевими апроксимаціями;
- значно залежать від близкості початкового наближення (або декількох наближень), необхідного для початку обчислень до розв'язку, від властивостей нелінійних функцій, які використовуються в математичних моделях, що накладає обмеження (для забезпечення єдиного розв'язку) на їх диференційованість, на швидкість зміни функцій та ін.;

Чисельні методи характеризуються:

- різною *швидкістю збіжності,* тобто числом ітерацій, виконання яких необхідне для отримання заданої точності розв'язку;
- різною *стійкістю,* тобто збереженням достовірності розв'язку під час подальших ітерацій;
- різною *точністю* отримуваного розв'язку в разі виконання однакового числа ітерацій або циклів обчислень.

Чисельні методи розрізняються:

- за широтою і легкістю застосування, тобто за ступенем своєї *універсальності* та *інваріантності* для розв'язання різних математичних задач;
- за *складністю* їх програмування;
- за можливостями використання у разі їх реалізації наявних бібліотек функцій і процедур, створених для підтримки різних алгоритмічних мов;
- за *ступенем чутливості* до погано обумовлених (або некоректних) математичних задач, коли малим змінам вхідних даних можуть відповідати великі зміни розв'язку.

Основні проблеми чисельного розв'язання задач

При застосуванні чисельних методів розв'язки задач виявляються, як правило, наближеними. Пояснюється це в багатьох випадках тим, що точні методи їх розв'язання дотепер невідомі. Крім того, навіть при застосуванні точного методу задовольняються наближеним розв'язком, зокрема, з таких причин:

— точний розв'язок виявляється трудомістким; тоді як наближений при істотно меншому об'ємі обчислень виявляється цілком прийнятним за своїм характером;

— точність отриманого результату не відіграє істотної ролі, тому що в будь-якому разі заокруглюється до цілого числа (наприклад, при визначенні кількості механізмів, необхідних для виконання даного обсягу робіт).

Наближений розв'язок задачі повинен «не набагато відрізнятися» від точного розв'язку, інакше ним не можна скористатися з конкретною метою. Що означає термін «не набагато відрізняється» або, інакше кажучи, що варто розуміти під неточністю (наближеністю) розв'язку? Кожен чисельний метод дозволяє оцінювати ступінь неточності розв'язку, одержуваного цим методом. У курсі чисельних методів ступінь неточності розв'язку характеризується поняттям похибки розв'язку. Потрібно зазначити, що теорія похибок є одним із основних розділів обчислювальної математики. Очевидно, що відхилення наближеного результату від точного напряму залежить від коректності поставленої задачі та від наявних вхідних даних. Тому актуальним є дослідження збіжності наближеного розв'язку, що пропонує чисельний алгоритм, до точного розв'язку поставленої задачі .

Таким чином, основними проблемами чисельного розв'язання задач можна вважати:

-проблему оцінки похибки наближеного розв'язку; -проблему коректності та обумовленості поставленої задачі; -проблему збіжності наближеного методу до точного.

1.1 Класифікація похибок

При розв'язанні прикладних задач дуже важливо мати уявлення про точність отриманих результатів. Похибки, що можуть бути закладені в таких результатах, утворюються з багатьох причин.

Можна визначити чотири основні джерела похибок результату чисельного методу:

1) вхідні дані;
2) математична модель;
3) наближений метод;
4) округлення при розрахунках. Проаналізуємо їх.

Похибки вхідних даних

Точні значення багатьох величин практично ніколи не можуть бути введені в процес обчислень, наприклад, ірраціональних величин π, e, $\sqrt{2}$ та ін. У цих випадках неминучі похибки округлення. При розв'язанні багатьох задач за вхідні беруться значення величин, отриманих з експерименту. З багатьох причин, у тому числі обмеженої точності вимірювальної апаратури і впливу різних випадкових чинників, експериментальні дані завжди мають похибки того або іншого порядку. Так, точність вимірювання температури, відстані, об'єму, ваги залежить від досконалості застосовуваних вимірювальних приладів. Похибки можуть бути у вхідних даних, отриманих теоретично. Природно, що вони впливають на результати розв'язку задачі, однак жодним чином їх усунути не можна. Тому похибки такого типу часто називають *неусувними.*

Похибки математичної моделі

Необхідно зазначити, що в більшості випадків фахівцю вдається підібрати для розв'язання задачі наближений метод, що дозволяє одержати цілком задовільні за ступенем точності результати. Однак розв'язувана задача є не тим реальним завданням, з яким фахівцю доводиться мати справу, а його спрощеною математичною моделлю. Так, при розрахунку авіаційного двигуна або несучої конструкції промислової споруди неможливо ввести до розгляду їх реальну надзвичайно складну форму, врахувати наявність усіх отворів, деталей сполучення і т.п. При визначенні оптимального складу персоналу універмагу, кас попереднього продажу залізничних квитків доводиться припускати, що покупці приходять через рівні проміжки часу, час обслуговування кожного з них однаковий і таке інше.

Розв'язок реальної задачі не збігається із результатом, отриманим при розгляді її математичної моделі навіть із застосуванням точних методів розв'язку, а похибки, що виникають при цьому, можна назвати *похибками математичного моделювання.*

Похибки наближеного методу

У випадку, коли розв'язати задачу точно неможливо, доводиться застосовувати різні наближені методи. Результати такого підходу завчасно містять похибки, характер яких залежить від використовуваного наближеного методу *(похибки методу).*

При застосуванні наближених методів розв'язання задач, наприклад ітераційних, точні значення шуканих величин можуть бути отримані тільки після виконання нескінченного числа етапів обчислень, що практично здійснити неможливо. Доводиться задовольнятися певним числом етапів і відповідними наближеними результатами із так званими *залишковими похибками.*

Похибки заокруглень при розрахунках При реалізації на ЕОМ алгоритмів, що містять велику кількість операцій множення і ділення, типовими є *похибки округлення.* При виконанні операцій множення кількість розрядів може зрости настільки, що всі вони вже не можуть бути розміщені в елементах запам'ятовуючих пристроїв ЕОМ.

Частину розрядів праворуч доводиться відкидати, округляти числа. Сам по собі процес округлення числа не обов'язково призводить до внесення в нього якої-небудь істотної похибки. Так, при обчисленні зі звичайною точністю в сучасних ЕОМ можна утримувати, наприклад, дев'ять десяткових розрядів. Природно, що простим відкиданням в ЕОМ десятого і наступних розрядів ми вносимо в число лише дуже незначні зміни. Порівняємо дванадцятирозрядне число 1000000,00297 і округлене дев'ятирозрядне число 1000000,00. Внесена в результаті округлення похибка становить величину 0,00297. Однак у процесі виконання великої кількості арифметичних операцій похибки, послідовно накопичуючись, породжують нові. Таке

нагромадження похибок округлення може призвести до дуже істотних помилок в остаточних результатах.

Похибки округлення особливо доводиться враховувати при реалізації нестійких обчислювальних процесів, у яких незначні похибки у вихідних даних або результатах проміжних обчислень можуть призвести до істотних помилок у остаточному результаті.

Приклад. Нехай необхідно обчислити величину *c* за формулою

$$c = a - b, \qquad (1.1)$$

де $a = 139{,}27$; $b = 138{,}97$. Одержимо $c = 0{,}3$.

Припустимо, що величини *a* і *b* обчислені з похибками, що не перевищують 1% їх точних значень, a=140,62, b=37,62. Обчислюючи величину *c* за формулою (1.1) із наближеними значеннями, одержимо c=140,62-137,62=3,0. Отже, похибки в обчисленні вихідних величин *a* і *b* призвели до десятикратного збільшення числа *c* .

1.2 Абсолютна і відносна похибки

Абсолютна похибка - це модуль різниці між відповідним точним значенням розглянутої величини *A* і наближеним її значенням a. Вона має вигляд

$$\Delta = |A - a|. \qquad (1.2)$$

Безпосередньо за значенням абсолютної похибки досить важко робити висновок про ступінь розбіжності між точним значенням *A* величини і його наближеним значенням. Так, похибка 2м цілком припустима при визначенні відстані між Києвом і Сумами та абсолютно неприпустима при вимірюванні розмірів кімнати. Тому застосовується ще одна характеристика наближених величин — їх відносна похибка.

Відносною похибкою δ наближеного значення величини, точне значення якої дорівнює *A,* називається відношення його абсолютної похибки A до модуля точного значення, тобто

$$\delta = \frac{\Delta}{|A|} \qquad (1.3)$$

Наприклад, нехай в результаті вимірювання довжини бігової доріжки отримано значення a=99,1м. Точне значення цієї величини $A = 100$м. Абсолютна похибка $\Delta = |100 - 99{,}1| = 0{,}9$. Відносна похибка за формулою (1.3)

становить $\delta = \frac{0{,}9}{|100|} = 0{,}009$.

Із формул (1.2)—(1.3) бачимо, що абсолютна похибка має розмірність оцінюваних цією похибкою величин, відносна похибка завжди безрозмірна.

Величини Δ і δ можуть бути обчислені точно лише в тих випадках, коли відоме не тільки наближене числове значення розглянутої величини, але і її точне значення. Останнє, однак, можливе далеко не у всіх випадках. Крім того, часто доводиться аналізувати похибки деякої множини наближених величин, наприклад, похибки вимірювання розмірів серії виготовлених деталей, викликані недосконалістю застосовуваних вимірювальних інструментів. Якість серії вимірювань для всіх деталей може оцінюватися найбільшою за модулем величиною абсолютної або відносної похибки їх розмірів. Тому часто вводяться поняття граничних абсолютної та відносної похибок.

За *граничну абсолютну похибку* Δ * наближеного числа може бути взяте будь-яке число, не менше абсолютної похибки цього числа,

$$\Delta^* \geq \Delta . \qquad (1.4)$$

Аналогічно за *граничну відносну похибку δ* * наближеного числа може бути взяте будь-яке число, що задовольняє умову

$$\delta^* \geq \delta. \qquad (1.5)$$

При аналізі серії вимірювань за Δ * і *δ* * беремо найбільші з отриманих відповідних значень Δ і *δ* і тим самим визначаємо межі, всередині яких знаходяться відповідні похибки.

Значущими цифрами числа *a* називають усі цифри в його записі, починаючи з першої ненульової зліва. Значущу цифру числа *a* називають *правильною,* якщо абсолютна похибка числа не перевищує одиниці відповідного цій цифрі розряду.

Приклад 1. Для ряду $\sum_{n=0}^{n} \frac{72}{n^2+5n+4}$ знайти суму S аналітично. Обчислити значення часткових сум ряду $S_N = \sum_{n=0}^{n} a_n$

і знайти величину похибки при значеннях N=10,10^2, 10^3,10^4,10^5. Побудувати гістограму залежності правильних цифр результату від N.

Знайдемо точну суму цього ряду:

$$S_N = \sum_{n=0}^{N} \frac{72}{n^2+5n+4} = \sum_{n=0}^{N} \frac{72}{(n+1)(n+4)} =$$

$$= 72 \cdot \sum_{n=0}^{N} \frac{1}{3} \cdot \left(\frac{1}{n+1} - \frac{1}{n+4} \right) = = 24 \cdot \left(1 + \frac{1}{2} + \frac{1}{3} - \frac{1}{N+2} - \frac{1}{N+3} - \frac{1}{N+4} \right),$$

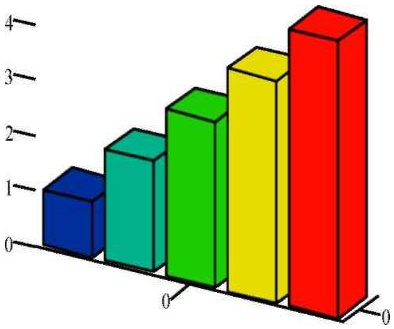
Отже, $S = \lim_{N\to\infty} S_N = 44$. Уведемо функцію часткових сум

$S(N) = \sum_{n=0}^{N} \frac{72}{n^2+5n+4}$. Тоді абсолютну похибку можна визначити за допомогою функції $d(N) = |S(N) - S|$.

Результати обчислювального експерименту

N	Значення частк. суми ряду S(N)	Абсолютна похибка *d(N)*	Кільк. правил. цифр M_i
10	S(10)=38.439560439	d(10) =5.56	$M_1 = 1$
$_2$ 10	S(100)=43.3009269	d (100)=0.699	$M_2 = {}^2$
$_3$ 10	S(1000)=43.9282153	d (1000)=0.072	$M_3 = {}^3$
$_4$ 10	S(10000)=43.992802	d (10000)=0.0072	$M_4 = 4$
$_5$ 10	S(100000)=43.9992802 1599	d (100000)=0.000 72	$M_5 = 5$

Висновок. Як бачимо з наведеного обчислювального експерименту, збільшення числа членів ряду в 10 разів порівняно з попереднім випадком збільшує число правильних цифр у відповіді на гістограмі



Приклад 2. Для матриці

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$$

Розв’язати питання про існування оберненої матриці в таких випадках:

1) елементи матриці задані точно;
2) елементи матриці задані наближено з відносною похибкою

а) $\delta = \alpha\%$ та b) $\delta = \beta\%$.

Знайти відносну похибку результату.

Це питання вирішується шляхом знаходження визначника й порівняння його з нулем. У випадку, коли елементи визначника задані точно, варто обчислити визначник і правильно відповісти на поставлене в задачі питання.

У випадку, коли елементи визначника задані наближено з відносною похибкою 5, питання є складнішим. Нехай елементи матриці позначені через a_{ij} . Тоді кожен елемент матриці a_{ij} тепер уже не дорівнює конкретному значенню, а може набувати будь-якого значення з відрізка [$a_{ij}(1-\delta)$; a_{ij} $(1+\delta)$], якщо $a_{ij} > 0$, і з відрізка [$a_{ij}(1+\delta)$; a^ $(1-\delta)$], якщо $a_{ij} < 0$. Множина всіх можливих значень елементів матриці являє собою замкнену обмежену множину в 9-вимірному просторі. Сам визначник є неперервною й диференційованою функцією 9 змінних - елементів матриці a_{ij}. За відомою теоремою Вейєрштрасса ця функція досягає на зазначеній множині свого найбільшого та найменшого значень *M* і т. Якщо відрізок [*m, M*] не містить точку 0 , то це означає, що при будь-яких припустимих значеннях елементів матриці *a* визначник не набуває значення 0. Якщо ж точка 0 належить відрізку [т, М], таке твердження буде неправомірним. Буде мати місце невизначеність.

З’ясувати *m* і *M* допомагають наступні міркування. Як функція своїх аргументів (елементів матриці a_{ij}.) визначник має таку властивість (принцип максимуму): ця функція досягає свого найбільшого і найменшого значень завжди на границі області . Більше того, можна довести, що ці значення досягаються в точках, координати яких мають вигляд (1±5).

Таких точок 2^9 =512. У кожній з них варто обчислити визначник, а потім вибрати з отриманих значень найбільше та найменше. Це й будуть числа *M* і *m.*

1.1 Середні квадратичні похибки

Нехай передбачається проведення серії вимірів деякої величини *X*. У кожному з вимірів буде отримане якесь її значення, причому залежно від точності приладу, зокрема, ці значення будуть знаходитися в деякому інтервалі, загальне їх число скінченне. Позначимо ці значення $x_1, x_2, \ldots, x_n$, їх ймовірності $p_1, p_2, \ldots p_n$. Оскільки заздалегідь невідомо, яке значення величини *X* буде отримано в кожному вимірі, ця величина є випадковою.

Математичне очікування *X* виражається формулою

$$M[X] = \sum_{i=1}^{n} x_i\, p_i \qquad (1.6)$$

Про якість вимірів, тобто ступінь розкиду помилок виміру, можна роб висновкиити за розмірами дисперсії, або середнього квадратичного відхилення випадкової величини:

$$D[X] = \delta_x^2 = \sum_{i=1}^{n} p_i (x_i - M[X])^2. \qquad (1.7)$$

Величина c_x називається в теорії похибок *середньою квадратичною похибкою* вимірювання.

Якщо результати вимірювання є незалежними, тобто результат довільного виміру не залежить від того, які результати отримані в інших вимірах, для них прийнятні теореми Чебишева і

Бернуллі. Зокрема, бувають наступні припущення.

1 Якщо випадкова величина X набуває тільки невід'ємних значень, частина яких менша деякого додатного числа a, то

$$p[(X<a)]\geq 1-\frac{M[X]}{a}. \qquad (1.8)$$

2 Якщо a > 0, то

$$p[|(X-M[X]|<a)] \geq 1-\frac{\delta_x^2}{a} \qquad (1.9)$$

Відзначимо, що формулою (1.7) користуються для обчислення середніх квадратичних похибок і в детермінованих процесах.

де A —точне значення числа X, а A_i — абсолютні похибки.

1.2 Поширення похибок

Важливим у чисельному аналізі є питання про те, як помилка, що виникла у визначеному місці в ході обчислень, поширюється далі, тобто чи стає її вплив більшим або меншим залежно від того, як виконуються наступні операції. Сформулюємо деякі правила оцінки похибок при виконанні операцій над наближеними числами:

- при додаванні або відніманні чисел їхні абсолютні похибки додаються;
- при множенні або діленні чисел їхні відносні похибки додаються.

Ці правила можна вивести безпосередньо. Нехай є два наближення a_1 і a_2 до чисел x_2 і x_2, а також відповідні абсолютні похибки Δa_1, Δa_2.

Оцінимо, наприклад, похибку суми

$\Delta(a1 + a2) = |(x1 + x2) - (a1 + a2)\backslash =$

$= |(x1 - a1) + (x2 - a2)| < |x1 - al| + |x2 - a2| < \Delta al + \Delta a2.$

Для визначення оцінок похибки арифметичних дій можна використовувати загальне правило оцінки похибки функції.

Розглянемо функцію $y=f(x)$. Нехай a - наближене значення аргумента x, Δa - його абсолютна похибка. Абсолютну похибку функції можна вважати її приростом, який можна замінити диференціалом $\Delta y \approx dy$.

Тоді одержимо

$\Delta y = |f'(a)|\Delta a$, $\delta y = |f'(a) / f(a)|\Delta a$.

Застосуємо загальне правило, наприклад, для оцінки похибки суми $f(x_1,x_2)= x_1 + x_2$

$\Delta(a_1+a_2) = |f'_{x1}| * \Delta a_1 + |f'_{x2}| * \Delta a_2 = \Delta a_1 + \Delta a_2$

та добутку $f(x_1,x_2)= x_1x_2$

$\Delta(a_1a_2) = |f'_{x1}(a_1,a_2)|\Delta a_1 + |f'(a_1a_2)|\Delta a_2 = |a_2|\Delta a_1 + |a_1|\Delta a_2.$

Тут через $a\ i$ і a_2 позначені значення величин xi і x_2, задані з абсолютними похибками Δa_1 і Δa_2.

Розглянемо віднімання двох майже рівних чисел. Запишемо вираз для відносної похибки різниці у вигляді

$\delta(a_1 — a_2) = \Delta(a_1 — a_2) / |a_1 — a_2| = (\Delta a_1 + \Delta a_2) / |a_1 - a_2|$

При $a_1 \approx a_2$ ця похибка може бути як завгодно великою. Нехай a1=2520, a2=2518. Абсолютні похибки вихідних даних $\Delta a1=\Delta a2=0.5$; відносні похибки - $\delta a1 \approx \delta\ a2 \approx 0.002$ (0.2%). Відносна похибку різниці буде дорівнювати $\delta(a1–a2)=(0.5+0.5)/2=0.5$ (50%). Оскільки в подальших обчисленнях ця велика відносна похибка буде поширюватися, може виявитися сумнівною точність остаточного результату обчислень.

1.3 Підвищення точності результатів обчислень (рекомендації)

Щоб зменшити можливу похибку результату при розв'язуванні задачі, рекомендується дотримуватися таких правил для практичної організації обчислень.

I Похибка суми кількох чисел при розрахунку на ЕОМ зменшиться, якщо починати додавання з менших за величиною доданків.

Якщо додається досить багато чисел, то їх краще розбити на групи з чисел близьких за величиною, провести додавання в групах за вищезгаданою рекомендацією, після чого отримані суми додати, починаючи з меншої.

Якщо задано *n* додатних чисел приблизно однакової величини, то загальна помилка округлення зменшиться, якщо числа додати спочатку групами по п-чисел, а потім додати *n* - часткових сум. При великих *n* верхня межа округлення при такому способі становить всього *1/n* від відповідної межі при довільному додаванні чисел одне до одного.

Причина того, що не виконується комутативний закон додавання, полягає в округленні проміжних результатів, коли багатозначні числа не вміщуються в розрядну сітку ЕОМ. Тому і не все одно, в якому порядку необхідно виконувати арифметичні операції, щоб результат був якомога точнішим.

II Варто уникати віднімання двох майже однакових чисел. Обчислюючи різницю двох чисел, доцільно винести за дужки їхній спільний множник. Для прикладу обчислимо величину

$$P = 6.250001*16 - 25.000003*4 = 1*10^{-5}.$$

Винесемо число “4” за дужки, одержимо точний результат:

$P = 4(6.250001*4 - 25.000003) = 4*10^{-6}$.

Зменшити похибку різниці дозволяють перетворення:

$(a + \varepsilon)^2 - a^2 = \varepsilon\,(2a + \varepsilon)$;

$\sqrt{a + \varepsilon} - \sqrt{a} = \varepsilon(\sqrt{a + \varepsilon} + \sqrt{a})$;

$a - \sqrt{a^2 + \varepsilon} = -\varepsilon\,/(a + \sqrt{a^2 + \varepsilon})$;

$1 - a/(a+\varepsilon) = \varepsilon/(a+\varepsilon)$.

Тут ε- мале в порівнянні з a число.

Тема 2. Метод Гауса, метод Краута, метод прогонки.

Розглянемо систему вигляду

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1, \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2, \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots, \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n = b_n. \end{cases}$$

Її матричний вигляд

$$AX = B. \qquad (3.2)$$

Тут A – $\{[a_{ij}],(i,j = \overrightarrow{1,n})\}$ – матриця системи,

$$B = \begin{bmatrix} b_1 \\ b_2 \\ \\ b_n \end{bmatrix}, \quad X = \begin{bmatrix} x_1 \\ x_2 \\ \\ x_n \end{bmatrix} \text{ – вектори-стовпці.}$$

Відомо, що система (3.1) має єдиний розв'язок, якщо її матриця невироджена (тобто визначник матриці А відмінний від нуля). У випадку виродженості матриці система може мати безліч розв'язків (якщо ранг матриці А і ранг розширеної матриці, отриманої додаванням до А стовпця вільних членів, однакові) або ж не мати розв'язків узагалі (якщо ранги матриці А і розширеної матриці не збігаються).

Методи чисельного розв'язання СЛАР поділяються на точні і наближені. Метод вважають точним, якщо, нехтуючи похибками округлення, він дає точний результат після виконання певної кількості обчислювальних операцій. Математичні пакети прикладних програм для ПЕОМ містять стандартні процедури розв'язання СЛАР такими поширеними точними методами, як метод Гауса, метод Жордана-Гауса, квадратного кореня та інші.

До наближених методів розв'язання СЛАР належать метод простої ітерації, метод Зейделя, метод релаксації та інші. Вони дозволяють отримати послідовність $\{X^к\}$ наближень до розв'язку X^* таку, що $\lim_{k\to\infty} X^k = X^*$.

Ітераційні методи прості, легко програмуються і мають малу похибку округлення, яка не накопичується, але вони дають збіжну послідовність наближень тільки за виконання певної умови, що гарантує виконання принципу стискаючих відображень

(дивись пункти 2.2 -2.5).

Розглянемо більш детально ці дві різні групи підходів до розв'язання СЛАР.

3.1 Метод Гауса

Цей метод базується на приведенні шляхом еквівалентних перетворень вихідної системи (3.1) до вигляду з верхньою трикутною матрицею.

$$\left.\begin{array}{l} c_{11}x_1 + c_{12}x_2 + \ldots + c_{1n}x_n = d_1 \\ 0 + c_{22}x_2 + \ldots + c_{2n}x_n = d_2 \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ 0 + \ldots + 0 + c_{n-1,n-1}x_{n-1} + c_{n-1,n}x_n = d_{n-1} \\ 0 + 0 + \ldots + 0 + c_{nn}x_n = d_n \end{array}\right\}$$

Тоді з останнього рівняння відразу визначаємо $x_n = \frac{d_n}{c_{nn}}$.

Підставляючи його в попереднє рівняння, знаходимо x_{n-1} і т.д. Загальні формули для отримання розв'язку мають вигляд

$$x_k = \frac{1}{c_{kk}}(d_k - \sum_{j=k+1}^{n} c_{ki}x_j), k = n, n-1,...,1.$$

При обчисленнях за формулами (3.4) треба буде виконати приблизно $1/2n^2$ арифметичних дій. Зведення системи (3.1) до вигляду (3.3) можна виконати, послідовно заміняючи рядки матриці системи їх лінійними комбінаціями. Перше рівняння не змінюється. Віднімемо з другого рівняння системи (3.1) перше, помножене на таке число, щоб звернувся в нуль коефіцієнт при x_1. Потім у такий самий спосіб віднімемо перше рівняння з третього, четвертого і т.д. Таким чином обнуляються всі коефіцієнти першого стовпця, що лежать нижче головної діагоналі. Потім за допомогою другого рівняння виключимо з третього, четвертого і т.д. рівнянь коефіцієнти другого стовпця. Послідовно продовжуючи цей процес, виключимо з матриці всі коефіцієнти, що лежать нижче головної діагоналі.

Запишемо загальні формули процесу. Нехай проведене виключення коефіцієнтів з *к-1* стовпця. Тоді залишилися такі рівняння з ненульовими елементами нижче головної діагоналі:

$$\sum_{j=k}^{n} a_{ij}^{(k)} x_j = b_i^{(k)}, k \le i \le n.$$

Помножимо *к-й* рядок на число

$$c_{mk} = \frac{a_{mk}^{(k)}}{a_{kk}^{(k)}}, m > k$$

і віднімемо від *т-го* рядка. Перший ненульовий елемент цього рядка звернеться в нуль, а інші зміняться за формулами

$$a_{ml}^{(k+1)} = a_{ml}^{(k)} - c_{mk}a_{kl}^{(k)},$$

$$b_m^{(k+1)} = b_m^{(k)} - c_{mk}b_k^{(k)}, k < m, l \le n.$$

Виконуючи обчислення при всіх зазначених індексах, виключимо елементи к-го стовпця. Будемо називати таке виключення циклом процесу. Виконання всіх циклів називається прямим ходом виключення.

Після виконання всіх циклів утвориться система, матриця якої має трикутний вигляд . Її легко розв'язати зворотним ходом за формулами (3.4).

Виключення за формулами (3.7) не можна проводити, якщо в ході розрахунків на головній діагоналі виявиться нульовий елемент $a_{kk}^{(k)} = 0$ Але в першому стовпці проміжної

системи (3.5) всі елементи не можуть бути нулями: це означало б, що *detA=0.* Перестановкою рядків можна перемістити ненульовий елемент на головну діагональ і продовжити розрахунки.

Для зменшення обчислювальної похибки можна кожне повторення зовнішнього циклу починати з вибору максимального за модулем елемента в *к-му* стовпці (головного елемента) і перестановки рівняння з головним елементом так, щоб він виявився на головній діагоналі. Цей варіант називається методом Гауса з вибором головного елемента.

Однією з характеристик ефективності того чи іншого алгоритму вважають обчислювальні витрати, що визначаються кількістю елементарних операцій, які необхідно виконати для одержання розв'язку. Для прямого ходу методу Гауса число арифметичних операцій, відповідно до (3.6), (3.7), становить

$$Q_1(n) = \sum_{k=1}^{n-1} \sum_{m=k+1}^{n} \left[\text{ділення} + \sum_{p=k}^{n+1} (\text{множення} + \text{віднімання}) \right] =$$

$$= \frac{1}{3} n(n-1)(2n + \frac{13}{2}) = \frac{2}{3} n^3 + \frac{3}{2} n^2 - \frac{13}{6} n.$$

Для зворотного ходу за формулами число арифметичних операцій дорівнює

$$Q_2(n) = \sum_{k=1}^{n} (\text{ділення} + \text{віднімання} + \sum_{j=k+1}^{n} \text{множення}) =$$

$$= \frac{1}{2} n(n+3) = \frac{1}{2} n^2 + \frac{3}{2} n.$$

Загальні обчислювальні витрати методу Гауса становлять

$Q(n) = \frac{2}{3} n^3 + 2n^2 - \frac{2}{3} n$, тобто $Q(n) \approx \frac{2}{3} n^3 = O(n^3)$.

Метод Краута

Суть методу Краута, або LU-розкладання, полягає в тому, що це своєрідний перезапис методу Гауса. Він дозволяє зробити зручною комп'ютерну реалізацію методу Гауса. Можна явно виділити два етапи, у яких один робить перетворення з матрицею *A* системи, інший - з вектором правих частин b. Отже, нехай дана СЛАР *Ax=b,* наприклад, система розміром 4x4. Запишемо розширену матрицю системи

$$[Ab] = \left[\begin{array}{cccc|c} a_{11} & a_{12} & a_{13} & a_{14} & b_1 \\ a_{22} & a_{23} & a_{24} & a_{25} & b_2 \\ a_{31} & a_{32} & a_{33} & a_{34} & b_3 \\ a_{41} & a_{42} & a_{43} & a_{44} & b_4 \end{array}\right]$$

Тоді, за Гаусом можна явно виділити два етапи (тобто два кроки) - прямий хід (ПХ) і зворотний (ЗХ):

1) ПХ: $a_{k1}^{(i)} = a_{k1}^{(i-1)} - \frac{a_{i1}^{(i-1)}}{a_{i1}^{(i-1)}} - a_{ki}^{(i-1)}$

2) ЗХ: $x_i = b_i^{(i)} - \sum_{j}^{i-1} a_{ij} x_j$

На прямому ході ми робимо так звані "виключення", тобто приводимо матрицю до трикутного вигляду. Тепер легко знайти x_4, а потім і x_3 і т.д. Це був зворотний хід методу Гауса. Всі ці перетворення виконувалися не із самою матрицею, а з розширеною матрицею.

Головна ідея і потреба методу *LU* - декомпозиції полягає в тому, щоб розділити окремо етап перетворення коефіцієнтів матриці і окремо етап перетворення вектора правих частин.

Розглянемо *к* -ий крок методу Гауса, на якому здійснюється занулення піддіагональних елементів *к* –го стовпчика матриці $A^{(к-1)}$. Як було зазначено раніше, з цією метою використовується операція

$$a_{ml}^{(k)} = a_{ml}^{(k-1)} - c_{mk} a_{kl}^{(k-1)},$$

$$c_{mk} = \frac{a_{mk}^{(k-1)}}{a_{kk}^{(k-1)}}, m > k \quad m = \overline{k+1, n}, \; l = \overline{k, n}.$$

У термінах матричних операцій така операція еквівалентна

множенню $A^{(\kappa)} = M_\kappa A^{(\kappa-1)}$, де елементи матриці M_k, визначаються таким чином:

$$m_{ij}^k = \begin{cases} 1,\ i = j; \\ 0,\ i \neq j,\ j \neq k; \\ -c_{k+1,k},\ i \neq j,\ j = k, \end{cases}$$ тобто матриця M_k має вигляд

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & \ldots -c_{k+1,k} & 1 & 0 & 0 \\ .. & .. & .. & .. & .. & .. \\ 0 & 0 & \ldots -c_{n,k} & 0 & 0 & 1 \end{pmatrix}.$$

При цьому вираз для зворотньої операції запишеться у вигляді $A^{(k-1)} = M_k^{-1}A^{(k)}$, *де*

У результаті прямого ходу методу Гауса отримаємо $A^{(n-1)} = U$,

$$A = M_k^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & ..c_{k+1,k} & 1 & 0 & 0 \\ .. & .. & .. & .. & .. & .. \\ 0 & 0 & ..c_{n,k} & 0 & 0 & 1 \end{pmatrix}.$$

$A^{(0)} = M_1^{-1}A^{(1)} = M_k^{-1}M_2^{-1}A^{(2)} = M_k^{-1}M_2^{-1} \ldots M_{n-1}^{-1}A^{(n-1)}$,

де $A^{(n-1)} = U$ – верхня трикутна матриця, а $L = M_k^{-1}M_2^{-1} \ldots M_{n-1}^{-1}$ нижня трикутна матриця, що має вигляд

$$L = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ c_{21} & 1 & 0 & 0 & 0 & 0 \\ c_{31} & c_{32} & 1 & 0 & 0 & 0 \\ .. & .. & ..c_{k+1,k} & 1 & 0 & 0 \\ .. & .. & .. & .. & .. & .. \\ c_{n1} & c_{n2} & ..c_{nk} & c_{n,k+1} & ..c_{n,n-1} & 1 \end{pmatrix}.$$

У подальшому *LU*- розкладання може бути ефективно використано для розв'язання систем лінійних алгебраїчних рівнянь. Це дозволяє один раз перетворити матрицю системи, а потім неодноразово розв'язувати декілька систем з різними правими частинами. Обчислювальні витрати при цьому будуть зводитися тільки до зворотного ходу.

Запишемо A x= b, як

$L \cdot U \cdot x = b$.

Позначимо

$U \cdot x$ = y.

I , отже ,

$$L \cdot y = b.$$

Таким чином, прямий хід методу LU-декомпозиції складається з розкладу матриці A на нижню L та верхню U трикутні матриці - це прямий хід.

Потім визначається вектор y на основі співвідношень:

$y_1 = \frac{b_1}{l_{11}}$, $y_1 = \frac{b}{l_{11}}\left(b_i - \sum_{j=1}^{i-1} l_{ij}\, y_j\right)$.

На зворотному ході методу LU – декомпозиції розв'язується рівняння $U \cdot x$ = y.

З урахуванням того, що U – трикутна матриця,

$$x_N = \frac{y_N}{u_{NN}} \quad , x_i = \frac{1}{u_{ii}}\left(y_i - \sum_{j=i+1}^{N} u_{ij}x_j\right).$$

Отже LU-розкладання є просто свого роду іншою формою запису еквівалентних перетворень матриці за методом Гауса, але проведених з урахуванням умови $A = L \cdot U$.

Приклад. Розв'яжемо СЛАР за схемою LU-розкладання:

$$\begin{cases} x_1 - 2x_2 + 3x_3 = 1 \\ 2x_1 + 3x_2 - x_3 = 2 \\ -x_1 - x_2 + x_3 = 3 \end{cases}$$

Виконаємо дії за алгоритмом і отримаємо матриці *L та U у вигляді:*

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -1 & -3/7 & 1 \end{bmatrix}, \; U = \begin{bmatrix} 1 & -2 & 3 \\ 0 & 7 & -7 \\ 0 & 0 & 1 \end{bmatrix}.$$

Спочатку знаходимо розв'язок системи $Lg = b$

$$\begin{cases} g_1 = 1 \\ 2g_1 + g_2 = 2 \\ -g_1 - 3/7 g_2 + g_3 = 3 \end{cases}$$

Отримаємо: $g = \{1,0,4\}$

Тепер реалізуємо зворотний хід методу Гауса, розв'язуючи систему $Ux = g$:

$$\begin{cases} x_1 - 2x_2 + 3x_3 = 1 \\ 7x_2 - 7x_3 = 0 \\ x_3 = 4 \end{cases}$$

Отже, остаточна відповідь: $x_1 = 4$, $x_2 = 4$, $x_3 = -3$.

Ітераційні методи розв'язування СЛАР. Методи простих ітерацій.

При великій кількості рівнянь прямі методи розв'язання СЛАР (за винятком методу прогонки) стають важко реалізованими на ЕОМ насамперед через складність зберігання й обробки матриць великої розмірності. У той же час характерною рисою багатьох СЛАР, що виникають у прикладних задачах є розрідженість матриць. Число ненульових елементів таких матриць є малим у порівнянні з їхньою розмірністю. Для розв'язання СЛАР з розрідженими матрицями краще використати ітераційні методи.

Методи послідовних наближень, у яких при обчисленні наступного наближення розв'язку використовуються попередні, уже відомі наближення розв'язку, називаються ітераційними

(дивись 2.4).

Розглянемо СЛАР (3.1) з невиродженою матрицею (det A≠ 0). Розв'яжемо систему (3.1) щодо невідомих при ненульових діагональних елементах *aii≠ 0, i= 1…n* (якщо який-небудь коефіцієнт на головній діагоналі дорівнює нулю, досить відповідне рівняння поміняти місцями з будь-яким іншим рівнянням). Одержимо систему у вигляді

$$\begin{cases} x_1 = \beta_1 + \alpha_{11}x_1 + \ldots + \alpha_{1n}x_n \\ x_2 = \beta_2 + \alpha_{21}x_1 + \ldots + \alpha_{2n}x_n \\ \ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots \\ x_n = \beta_n + \alpha_{n1}x_1 + \ldots + \alpha_{nn}x_n \end{cases}$$

або у векторно-матричній формі $X = \beta + \alpha X$.

$$X = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}, \beta = \begin{pmatrix} \beta_1 \\ \vdots \\ \beta_n \end{pmatrix}, \alpha = \begin{pmatrix} \alpha_{11} & \cdots & \alpha_{n1} \\ \vdots & \cdots & \vdots \\ \alpha_{n1} & \cdots & \alpha_{nn} \end{pmatrix}.$$

Вирази для компонентів вектора β та матриці α еквівалентної системи:

$$\beta_i = \frac{b_i}{a_{ii}}; \alpha_{ij} = -\frac{a_{ij}}{a_{ii}}, i, j = 1...n, i \neq j; \alpha_{ij} = 0, i = j, i = 1...n.$$

При такому способі приведення вихідної СЛАР до еквівалентного вигляду метод простих ітерацій ще називають методом Якобі. За нульове наближення *X*(0) вектора невідомих візьмемо вектор правих частин *X*(0)=β або *(x1 (0),x2 (0),…,xn (0))**=(β1,β2,…,βn)*.

Тоді метод простих ітерацій набере вигляду.

$$\begin{cases} X^{(0)} = \beta \\ X^{(1)} = \beta + \alpha X^{(0)} \\ X^{(2)} = \beta + \alpha X^{(1)} \\ \cdots\cdots\cdots\cdots \\ X^{(k)} = \beta + \alpha X^{(k-1)} \end{cases}$$

Бачимо перевагу ітераційних методів у порівнянні, наприклад, з розглянутим вище методом Гауса. В обчислювальному процесі беруть участь тільки добутки матриці на вектор, що дозволяє працювати тільки з ненульовими елементами матриці, значно спрощуючи процес зберігання й обробки матриць. При цьому не відбувається накопичення похибки заокруглення.

Визначення збіжності ітераційного процесу можна знайти в 2.3-2.5. З огляду на сформульовані там теореми, має місце достатня умова збіжності методу простих ітерацій для СЛАР.

Метод Зейделя розв'язання СЛАР

Метод простої ітерації досить повільно збігається. Для його прискорення існує метод Зейделя. Суть його в тому, що при обчисленні компонентів *xi(к+1)* вектора невідомих на *(k+1)*-ій ітерації використовуються *x1(к+1), x2 (к+1),...,xi-1(к+1)*, уже обчислені

на *(k+1)*-ій ітерації. Значення інших компонентів беруться з попередньої ітерації. Так само, як і у методі простих ітерацій, будується еквівалентна СЛАР (3.26) і за початкове наближення береться вектор правих частин *X0=(β1,β2,..., βn)**. Тоді метод Зейделя для пошуку наближення *X(к+1)* має вигляд

$$\begin{cases} x_1^{k+1} = \beta_1 + \alpha_{11}x_1^k + \alpha_{12}x_2^k + \ldots + \alpha_{1n}x_n^k \\ x_2^{k+1} = \beta_2 + \alpha_{21}x_1^{k+1} + \alpha_{22}x_2^k + \ldots + \alpha_{2n}x_n^k \\ x_3^{k+1} = \beta_3 + \alpha_{31}x_1^{k+1} + \alpha_{32}x_2^{k+1} + \alpha_{33}x_3^k + \ldots + \alpha_{3n}x_n^k \\ \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\ x_n^{k+1} = \beta_n + \alpha_{n1}x_1^{k+1} + \alpha_{n2}x_2^{k+1} + \ldots + \alpha_{nn-1}x_{n-1}^{k+1} + \alpha_{nn}x_n^k \end{cases}$$

Із цієї системи бачимо, що *Xk+1*=β+*BXk+1* +*CXk* , де *B* – нижня трикутна матриця з діагональними елементами, що дорівнюють нулю, а C - верхня трикутна матриця з діагональними елементами, відмінними від нуля, *α=B+C*.

Таким чином, метод Зейделя є методом простих ітерацій з матрицею правих частин □*=(E-B)-1C* і вектором правих частин *(E-B)-1β,* й, отже, збіжність і похибку методу Зейделя можна досліджувати за допомогою формул, виведених для методу простих ітерацій, у яких замість

матриці підставлена матриця *(E-B)-1C*, а замість вектора правих частин - вектор *(E-B)-1β*. Для практичних обчислень важливо, що як достатні умови збіжності методу Зейделя можуть бути використані умови, наведені вище для методу простих ітерацій (1 або, якщо

використовується еквівалентна СЛАР у формі (3.1), -діагональна перевага матриці *A*). У випадку виконання цих умов для оцінки похибки на k -ій ітерації можна використати вираз

Відзначимо, що, як і метод простих ітерацій, метод Зейделя може збігатися й при порушенні умови 1.

Приклад. Методом Зейделя розв'язати СЛАР із попереднього прикладу.

Розв'язання. Діагональна перевага елементів вихідної матриці СЛАР гарантує збіжність методу Зейделя. Ітераційний процес будуємо в такий спосіб:

$$x^{(0)} = (1{,}2 \quad 1{,}3 \quad 1{,}4)^T$$

$$\begin{cases} x_1^{(1)} = 1{,}2 - 0{,}1*1{,}3 - 0{,}1*1{,}4 = 0{,}93 \\ x_2^{(1)} = 1{,}3 - 0{,}2*0{,}93 - 0{,}1*1{,}4 = 0{,}974 \\ x_3^{(1)} = 1{,}4 - 0{,}2*0{,}93 - 0{,}2*0{,}974 = 1{,}0192 \end{cases}$$

$$\begin{cases} x_1^{(2)} = 1{,}2 - 0{,}1*0{,}974 - 0{,}1*1{,}0192 = 1{,}0007 \\ x_2^{(2)} = 1{,}3 - 0{,}2*1{,}0007 - 0{,}1*1{,}0192 = 0{,}998 \\ x_3^{(2)} = 1{,}4 - 0{,}2*1{,}0007 - 0{,}2*0{,}998 = 1{,}0003 . \end{cases}$$

Таким чином, уже на другій ітерації похибка $\left\|x^{(2)} - x^{(*)}\right\| < 10^{-2} = \varepsilon$, тобто метод Зейделя в цьому випадку збігається швидше ніж метод простих ітерацій.

Приклад . Розв'язання СЛАР **Ax=b**, отримане за допомогою вбудованої функції *lsolve* (пакет Mathcad).

Перевірка достатньої умови збіжності методу Зейделя

$$\text{normB}(B,n) := \left| \begin{array}{l} \text{for } i \in 1..n \\ \quad s_i \leftarrow \sum_{j=1}^{n} |B_{i,j}| \\ \max(s) \end{array} \right.$$

$$\text{normB}(B,4) = 0.8$$

Достатня умова виконана.

$$A := \begin{pmatrix} 15 & 3 & 4 & 5 \\ 2 & 16 & 4 & 5 \\ 2 & 3 & 17 & 5 \\ 2 & 3 & 4 & 18 \end{pmatrix} \qquad b := \begin{pmatrix} 13 \\ -1 \\ 17 \\ -50 \end{pmatrix}$$

Перетворення системи $Ax=b$ до вигляду $x=Bx+c$, зручного для ітерацій.

x := lsolve(A, b)

$$PB(A,n) := \left| \begin{array}{l} \text{for } i \in 1..n \\ \quad \text{for } j \in 1..n \\ \quad\quad B_{i,j} \leftarrow 0 \text{ if } i = j \\ \quad\quad B_{i,j} \leftarrow \frac{-A_{i,j}}{A_{i,i}} \text{ if } i \neq j \\ B \end{array} \right.$$

$$Pc(A,b,n) := \left| \begin{array}{l} \text{for } i \in 1..n \\ \quad c_i \leftarrow \frac{b_i}{A_{i,i}} \\ c \end{array} \right. \qquad c := Pc(A,b,4)$$

ORIGIN := 1 - нумерація масивів починається з одиниці.

$$B := PB(A,4) \qquad B = \begin{bmatrix} 0 & -0.2 & -0.2666666667 & -0.3333333333 \\ -0.125 & 0 & -0.25 & -0.3125 \\ -0.1176470588 & -0.1764705882 & 0 & -0.2941176471 \\ -0.1111111111 & -0.1666666667 & -0.2222222222 & 0 \end{bmatrix}$$

Змістовий модуль 6. Чисельні методи розв'язання нелінійних рівнянь.

Тема 1. Метод простих ітерацій. Ітераційний метод Ньютона, модифікаційний метод Ньютона.

Метод простих ітерацій

Припустимо, що рівняння *f(x)=0* за допомогою деяких тотожних перетворень зведене до вигляду $x = \varphi(x)$. Відмітимо, що таке перетворення можна робити різними способами, і при цьому матимемо різні функції $\varphi(x)$ в правій частині рівняння. Рівняння *f(x)=0* еквівалентне рівнянню $x = x + \lambda(x) f(x)$ для будь-якої функції $\lambda(x) \neq 0$. Таким чином, можна взяти $\varphi(x) = x + \lambda(x) f(x)$ і при цьому вибрати функцію (або постійну) $\lambda \neq 0$ так, щоб функція $\varphi(x)$ задовольняла тим властивостям, які знадобляться нам для забезпечення знаходження кореня рівняння.

Для знаходження кореня рівняння $x = \varphi(x)$ виберемо деяке початкове наближення x_0 (розташоване, по можливості, близько до кореня). Далі будемо обчислювати подальші наближення $x_1, x_2, .., x_i, x_{i+1}, ...$ за формулами $x_1 = \varphi(x_0), x_2 = \varphi(x_1),$ і так далі, тобто використовуючи кожне обчислене наближення до кореня як аргумент функції $\varphi(x)$ в черговому обчисленні. Такі обчислення за однією і тією ж формулою $x_{i+1} = \varphi(x_i)$, коли отримане на попередньому кроці значення використовується на подальшому кроці, називаються ітераціями. Ітераціями називають часто і самі значення x_i, отримані в цьому процесі (тобто, в нашому випадку, послідовні наближення до кореня). Відмітимо той факт, що x^* – корінь рівняння $x = \varphi(x)$, означає, що x^* є абсциса точки перетину графіка $y = \varphi(x)$ з прямою $y=x$. Якщо ж при якому-небудь x_0 обчислено значення $x_1 = \varphi(x)$ і взято за новий аргумент функції, то це означає, що через точку графіка $(x_0, \varphi(x_0))$ проводиться горизонталь до прямої $y=x$, а звідти опускається перпендикуляр на вісь. Там і знаходитиметься новий аргумент x_1.

Прослідкуємо, як змінюються послідовні наближення x_i при різних варіантах взаємного розташування графіка $y = \varphi(x)$ і прямої $y=x$.

1) Графік $y = \varphi(x)$ розташований, принаймні в деякому околі кореня , що включає початкове наближення x_0, в деякому куті зі сторонами, що мають нахил менше $\frac{\pi}{4}$ до горизонталі (тобто сторони кута – прямі $y = f(x^*) \pm k(x - x^*)$, де $0<k<1$):

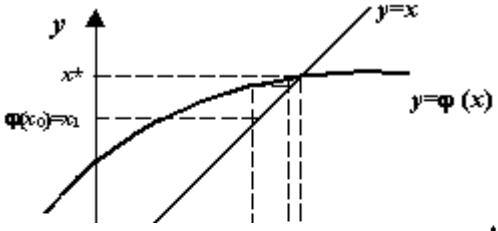
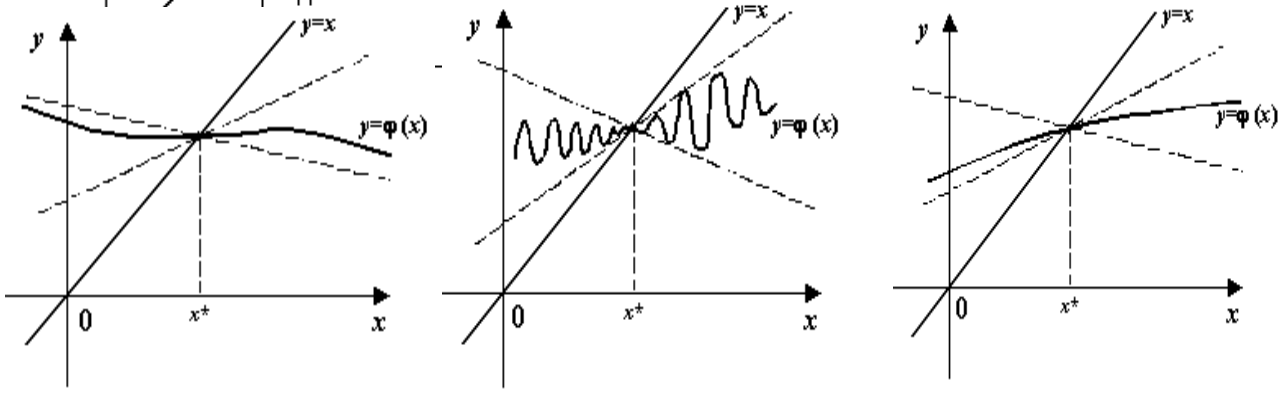


Рис.2.Графік перетинає пряму $y=x$ під малим кутом:



варіанти розташування.

Якщо припустити додатково, що функція $\varphi(x)$ має похідну $\varphi'(x)$, то цей випадок відповідає тому, що виконується нерівність $|\varphi'(x)| < 1$, при x, близьких до кореня x^*. Простежимо в цьому випадку за поведінкою послідовних наближень $x_0, x_1, ...$

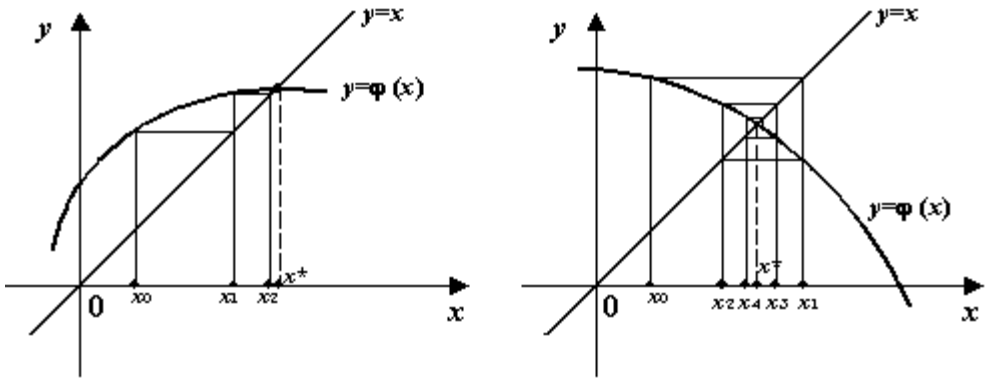


Рис.3.Наближення, що збігаються до кореня у випадку $|\varphi'(x)| < 1$.

Ми бачимо, що кожне наступне наближення x_{i+1} буде в цьому випадку розташовано ближче до кореня x^*, ніж попереднє x_i. При цьому, якщо графік при $x<x^*$, лежить нижче за горизонталь $y = \varphi(x^*)$, а при $x>x^*$– вище за неї (що, у разі наявності похідної, вірно, якщо $0 < \varphi'(x) < 1$), то наближення x_i поводяться монотонно: якщо $x_o<x^*$, то послідовність $\{x_i\}$ монотонно зростає і прямує до x^*, а якщо $x_o>x^*$, то монотонно спадає і також прямує до x^*. Якщо ж графік функції $\varphi(x)$ лежить вище за горизонталь $y = \varphi(x^*)$ при $x<x^*$ і нижче за неї при $x>x^*$ (якщо $-1 < \varphi'(x) < 0$), то послідовні наближення поводяться інакше: вони "скачуть" навколо кореня, з кожним стрибком наближаючись до нього, але так само прямують до x^* при $i \to \infty$.

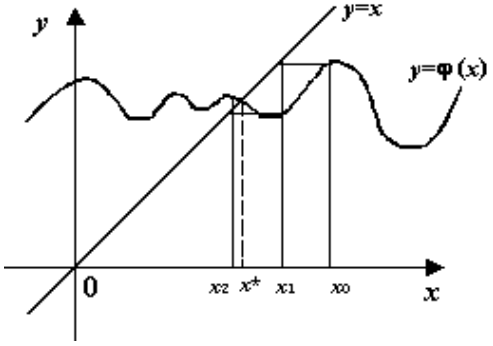


Рис.4.У випадку немонотонної функції ітерації, що сходяться, можуть поводитися нерегулярно

Відмітимо, що якщо функція $\varphi(x)$ не монотонна в околі точки x^*, то послідовні наближення можуть поводитися нерегулярно (тобто не монотонно і не потрапляючи почергово то лівіше, то правіше кореня, а роблячи стрибки відносно кореня при довільних номерах.

2) Графік $y = \varphi(x)$ розташований, принаймні в деякому околі кореня, що включає початкове наближення x_0, в деякому куті зі сторонами, що мають нахил більше $\frac{\pi}{4}$ до горизонталі (тобто сторони кута – прямі $y = f(x^*) \pm k(x - x^*)$, де $k>1$):

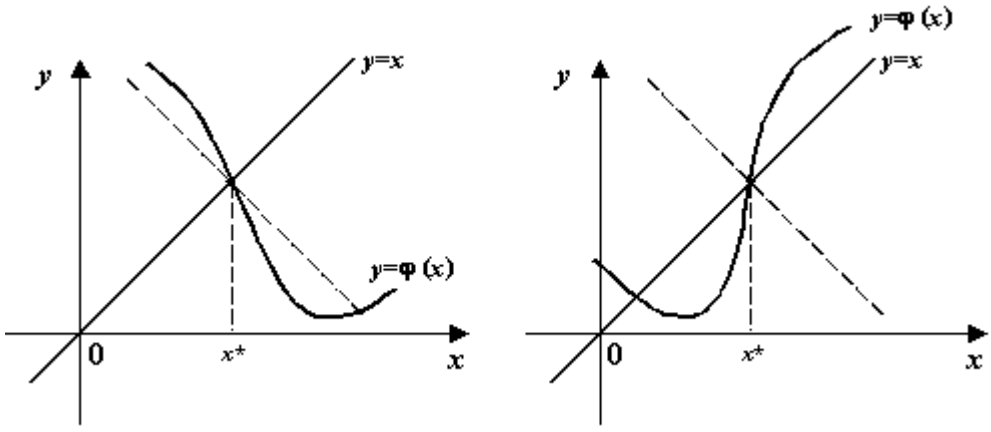


Рис.5.Графік перетинає пряму $y=x$ під великим кутом: варіанти розташування.

Якщо функція $\varphi(x)$ має похідну $\varphi'(x)$, то при x, близьких до кореня x^* виконується нерівність $|\varphi'(x)| > 1$.

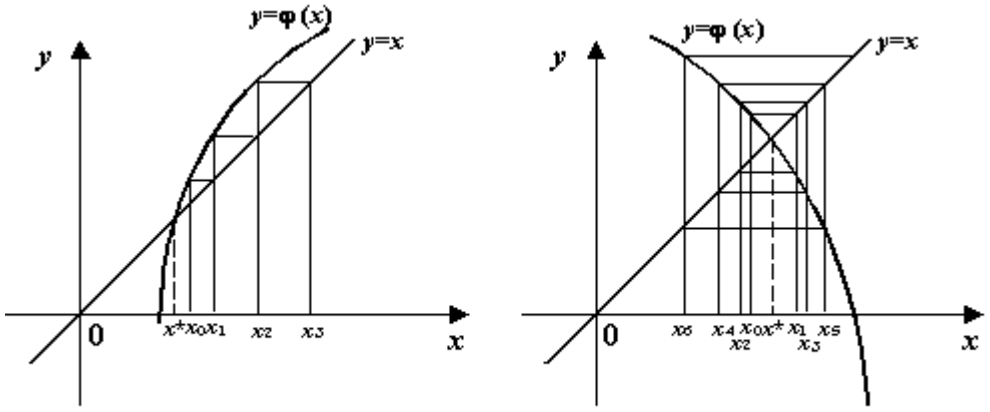
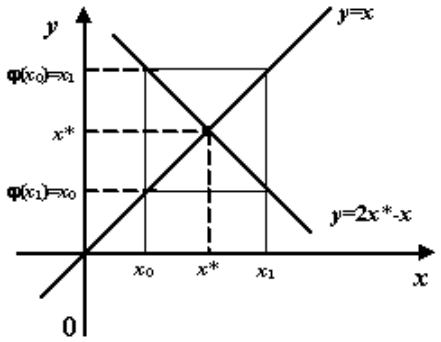


Рис.6.Послідовність $x_0, x_1, x_2, \ldots$ розбіжна у випадку $|\varphi'(x)| > 1$.

Кожна наступна ітерація x_{i+1} буде в цьому випадку розташована далі від кореня x^*, ніж попередня x_i. При цьому, залежно від того, чи перетинає графік пряму $y=x$ "знизу вгору" або "згори донизу", послідовність $\{x_i\}$ монотонно віддаляється від кореня x^* або ж ітерації віддаляються від x^*, потрапляючи почергово то справа, то зліва від кореня.

Ще одне зауваження: якщо не виконується ні умова $|\varphi'(x)| < 1$, ні $|\varphi'(x)| > 1$, то ітерації $x_1, x_2, .., x_i, x_{i+1}, \ldots$ можуть зациклюватися. Напри-лад, якщо рівняння має вигляд: x=2x*-x:



Розв'язування нелінійних рівнянь

3.4.1. Постановка задачі:

Розглянемо задачу знаходження коренів рівняння

$$f(x) = 0, \qquad (1)$$

де $f(x)$ – задана функція дійсного змінного.

Розв'язування даної задачі можна розкласти на декілька етапів:

а) досліджена розташування коренів (в загальному випадку на комплексній площині) та їх кратність;

б) відділення коренів, тобто виділення областей, що містять тільки один корінь;

в) обчислення кореня з заданою точністю за допомогою одного з ітераційних алгоритмів.

Далі розглядаються ітераційні процеси, що дають можливість побудувати числову послідовність x_n, яка збігається до шуканого кореня x_* рівняння (1).

3.5. 1. Метод ділення проміжку навпіл (метод дихотомії)

Нехай $f \in C[a,b]$, $f(a)f(b) < 0$ і відомо, що рівняння (1) має єдиний корінь $x_* \in [a,b]$. Покладемо $a_0=a$, $b_0=b$, $x_0=(a_0+b_0)/2$. Якщо $f(x_0) = 0$, то $x_* = x_0$. Якщо $f(x_0) \neq 0$, то покладемо

$$a_{n+1}=\begin{cases} x_n, \text{ якщо } \operatorname{sign} f(a_n)=\operatorname{sign} f(x_n), \\ a_n, \text{ якщо } \operatorname{sign} f(a_n)\neq\operatorname{sign} f(x_n), \end{cases} \quad (2)$$

$$b_{n+1}=\begin{cases} x_n, \text{ якщо } \operatorname{sign} f(b_n)=\operatorname{sign} f(x_n), \\ b_n, \text{ якщо } \operatorname{sign} f(b_n)\neq\operatorname{sign} f(x_n), \end{cases} \quad (3)$$

$$x_{n+1}=\frac{a_{k+1}+b_{k+1}}{2}, \quad n=0,1,2,\ldots, \quad (4)$$

і обчислимо $f(x_{n+1})$. Якщо $f(x_{n+1})=0$, то ітераційний процес зупинимо і будемо вважати, що $x_*\approx x_{n+1}$. Якщо $f(x_{n+1})\neq 0$, то повторюємо розрахунки за формулами (2)-(4).

З формул (2), (3) видно, що $\operatorname{sign} f(a_{n+1})=\operatorname{sign} f(a_n)$ і $\operatorname{sign} f(b_{n+1})=\operatorname{sign} f(b_n)$. Тому $f(a_{n+1})f(b_{n+1})<0$, а отже шуканий корінь x_* знаходиться на проміжку $[a_{n+1},b_{n+1}]$. При цьому має місце оцінка збіжності

$$|x_n-x_*|\le\frac{b-a}{2^{n+1}}. \quad (5)$$

Звідси випливає, що кількість ітерацій. які необхідно провести для знаходження наближеного кореня рівняння (1) з заданою точністю ε задовольняє співвідношенню

$$n\ge\left[\log_2\frac{b-a}{\varepsilon}\right]. \quad (6)$$

де [*c*] – ціла частина числа *c*.

Серед переваг даного методу слід відзначити простоту реалізації та надійність. Послідовність $\{x_n\}$ збігається до кореня x_* для довільних неперервних функцій *f*(*x*). До недоліків можна віднести невисоку швидкість збіжності методу та неможливість безпосереднього узагальнення систем нелінійних рівнянь.

3.6. 2. Метод простої ітерації

Метод простої ітерації застосовується до розв'язування нелінійного рівняння виду

$$x=\varphi(x). \quad (7)$$

Перейти від рівняння (1) до рівняння(7) можна багатьма способами, наприклад, вибравши

$$\varphi(x)=x+\psi(x)f(x), \quad (8)$$

де $\psi(x)$ – довільна знакостала неперервна функція.

Вибравши нульове наближення x_0, наступні наближення знаходяться за формулою

$$x_{n+1}=\varphi(x_n), \quad n=0,1,2,\ldots. \quad (9)$$

Наведемо достатні умови збіжності методу простої ітерації.

Теорема 1. *Нехай для вибраного початкового наближення x_0 на проміжку*

$$S=\{x:|x-x_0|\le\delta\} \quad (10)$$

функція φ(*x*) *задовольняє умові Ліпшиця*

$$|\varphi(x')-(x'')|\le q|x'-x''|, \quad x',x''\in S \quad (11)$$

де 0<*q*<1, *і виконується нерівність*

$$|\varphi(x_0)-x_0|\le(1-q)\delta. \quad (12)$$

Тоді рівняння (7) має на проміжку S єдиний корінь x_*, до якого збігається послідовність (9), причому швидкість збіжності визначається нерівністю

$$|x_n-x_*|\le\frac{q^n}{1-q}|\varphi(x_0)-x_0|. \quad (13)$$

Зауваження: якщо функція φ(*x*) має на проміжку *S* неперервну похідну $\varphi'(x)$, яка задовольняє умові

$$|\varphi'(x)| \le q < 1, \qquad (14)$$

то функція φ(*x*) буде задовольняти умові (11) теореми 1.

З (13) можна отримати оцінку кількості ітерацій. які потрібно провести для знаходження розв'язку задачі (7) з наперед заданою точністю ε:

$$n \ge \left[\frac{\ln \dfrac{|\varphi(x_0) - x_0|}{(1-q)\cdot\varepsilon}}{\ln(1/q)} \right] + 1. \qquad (15)$$

Наведемо ще одну оцінку. що характеризує збіжність методу простої ітерації:

$$|x_n - x_*| \le \frac{q}{1-q}|x_n - x_{n-1}|. \qquad (16)$$

3.7. 3. Метод релаксації

Для збіжності ітераційного процесу (9) суттєве значення має вибір функції φ(*x*). Зокрема, якщо в (8) вибрати $\psi(x) = \tau = \text{const}$, то отримаємо метод релаксації.

$$x_{n+1} = x_n + \tau f(x_n), \quad n = 0,1,2,\ldots, \qquad (17)$$

який збігається при

$$-2 < \tau f'(x) < 0. \qquad (18)$$

Якщо в деякому околі кореня виконуються умови

$$f'(x) < 0, \quad 0 < m_1 < |f'(x)| < M_1, \qquad (19)$$

то метод релаксації збігаються при $\tau \in (0, 2/M_1)$. Збіжність буде найкращою при

$$\tau = \tau_{\text{опт}} = 2/(m_1 + M_1). \qquad (20)$$

При такому виборі τ для похибки $z_n = x_n - x_*$ буде мати місце оцінка

$$|z_n| \le q^n |z_0|, \quad n = 0,1,2,\ldots, \qquad (21)$$

де $q = (M_1 - m_1)/(M_1 + m_1)$.

Кількість ітерацій, які потрібно провести для знаходження розв'язку з точністю ε визначається нерівністю

$$n \ge \left[\frac{\ln(|z_0|/\varepsilon)}{\ln(1/q)} \right] + 1. \qquad (22)$$

Зауваження: якщо виконується умова $f'(x) > 0$, то ітераційний метод (17) потрібно записати у вигляді

$$x_{n+1} = x_n - \tau f(x_n).$$

3.8. 4. Метод Ньютона

Метод Ньютона застосовується до розв'язування задачі (1), де *f*(*x*) є неперервно-диференційованою функцією. На початку обчислень вибирається початкове наближення x_0. Наступні наближення обчислюються за формулою

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}, \quad n = 0,1,2,\ldots, \quad f'(x_n) \ne 0. \qquad (23)$$

З геометричної точки зору x_{n+1} є значенням абсциси точки перетину дотичної до кривої $y=f(x)$ в точці $(x_n, f(x_n))$ з віссю абсцис. Тому метод Ньютона називають також методом дотичних.

Теорема 2. *Якщо* $f(x)\in C^2[a,b]$, $f(a)f(b)<0$, а $f''(x)$ *не змінює знака на* [a,b], *то виходячи з початкового наближення* $x_0\in[a,b]$, *що задовольняє умові* $f(x_0)f''(x_0)>0$, *можна обчислити методом Ньютона єдиний корінь* x_* *рівняння (1) з будь-якою степінню точності.*

Теорема 3. *Нехай* x_* *– простий дійсний корінь рівняння (1) і* $f(x)\in C^2(S)$, *де* $S=\{x:|x-x_*|\le\delta\}$,

$$0<m_1=\min_{x\in S}|f'(x)|,\quad M_2=\max_{x\in S}|f''(x)|, \tag{24}$$

причому

$$q=\frac{M_2|x_0-x_*|}{2m_1}<1. \tag{25}$$

Тоді для $x_0\in S$ *метод Ньютона збігається, причому для похибки справедлива оцінка*

$$|x_n-x_*|\le q^{2^n-1}|x_0-x_*|. \tag{26}$$

З оцінки (26) видно, що метод Ньютона має квадратичну збіжність, тобто похибка на (n+1)-й ітерації пропорційна квадрату похибки на n-й ітерації.

Модифікований метод Ньютона

$$x_{n+1}=x_n-\frac{f(x_n)}{f'(x_0)},\quad n=0,1,2,\ldots \tag{27}$$

дозволяє не обчислювати похідну $f'(x)$ на кожній ітерації, а отже і позбутися можливого ділення на нуль. Однак цей алгоритм має тільки лінійну збіжність.

Кількість ітерацій, які потрібно провести для знаходження розв'язку задачі (1) з точністю ε задовольняє нерівності

$$n\ge\left[\log_2\left(\frac{\ln(|x_0-x_*|/\varepsilon)}{\ln(1/q)}\right)+1\right]+1. \tag{28}$$

Приклад 1. *Розв'язати рівняння*

$$x+\sin x-1=0 \tag{29}$$

методом ділення проміжку навпіл з точністю $\varepsilon=10^{-4}$.

Розв'язання. Спочатку знайдемо проміжок, де рівняння має єдиний корінь. Оскільки похідна функції $f(x)=x+\sin x-1$ не змінює знак, то корінь у рівнянні (29) буде один. Легко бачити, що $f(0)=-1<0$, а $f\left(\frac{\pi}{2}\right)=\frac{\pi}{2}>0$. Отже корінь належить проміжку $\left[0,\frac{\pi}{2}\right]$. Виберемо $a_0=0,\quad b_0=\frac{\pi}{2}$. Згідно з формулою (6), отримаємо, що для знаходження кореня з точністю 10^{-4} необхідно провести 13 інтеграцій. Відповідні значення x_n наведені в табл. 1.

Табл.1

n	x_n	$f(x_n)$
0	0785398E+00	0492505E+00
1	0392699E+00	0224617E+00
2	0589049E+00	0144619E+00
3	0490874E+00	0377294E-01
4	0539961E+00	0540639E-01
5	0515418E+00	0831580E-02

6	0503146E+00	0146705E-01
7	0509282E+00	0316819E-02
8	0512350E+00	0257611E-02
9	0510816E+00	0295467E-03
10	0511583E+00	0114046E-02
11	0511199E+00	0422535E-03
12	0511007E+00	0635430E-04
13	0510911E+00	0116016E-03

Приклад 2. *Знайти додатні корені рівняння*

$x^3-x-1=0$ (30)

методом простої ітерації з точністю $\varepsilon=10^{-4}$.

Розв'язання. Графічне дослідження рівняння (30) показує, що існує єдиний дійсний додатній корінь цього рівняння і він належить проміжку [1,2]. Оскільки на цьому проміжку $x \neq 0$, то рівняння (30) можна подати у вигляді

$$x = \sqrt{\frac{1}{x}+1}. \qquad (31)$$

Позначимо $\varphi(x) = \left(\frac{1}{x}+1\right)^{1/2}$. Перевіримо виконання умов теореми про збіжність методу простої ітерації. Виберемо $x_0=1{,}5$, тоді $\delta=0{,}5$. Розглянемо

$$\varphi'(x) = -\frac{1}{2\sqrt{x^3+x^4}}; \quad \max_{1\le x\le 2}|\varphi'(x)| = \frac{1}{2\sqrt{2}},$$

тобто $q = \frac{1}{2\sqrt{2}}$.

тоді $|\varphi(x_0)-x_0| = \left|\sqrt{\frac{2}{3}+1}-1{,}5\right| = 0{,}205, \quad (1-q)\delta = 0{,}5\cdot\left(1-\frac{1}{2\sqrt{2}}\right) \approx 0{,}3232,$

а отже умова (12) виконується. З формули (15) маємо, що кількість ітерацій, які необхідно провести для знаходження кореня з точністю $\varepsilon=10^{-4}$ повинна задовольняти умові $n \ge 8$. Відповідні значення x_n та $x_n-\varphi(x_n)$ наведені в табл.2.

Табл.2

n	x_n	$x_n-\varphi(x_n)$
0	0150000E+01	0209006E+00
1	0129099E+01	0411454E-01
2	0133214E+01	0901020E-02
3	0132313E+01	0193024E-02
4	0132506E+01	0415444E-03
5	0132464E+01	0892878E-04
6	0132473E+01	0191927E-04
7	0132471E+01	0417233E-05
8	0132472E+01	0953674E-06

Виходячи з нерівності (16) і отриманих результатів видно, що для досягнення заданої точності достатньо було провести 5 ітерацій (n=5). Взагалі слід відзначити, що апостеріорна оцінка (16) є більш точною і її використання може заощадити деяку кількість обчислень.

Приклад 3. *Методом релаксації знайти найменший за модулем від'ємний корінь рівняння*

$x^3-3x^2-1=0$ (32)

з точністю $\varepsilon=10^{-4}$.

Розв'язання. Спочатку виділимо корені рівняння (32) користуючись наступною таблицею

Табл.3

x	-4	-3	-2	-1	0	1	2	3
sign$f(x)$	–	–	+	+	–	+	+	+

З даної таблиці видно, що рівняння має три корені розташовані на проміжках [–3;–2], [–1;0], [0;1]. Будемо знаходити корінь на проміжку [–1;0]. Обчисливши значення $f(-0,5)=-0,375$ можна уточнити проміжок існування кореня [–1;–0,5].

Позначимо $f(x)=x^3-3x^2-1$. Тоді $f'(x)=3x^2+6x<0, \quad x\in[-1;-0,5]$ і є монотонно зростаючою функцією на [–1;–0,5] (оскільки $f''(x)=6x+6\geq 0$).

Тому $$m_1=\min_{x\in[-1;-0,5]}|f'(x)|=|f'(-0,5)|=2,25,$$

$$M_1=\max_{x\in[-1;-0,5]}|f'(x)|=|f'(-1)|=3.$$

Тоді, відповідно до формул (20) і (21), будемо мати вигляд

$$x_{n+1}=x_n+\tau_{\text{опт}}(x_n^3+3x_n^2-1). \qquad (33)$$

Вибравши за початкове наближення точку $x_0=-0,5$ будемо мати оцінку $|z_0|\leq 0,5$, а кількість ітерацій, які потрібно провести для знаходження розв'язку з точністю $\varepsilon=10^{-4}$ буде дорівнювати 5 (див. (22)). В табл. 4 наведені відповідні дані ітераційної послідовності:

Табл.4

n	x_n	$f(x_n)$
0	0500000E+00	0142857E+00
1	0642857E+00	0985700E-02
2	0652714E+00	0105500E-04
3	0652704E+00	0596046E-07
4	0652704E+00	0000000E+00
5	0652704E+00	0000000E+00

Із наведених даних видно, що необхідна точність досягається раніше 5-ї ітерації. Це досить характерно для апріорних оцінок типу (22).

Приклад 4. *Методом Ньютона знайти найменший додатній корінь рівняння*

$x^3+3x^2-1=0$ (34)

з точністю $\varepsilon=10^{-4}$.

Розв'язання. З табл. 3 видно, що рівняння (34) має єдиний додатній корінь, що належить проміжку [0;1]. обчислимо $f(0,5)=-0,125$. Тепер будемо шукати корінь на проміжку [0,5;1]. Нехай $f(x)=x^3+3x^2-1$. Тоді $f'(x)=3x^2+6x>0, \quad f''(x)=6x+6>0, \quad x\in[0,5;1]$.

$$m_1=\min_{x\in[0,5;1]}|f'(x)|=|f'(0,5)|=3,75,$$

$$M_2 = \max_{x\in[-1;-0,5]} |f''(x)| = |f''(1)| = 12.$$

Виберемо x_0=1, тоді $|x_0 - x_*| \le 0,5$. З формули (25) маємо

$$q = \frac{12 \cdot 0,5}{2 \cdot 3,75} = 0,8 < 1.$$

Тобто всі умови теореми про збіжність методу Ньютона виконані. З формули (28) маємо, що для досягнення заданої точності достатньо провести 7 ітерацій. Відповідні обчислення наведені в табл. 5.

Табл.5

n	x_n	$f(x_n)$
0	01000000E+01	03000000E+01
1	06666667E+00	06296297E+00
2	05486111E+00	06804019E-01
3	05323902E+00	01218202E-02
4	05320890E+00	04395228E-06
5	05320889E+00	04230802E-07
6	05320889E+00	04230802E-07
7	05320889E+00	04230802E-07

3.8.1. Задачі

Знайти одним з ітераційних методів дійсні корені рівнянь з точністю ε *(наприклад* $\varepsilon=10^{-4}$*).*

1) $x^3 - 5x^2 + 4x + 0,092 = 0$
2) $x^3 - 4x^2 - 7x + 13 = 0$
3) $x^4 + x^3 - 6x^2 + 20x - 16 = 0$
4) $x^3 + \sin x - 12x + 1 = 0$
5) $x^3 - 10x^2 + 44x + 29 = 0$
6) $x + \sin x - 12x = 0,25$
7) $3x + \cos x + 1 = 0$
8) $x^3 - 3x^2 - 17x + 22 = 0$
9) $x^4 - 2x^3 - 3,74x^3 + 8,18x - 3,48 = 0$
10) $x^2 + 4\sin x - 1 = 0$
11) $x^3 + 4\sin x = 0$
12) $x^4 - 10x^3 + 48,16x^2 + 108,08x + 70,76 = 0$
13) $x^4 - 3x^3 + 20x^2 + 44x + 54 = 0$
14) $x^3 - 3x^2 - 14x - 8 = 0$
15) $x^3 - x - 1 = 0$
16) $3x - \cos x - 1 = 0$
17) $3x^2 - \cos^2 \pi x = 0$
18) $x^2 + 4\sin x = 0$
19) $(x-1)^3 + 0,5e^x = 0$
20) $x^3 + 4x - 6 = 0$

21) $x^3 - 2x^2 + x + 1 = 0$

22) $x^2 \lg x - 1 = 0$

23) $x^3 + 6x^2 + 9x + 2 = 0$

24) $\mathrm{sh}x - 12\mathrm{th}x - 0{,}311 = 0$

25) $e^x - 2(x-1)^2 = 0$

26) $e^{-x} + x^2 - 2 = 0$

27) $x^4 + 4x - 2 = 0$

28) $x^4 + 2x - 1 = 0$

29) $x^3 - x^2 + x - 3 = 0$

30) $x^5 + x - 3 = 0$

31) $x^7 + x + 4 = 0$

32) $2^x + x^2 - 1{,}15 = 0$

33) $3^{-x} - x^2 + 1 = 0$

34) $x^4 - 2x^3 + x^2 - 2x + 1 = 0$

35) $x^5 - 5x + 2 = 0$

36) $x^7 + 6x - 5 = 0$

37) $x^4 + 2x - 2 = 0$

38) $(x-1)^2 - \sin 2x = 0$

39) $x^4 + 2x^2 - 6x + 2 = 0$

40) $x^5 - 3x^2 + 1 = 0$

41) $5x^3 + 2x^2 - 15x - 6 = 0$

42) $x^6 - 3x^2 + x - 1 = 0$

43) $(x-1)^2 - 0{,}5e^x = 0$

44) $3x^4 + 4x^3 - 12x^2 - 5 = 0$

45) $x^2 \cos 2x = 1$

46) $x^2 - 3 + 0{,}5^x = 0$

47) $x^2 - 10\sin x = 0$

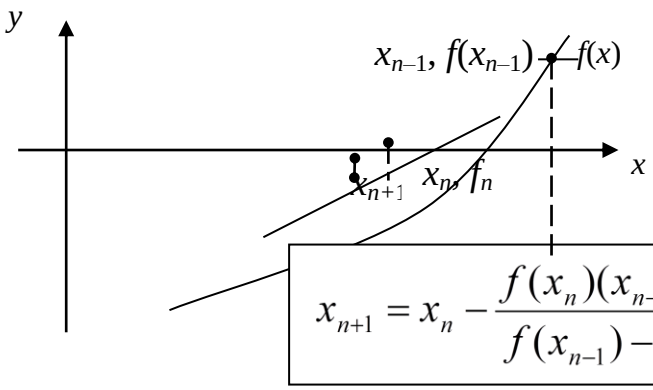
Тема 2. Метод січних. Метод градієнтного спуску. Метод релаксацій.

Якщо знаходження $f'(x)$ коштує дорогого, або неможливе, метод січних є кращим вибором, ніж метод Ньютона.

В цьому алгоритмі починають з двома початковими числами x_n та x_{n-1}. Абсциси *n, n-1* вибирають по одну сторону від кореня. На наступне уточнення x_{n+1} одержують з x_n та x_{n-1} як єдиний нуль лінійної функції, що приймає значення $f(x_n)$ в x_n та $f(x_{n-1})$ в x_{n-1}. Ця лінійна функція являє собою січну до кривої $f(x)$, що проходить через її точки з абсцисами x_n та x_{n-1} – звідси назва методу січних де $f_n = f(x_n)$. Праву частину краще не зводити до спільного знаменника.

Оскільки крок методу січних вимагає лишень одного обчислення функції, цей метод можна оцінити як більш швидкий в порівнянні з методом Ньютона.

Схема алгоритму для цього методу така ж, як і для методу Ньютона (дещо інший вигляд має ітераційна формула).



$$\frac{y - f(x_n)}{f(x_{n-1}) - f(x_n)} = \frac{x - x_n}{x_{n-1} - x_n}$$

$$x_{n+1} = x_n - \frac{f(x_n)(x_{n-1} - x_n)}{f(x_{n-1}) - f(x_n)}$$

$$x_{n+1} = x_n - \frac{f_n}{(f_{n-1} - f_n)/(x_{n-1} - x_n)};$$

Слід мати на увазі, що поблизу кореня x^* значення $f(x_n)$ та $f(x_{n-1})$ мало відрізняються, при діленні на їх різницю в методі виникає втрата значущих цифр, тому краще проводити обчислення за такою формулою:

$$x_{n+1} = x_n - \frac{f(x_n)}{[f(x_{n-1}) - f(x_n)]/(x_{n-1} - x_n)}.$$

Сутність градієнтного методу оптимізації полягає в тому, що задаються довільно, або виходячи з наявної апріорної інформації про положення точки екстремуму, початковим значенням вектора незалежних змінних $\vec{u}^{(0)}$.

Потім виконується зміна $\vec{u}^{(0)}$ на $\Delta\vec{u}^{(0)}$,тобто роблять крок $\Delta\vec{u}^{(0)}$ з метою наблизитися до точки екстремуму u_{OPT}. Потім роблять новий крок $\Delta\vec{u}^{(1)}$ і т.д.

Таким чином, на кожній ітерації обчислюється значення вектора для наступної ітерації:

$$\vec{u}^{(k+1)} = \vec{u}^{(k)} + \Delta\vec{u}^{(k)}.$$

Оскільки напрямок вектора градієнта вказує напрямок найшвидшого збільшення функції, то кроки Δu виконують у напрямку градієнта при пошуку максимуму й антиградієнта при пошуку мінімуму. Надалі, для визначеності, будемо розглядати задачу на мінімум. Тоді $\Delta\vec{u}^{(k)} = -\lambda\vec{S}^{(k)}$, де λ- множник, що визначає величину кроку $\Delta\vec{u}^{(k)}$; $\vec{S}^{(k)}$-одиничний вектор градієнта; k - номер ітерації. Знак "-" указує на напрямок антиградієнта.

У такий спосіб:

$$\vec{u}^{(k+1)} = \vec{u}^{(k)} - \lambda\vec{S}^{(k)}.$$

Алгоритм градієнтного пошуку часто застосовують у наступному виді:

$$\vec{u}^{(k+1)} = \vec{u}^{(k)} - h\nabla f(\vec{u}^{(k)}). \qquad (1.1)$$

У цьому випадку величина кроку $h\nabla f(\vec{u}^{(k)})$ змінюється автоматично відповідно до зміни величини градієнта.

Величина h зветься параметром кроку й залишається постійною. Алгоритм має ту перевагу, що при наближенні до точки мінімума довжина кроку автоматично зменшується.

Ітераційна формула (1.1) може бути записана в наступній формі:

$$\begin{bmatrix} u_1^{(k+1)} \\ --- \\ --- \\ --- \\ u_n^{(k+1)} \end{bmatrix} = \begin{bmatrix} u_1^{(k)} \\ --- \\ --- \\ --- \\ u_n^{(k)} \end{bmatrix} - h \begin{bmatrix} \dfrac{\partial f(u^{(k)})}{\partial u_1} \\ ----- \\ \dfrac{\partial f(u^{(k)})}{\partial u_n} \end{bmatrix},$$

або в скалярному виді:

$$u_i^{(k+1)} = u_i^{(k)} - h\frac{\partial f(\vec{u}^{(k)})}{\partial u_i} = u_i^{(k)} - h \cdot grad_{u_i} f(\vec{u}^{(k)}).$$

1.2.3 Вплив величини кроку на градієнтний пошук.

Питання вибору величини кроку є досить важливим і в остаточному підсумку визначає працездатність і швидкість збіжності алгоритму.

Якщо розмір кроку обраний занадто малим, то рух до оптимуму буде довгим через необхідність обчислення частинних похідних у багатьох точках.

При великому кроці в районі оптимуму можуть виникнути незатухаючі коливання незалежних змінних і знижується точність знаходження екстремуму.

При дуже великому кроці можливі розбіжні коливання.

На Рис зображені лінії постійного рівня функції $f(u_1,u_2)$.

Процес пошуку при великому h зображений послідовністю точок A_0, A_1, A_2, A_3.

Змістовий модуль 7. Апроксимація функцій.

Тема 1. Поняття про наближення функцій. Інтерполювання функції. Інтерполювання за Лагранжем.

Апроксимація функцій

Поняття про наближення функцій

Нехай величина "*y*" є функцією аргумента "*x*", тобто будь−якому значенню "*x*" з області визначення поставлено у відповідність значення "*y*".

На практиці досить часто бувають випадки, коли неможливо записати зв'язок між "*x*" та "*y*" у вигляді деякої залежності $y = f(x)$. Найбільш поширеним випадком, коли вид зв'язку між параметрами *x* та *y* невідомий, є задання цього зв'язку у вигляді таблиці $\{x_i, y_i\}$. Це означає, що дискретній множині значень аргумента $\{x_i\}$ поставлена у відповідність множина значень функції $\{y_i\}$ ($i=\overline{0,n}$). Цими значеннями можуть бути, наприклад, експериментальні дані. На практиці можуть бути потрібні значення величини у і в інших точках, відмінних від вузлів x_i. Однак одержати ці значення можна тільки експериментальним шляхом, що не завжди зручно і вигідно.

З точки зору економії часу та засобів доцільно було б використати наявні табличні дані для наближеного обчислення шуканого параметра "*y*" при будь−якому значенні (з деякої області, звичайно) визначального параметра "*x*", оскільки точний зв'язок $y = f(x)$ невідомий.

Цій меті служить задача про наближення (апроксимацію) функцій:

– дану функцію $f(x)$ потрібно наближено замінити (апроксимувати) деякою функцією $\varphi(x)$ так, щоб відхилення (в певному розумінні) $\varphi(x)$ від $f(x)$ в заданій області було найменшим. При цьому функція $\varphi(x)$ називається апроксимуючою.

Наприклад, в тому випадку, коли функція $f(x)$ задається у вигляді таблиці значень, задача апроксимації полягає в наступному: за табличними даними підібрати таку аналітичну залежність $\varphi(x)$, яка мала б просту структуру, згладжувала б особливості заданої експериментальної таблиці і найкращим чином відбивала б загальний хід зміни $f(x)$ в середньому. Тобто основна мета апроксимації – одержати швидкий (економний) алгоритм обчислення значень $f(x)$ для значень *x*, що не містяться в таблиці даних. Основне питання апроксимації – як вибрати $\varphi(x)$ і як оцінити відхилення $\varphi(x)$ від $f(x)$.

На практиці досить часто $\varphi(x)$ вибирається з класу алгебраїчних поліномів (многочленів)

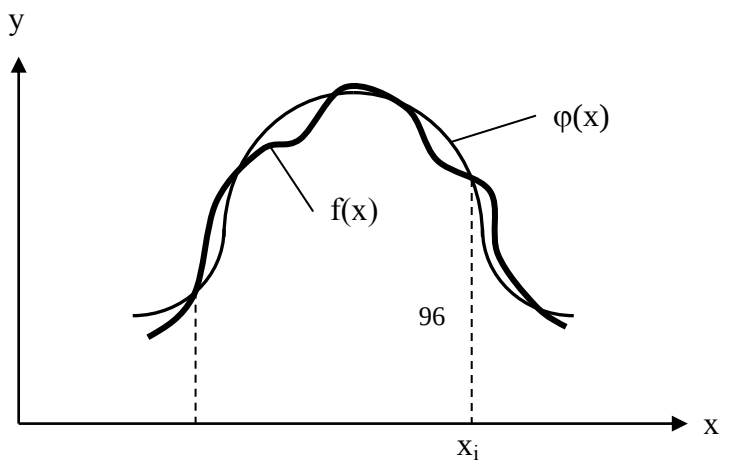
$$\varphi(x)=a_0 + a_1x + a_2x^2 +\ldots+ a_mx^m \qquad (1)$$

Якщо початкова функція задана таблично, тобто на множині окремих точок, то апроксимація називається точковою. Якщо ж початкова функція задана на неперервній множині точок (наприклад, на відрізку $[a; b]$), то апроксимація називається інтегральною (неперервною).

Одним з основних типів точкової апроксимації є інтерполяція. Вона полягає в наступному: для даної функції $y = f(x)$ будуємо функцію $\varphi(x)$, яка в заданих точках x_i ($i=\overline{0,n}$) приймає ті ж значення, що і функція $f(x)$, тобто

$$\varphi(x_i) = f(x_i),$$

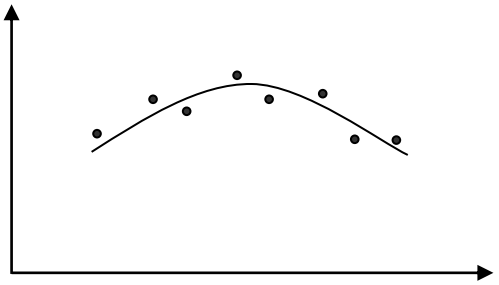
а в решті точок відрізку $[a; b]$ з області визначення $f(x)$, наближено представляє $f(x)$ з деякою похибкою. Точки x_i називають вузлами інтерполяції, а $\varphi(x)$ – інтерполюючою функцією. Найчастіше інтерполюючу функцію $\varphi(x)$ виражають через алгебраїчний многочлен степені *m*.



Інтерполяція в цьому випадку називається алгебраїчною. Якщо використовується один многочлен $\varphi(x) = P_n(x)$ для інтерполяції функції $f(x)$ на всьому інтервалі зміни аргумента x, тобто коли $m = n$ (m – максимальний степінь інтерполяційного многочлена), то це – глобальна інтерполяція.

Інтерполяційні многочлени можуть будуватися окремо для різних частин інтервалу зміни x. В цьому випадку маємо кускову (локальну) інтерполяцію. Як правило, інтерполяційні многочлени використовують для апроксимації функцій у проміжних точках між крайніми вузлами інтерполяції, тобто $x_0 < x < x_n$. Однак іноді вони використовуються і для наближеного обчислення функції зовні інтервалу ($x < x_0$, $x > x_n$). Це наближення називається екстраполяцією.

Таким чином, при інтерполюванні основною умовою є проходження графіка інтерполяційного многочлена через дані значення функції у вузлах інтерполяції. Однак виконання цієї умови в деяких випадках є недоцільним. Наприклад, при великому числі вузлів інтерполяції одержуємо високу степінь полінома у випадку глобальної інтерполяції (це пов'язано з рядом неприємностей – осциляція функції). Крім того, табличні дані можуть містити в собі похибки (якщо ці дані одержані шляхом вимірювань). Отже, інтерполюючий многочлен теж повторював би ці похибки. Вихід із цього становища може бути знайдений вибором такого многочлена, графік якого близько проходить від даних точок.



Поняття "близько" уточнюється при розгляді окремих видів наближення.

Середньо-квадратичне наближення. Степінь полінома m при цьому, як правило, значно менша від n. На практиці не вище 5,6. Мірою відхилення многочлена $\varphi(x)$ від заданої функції $f(x)$ на множині точок (x_i, y_i) ($i = \overline{0, n}$) є величина S, яка дорівнює сумі квадратів різниць між значеннями многочлена та функції в даних точках

$$S = \sum_{s=0}^{n} \left[\varphi(x_i) - y_i \right]^2$$

При побудові апроксимуючого многочлена потрібно підібрати коефіцієнти $a_0, a_1, \ldots, a_m$ так, щоб величина S була мінімальна. В цьому полягає ідея методу найменших квадратів.

Рівномірне наближення. В багатьох випадках, особливо при обробці експериментальних даних, середньоквадратичне наближення зручне, оскільки воно згладжує деякі неточності функції $f(x)$ і дає достатньо правильне уявлення про неї. Однак, іноді ставиться більш жорстка умова і вимагається, щоб у всіх точках деякого відрізку $[a, b]$ модуль відхилення многочлена $\varphi(x)$ від $f(x)$ був менший від деякого ε

$|f(x) - \varphi(x)| < \varepsilon$, $a \le x \le b$

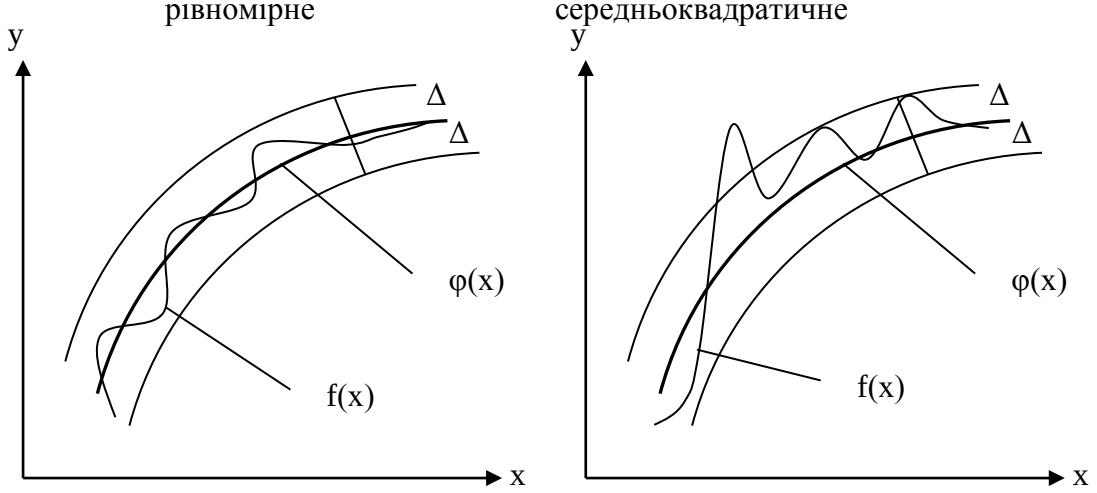
В цьому випадку маємо рівномірну апроксимацію. Тепер введемо такі поняття. Абсолютним відхиленням Δ многочлена $\varphi(x)$ від функції $f(x)$ на відрізку $[a, b]$ називається максимальне значення абсолютної різниці між ними на даному відрізку:

$\Delta = \max |f(x) - \varphi(x)|$, $a \le x \le b$

За аналогією можна ввести середньоквадратичне відхилення

$$\bar{\Delta} = \sqrt{\frac{S}{n}}.$$

На малюнку показано відмінність цих двох видів наближень.



Існує також поняття найкращого наближення функції $f(x)$ многочленом $\varphi(x)$ фіксованої степені m. В цьому випадку коефіцієнти многочлена $a_0, a_1, \dots, a_m$ слід вибирати так, щоб на заданому відрізку $[a, b]$ значення абсолютного відхилення Δ було мінімальне. Многолен $\varphi(x)$ при цьому називається многочленом найкращого рівномірного наближення.

4. Інтерполяція

Під апроксимацією розуміють операцію знаходження невідомих чисельних значень якоїсь величини за відомими її значеннями і чисельними значеннями інших величин, які пов'язані з розглядуваною.

Інтерполяція - частковий випадок апроксимації. Нехай в точках $x_0, x_1, x_2, \dots, x_n$ відомі значення $f(x_0), f(x_1), f(x_2) \dots f(x_n)$ деякої функції $f(x)$. Потрібно відновити функцію $f(x)$ при інших значеннях $x \neq x_i$ ($i = 0, 1, 2, \dots, n$). У цьому випадку будують достатньо просту для обчислення функцію $\varphi(x)$, яка в заданих точках $x_0, x_1, x_2, \dots, x_n$ приймає значення $f(x_0), f(x_1), f(x_2), \dots, f(x_n)$, а в решті точках відрізку $[a, b]$ (область визначення $f(x)$), наближено представляє $f(x)$ з деякою точністю. Задача побудови $\varphi(x)$ називається задачею інтерполювання. Найчастіше інтерполюючу функцію $\varphi(x)$ виражають через алгебраїчний многочлен деякої степені n.

Якщо аргумент x знаходиться зовні відрізку $[a, b]$, то поставлена задача називається екстраполюванням (екстраполяція).

Інтерполяція в цьому випадку називається алгебраїчною. Алгебраїчне інтерполювання функції $y = f(x)$ на відрізку $[a, b]$ полягає в наближеній заміні цієї функції на даному відрізку многочленом $P_n(x)$ степені n, тобто

$$f(x) \approx P_n(x), \qquad (1)$$

причому в точках $x_0, x_1, x_2, \dots, x_n$, $f(x_i) = P_n(x_i)$, $(i = \overline{0, n})$.

Відмітимо, що двох різних інтерполяційних многочленів одної й тої ж степені n існувати не може. Якщо вважати протилежне, приходимо до висновку, що різниця двох таких многочленів, що є многочленом степені не вище n, має $n + 1$ корінь, а отже тотожно дорівнює нулю.

4.1. Інтерполяційний поліном Лагранжа

Поставимо задачу: знайти многочлен степені $P_n(x)$ степені n, котрий в $n + 1$ даних точках $x_0, x_1, x_2, \dots, x_n$ (ці точки називаються вузлами інтерполяції) приймає дані значення $y_0, y_1, \dots, y_n$.

Для побудови $P_n(x)$ спочатку розглянемо допоміжні (іноді їх називають фундаментальні) многочлени $Q_n^k(x)$, тобто многочлени n-ї степені відносно x, котрі задовільняють таким умовам:

$$Q_n^k(x_i) = \begin{cases} 1, & i = k \\ 0, & i \neq k \end{cases} \quad \text{при} \quad (\text{к} = \overline{0, n}).$$

Ця властивість означає, що, наприклад, многочлен $Q_n^0(x)$ приймає в точці x_0 значення, рівне одиниці, а в решті вузлів – нуль; многочлен $Q_n^1(x)$ в вузлі x_1 приймає значення 1, а в решті

– нуль і т. д. В загальному випадку многочлен $Q_n^i(x)$ в вузлі x_i приймає значення 1, а в решті вузлів 0. Тоді шуканий многочлен:

$P_n(x) = y_0Q_n^0(x) + y_1Q_n^1(x) + y_2Q_n^2(x) + \ldots + y_nQ_n^n(x)$ (2).

Оскільки $x_0, x_1, x_2, \ldots, x_{k-1}, x_{k+1}, \ldots, x_n$ – нулі многочлена $Q_n^k(x)$, то

$Q_n^k(x) = c_k(x - x_0)(x - x_1)(x - x_2) \ldots (x - x_{k-1})(x - x_{k+1}) \ldots (x - x_n)$

(це просто інша форма запису полінома степені n).

Визначаючи c_k з умови $Q_n^k(x_k) = 1$, одержимо вираз для c_k (замість x підставляємо x_k)

$$c_k = \frac{1}{(x_k - x_0)(x_k - x_1)\ldots(x_k - x_{k-1})(x_k - x_{k+1})\ldots(x_k - x_n)} \qquad (2)$$

Тоді явний вираз для допоміжних многочленів

$$Q_n^k(x) = \frac{(x - x_0)(x - x_1)\ldots(x - x_{k-1})(x - x_{k+1})\ldots(x - x_n)}{(x_k - x_0)(x_k - x_1)\ldots(x_k - x_{k-1})(x_k - x_{k+1})\ldots(x_k - x_n)} = \prod_{i=0, i\neq k}^{n} \frac{x - x_i}{x_k - x_i} \qquad (3)$$

Формула (2) з врахуванням (3) приймає вигляд :

$$P_n(x) = \sum_{k=0}^{n} y_k \left(\prod_{i=0, i\neq k}^{n} \frac{x - x_i}{x_k - x_i} \right) = \qquad (4)$$

$$= \sum_{k=0}^{n} y_k \frac{(x - x_0)(x - x_1)\ldots(x - x_{k-1})(x - x_{k+1})\ldots(x - x_n)}{(x_k - x_0)(x_k - x_1)\ldots(x_k - x_{k-1})(x_k - x_{k+1})\ldots(x_k - x_n)}$$

Многочлен, що визначається за формулою (4) називається інтерполяційним многочленом Лагранжа, а допоміжні многочлени (3) – коефіцієнтами Лагранжа.

Введемо позначення

$$\omega(x) = (x - x_0)(x - x_1)\ldots(x - x_n)$$

Розглянемо похідну в точці $x_к$

$$\omega'(x_k) = (x_k - x_0)(x_k - x_1)\ldots(x_k - x_{k-1})(x_k - x_{k+1})\ldots(x_k - x_n)$$

Звідси

$$Q_n^k(x) = \frac{\omega(x)}{(x - x_k)\omega'(x)}$$

$$P_n(x) = \omega(x)\sum_{k=0}^{n} \frac{y_k}{(x - x_k)\omega'(x_k)}$$

Розглянемо інтерполяційну формулу Лагранжа для випадку рівновіддалених вузлів інтерполяції, тобто $x_1 - x_0 = x_2 - x_1 = \ldots = x_n - x_{n-1} = h$. Зробимо заміну $x = ht + x_0$, тоді

$t_0 = 0; t_1 = 1; t_2 = 2; \ldots t_n = n$

$x - x_k = h(t - k)$, $\omega(x) = n^{n+1}\omega^*(t)$

$\omega^*(t) = t(t - 1)(t - 2) \ldots (t - n)$

$\omega'(x_k) = (-1)^{n-k}k!(n - k)!h^n$

$$f(x) = f(ht + x_0) \approx t(t - 1)(t - 2) \ldots (t - n)^* \sum_{k=0}^{n} \frac{(-1)^{n-k} y_k}{(t - k)k!(n - k)!}$$

Приклад.

Знайти інтерполяційний многочлен Лагранжа для функції, заданої таблицею

x_i	-3	-1	1	2
y_i	8	6	4	18

При n=3, формула (4) приймає вигляд

$$P(3)=y_0\frac{(x-x_1)(x-x_2)(x-x_3)}{(x_0-x_1)(x_0-x_2)(x_0-x_3)}+y_1\frac{(x-x_0)(x-x_2)(x-x_3)}{(x_1-x_0)(x_1-x_2)(x_1-x_3)}+$$

$$+y_2\frac{(x-x_0)(x-x_1)(x-x_3)}{(x_2-x_0)(x_2-x_1)(x_2-x_3)}+y_3\frac{(x-x_0)(x-x_1)(x-x_2)}{(x_3-x_0)(x_3-x_1)(x_3-x_2)}$$

Підставляючи значення $x_к$ та $y_к$ ($к=\overline{0,3}$).

$$P_3(x)=8\frac{(x+1)(x-1)(x-2)}{(-3+1)(-3-1)(-3-2)}+6\frac{(x+3)(x-1)(x-2)}{(-1+3)(-1-1)(-1-2)}+$$

$$+4\frac{(x+3)(x+1)(x-2)}{(1+3)(1+1)(1-2)}+18\frac{(x+3)(x+1)(x-1)}{(2+3)(2+1)(2-1)}=x^3+3x^2-2x+2$$

$P_3(x)=x^3+3x^2-2x+2$

Тема 2. Інтерполювання за Ньютоном. Інтерполювання за Ермітом. Інтерполяція таблиць.

Інтерполяційний поліном Ньютона

Інтерполяційна формула Лагранжа має два суттєвих недоліки:

1) формула громіздка- кожен доданок є многочленом n-го степеня;
2) якщо з якоїсь причини додаються вузли інтерполювання (наприклад, якщо отримана інтерполяційна формула неточна), то всі обчислення необхідно повторювати знову – ні один із доданків формули Лагранжа не зберігається.

Розглянем форму запису інтерполяційного полінома $P_n(x)$, яка допускає уточнення результатів інтерполяції послідовним додаванням нових вузлів. При цьому будем використовувати таке поняття як розділені різниці функцій.

Нехай маємо функцію $f(x)$ і не обов"язково рівновіддалені вузли інтерполяції x_i (i=0, 1, 2, … , n).

Розділеними різницями 1-го порядку називають величини, які мають зміст, наприклад, середніх швидкостей зміни функції:

$$f(x_i;x_j)=\frac{f(x_j)-f(x_i)}{x_j-x_i}$$

$$f(x_0;x_1)=\frac{f(x_1)-f(x_0)}{x_1-x_0} \qquad (5)$$

$$f(x_1;x_2)=\frac{f(x_2)-f(x_1)}{x_2-x_1}$$

Розділені різниці другого порядку визначаються співвідношеннями

$$f(x_i;x_j;x_k)=\frac{f(x_j;x_k)-f(x_i;x_j)}{x_k-x_i}$$

$$f(x_0;x_1;x_2)=\frac{f(x_1;x_2)-f(x_0;x_1)}{x_2-x_0} \qquad (6)$$

$$f(x_1;x_2;x_3)=\frac{f(x_2;x_3)-f(x_1;x_2)}{x_3-x_1}$$

Аналогічно, розділена різниця k−го порядку визначається через розділені різниці (k−1) порядку за рекурентною формулою:

$$f(x_i;x_{i+1};...;x_{i+k})=\frac{f(x_{i+1};x_{i+2}...x_{i+k})-f(x_i;x_{i+1};...;x_{i+k-1})}{x_{i+k}-x_i} \qquad (7)$$

Тепер перейдемо безпосередньо до самого інтерполяційного полінома Ньютона.

Маєм, наприклад, один вузол інтерполяції x_0.

Виходячи із визначення розділеної різниці 1−го порядку $f(x; x_0)$ маємо:

$$f(x_0;x)=\frac{f(x)-f(x_0)}{x-x_0}=\frac{f(x_0)-f(x)}{x_0-x}$$

$$f(x_0;x)=\frac{f(x)-f(x_0)}{x-x_0}=y_0 \Rightarrow f(x)=y_0+(x-x_0)f(x;x_0)$$

Для розділених різниць другого порядку (два вузли − x_0, x_1)

$$f(x;x_0;x_1)=\frac{f(x;x_0)-f(x_0;x_1)}{x-x_1}$$

$$f(x;x_0)=f(x_0;x_1)+(x-x_1)f(x;x_0;x_1)$$

Підставляючи це значення у формулу для f(x)

$$f(x)=y_0+(x-x_0)\left[f(x_0;x_1)+(x-x_1)f(x;x_0;x_1)\right]=$$

$$y_0+(x-x_0)f(x_0;x_1)+(x-x_0)(x-x_1)f(x;x_0;x_1)$$

Повторюючи цей процес, отримаємо (для n+1 вузлів інтерполяції):

$$f(x)=y_0+(x-x_0)f(x_0;x_1)+(x-x_0)(x-x_1)f(x_0;x_1;x_2)+$$

$$+...+(x-x_0)(x-x_1)...(x-x_{n-1})f(x_0;x_1;x_2...x_{n-1};x_n)+$$

$$+(x-x_0)(x-x_1)...(x-x_n)f(x;x_0;x_1;...x_n)= \qquad (8)$$

$$=P_n(x)+(x-x_0)(x-x_1)...(x-x_n)f(x;x_0;x_1;...x_n)$$

Оскільки $P_n(x)$ − інтерполяційний поліном для функції $f(x)$, то його значення у вузлах інтерполяції співпадають із значеннями функції $f(x)$ (а, значить, і співпадають і розділені різниці)

$P_n(x_i) = f(x_i) = y_i$, ($i=\overline{0,n}$),

оскільки залишковий член в цих вузлах

$$R_n(x)=(x-x_0)(x-x_1)...(x-x_n)f(x;x_0;x_1;...x_n)=0$$

(x приймає значення x_0, x_1, … , x_n, тому один із співмножників завжди рівний 0, через те залишковий член у вузлах інтерполяції дорівнює нулю).

Тому замість (8) можна записати

$$P_n(x) = y_0 + (x - x_0)f(x_0; x_1) + \ldots + (x - x_0)(x - x_1)\ldots$$
$$\ldots(x - x_{n-1})f(x_0; x_1; x_2 \ldots x_{n-1}; x_n) =$$
$$= y_0 + \sum_{k=1}^{n}(x - x_0)(x - x_1)\ldots(x - x_{k-1})f(x_0; x_1; \ldots x_k) = \quad (9)$$
$$= y_0 + \sum_{k=1}^{n}\left(\prod_{i=0}^{k-1}(x - x_i)\right)f(x_0; x_1; \ldots x_k)$$

Це і є інтерполяційний поліном Ньютона з розділеними різницями.

Для того, щоб пересвідчитись, що інтерполяційний поліном приймає значення y_i в вузлах інтерполяції x_i, візьмемо два вузли x_0 та x_1

$n = 2$ $\quad f(x) = y_0 + (x - x_0)f(x_0; x_1) + (x - x_0)(x - x_1)f(x; x_0; x_1)$

При $x = x_1$

$f(x_1) = y_0 + (x_1 - x_0)(f(x_1) - f(x_0))/(x_1 - x_0) = y_0 + f(x_1) - f(x_0)$;

$f(x_1) = f(x_1)$

Якщо маєм чотири вузли інтерполяції (n=3), то поліном Ньютона має вигляд:

$P_n(x) = y_0 + (x - x_0)f(x_0; x_1) + (x - x_0)(x - x_1)f(x_0; x_1; x_2) +$
$+ (x - x_0)(x - x_1)(x - x_2)f(x_0; x_1; x_2; x_3)$

Якщо ж маєм вже шість вузлів, тобто n=5, то йде просте нарощу-вання формули:

$P_n(x) = y_0 + (x - x_0)f(x_0; x_1) + (x - x_0)(x - x_1)f(x_0; x_1; x_2) +$
$+ (x - x_0)(x - x_1)(x - x_2)f(x_0; x_1; x_2; x_3) +$
$+ (x - x_0)(x - x_1)(x - x_2)(x - x_3)f(x_0; x_1; x_2; x_3; x_4) +$
$+ (x - x_0)(x - x_1)(x - x_2)(x - x_3)(x - x_4)f(x_0; x_1; x_2; x_3; x_4; x_5)$.

Приклад.

Знайти інтерполяційний поліном Ньютона:

x	−3	−1	1	2
y	8	6	4	18

При n = 3 інтерполяційний поліном Ньютона буде мати вигляд:

$P_n(x) = y_0 + (x - x_0)f(x_0; x_1) + (x - x_0)(x - x_1)f(x_0; x_1; x_2) +$
$+ (x - x_0)(x - x_1)(x - x_2)f(x_0; x_1; x_2; x_3)$

j	x_j	y_j	k=1	k=2	k=3
0	$x_0 = -3$	$y_0 = 8$	$\frac{6-8}{-1+3} = -1$	0	
1	$x_1 = -1$	$y_1 = 6$	$\frac{4-6}{1+1} = -1$	$\frac{14+1}{2+1} = 5$	$\frac{5-0}{2+3} = 1$
2	$x_2 = 1$	$y_2 = 4$	$\frac{18-4}{2-1} = 14$		
3	$x_3 = 2$	$y_3 = 18$			

$P_3(x) = 8 + (x + 3)(-1) + (x + 3)(x + 1)*0 + (x + 3)(x + 1)(x - 1)*1 =$
$= 8 - x - 3 + (x + 3)(x^2 - 1) = 5 - x + x^3 + 3x^2 - x - 3 =$
$= x^3 + 3x^2 - 2x + 2$

Перш ніж приступати до заповнення таблиці, розпишемо розділені різниці

Розділені різниці 1−го порядку

$$f(x_0;x_1)=\frac{y_1-y_0}{x_1-x_0}=\frac{6-8}{-1+3}=-1;$$

$$f(x_1;x_2)=\frac{y_2-y_1}{x_2-x_1}=\frac{4-6}{1+1}=-1;$$

$$f(x_2;x_3)=\frac{y_3-y_2}{x_3-x_2}=\frac{18-4}{2-1}=14$$

Розділені різниці 2−го порядку

$$f(x_0;x_1;x_2)=\frac{f(x_1;x_2)-f(x_0;x_1)}{x_2-x_0}=\frac{-1+1}{1+3}=0$$

$$f(x_1;x_2;x_3)=\frac{f(x_2;x_3)-f(x_1;x_2)}{x_3-x_1}=\frac{14+1}{2+1}=5$$

Розділені різниці 3−го порядку

$$f(x_0;x_1;x_2;x_3)=\frac{f(x_1;x_2;x_3)-f(x_0;x_1;x_2)}{x_3-x_0}=\frac{5-0}{2+3}=1$$

При написанні програми будемо користуватися наступним алгоритмом: позначимо через k – порядок розділених різниць ($k=\overline{1,n}$ – межі зміни k, де n – порядок (найвищий) інтерполюючого полінома), а через i – число розділених різниць ($i=\overline{n,k}$ – межі зміни i) для даного порядку k.

k=1 $y_3'=\frac{y_3-y_2}{x_3-x_2}\Rightarrow y_2'$ збереглося

$$y_2'=\frac{y_2-y_1}{x_2-x_1}=y_1$$

$$y_1'=\frac{y_1-y_0}{x_1-x_0}=y_0$$

k=2 $y_3''=\frac{y_3'-y_2'}{x_3-x_1}$

$$y_2''=\frac{y_2'-y_1'}{x_2-x_0}$$

k=3 $y_3'''=\frac{y_3''-y_2''}{x_3-x_0}$

y^k – кількість штрихів – це порядок

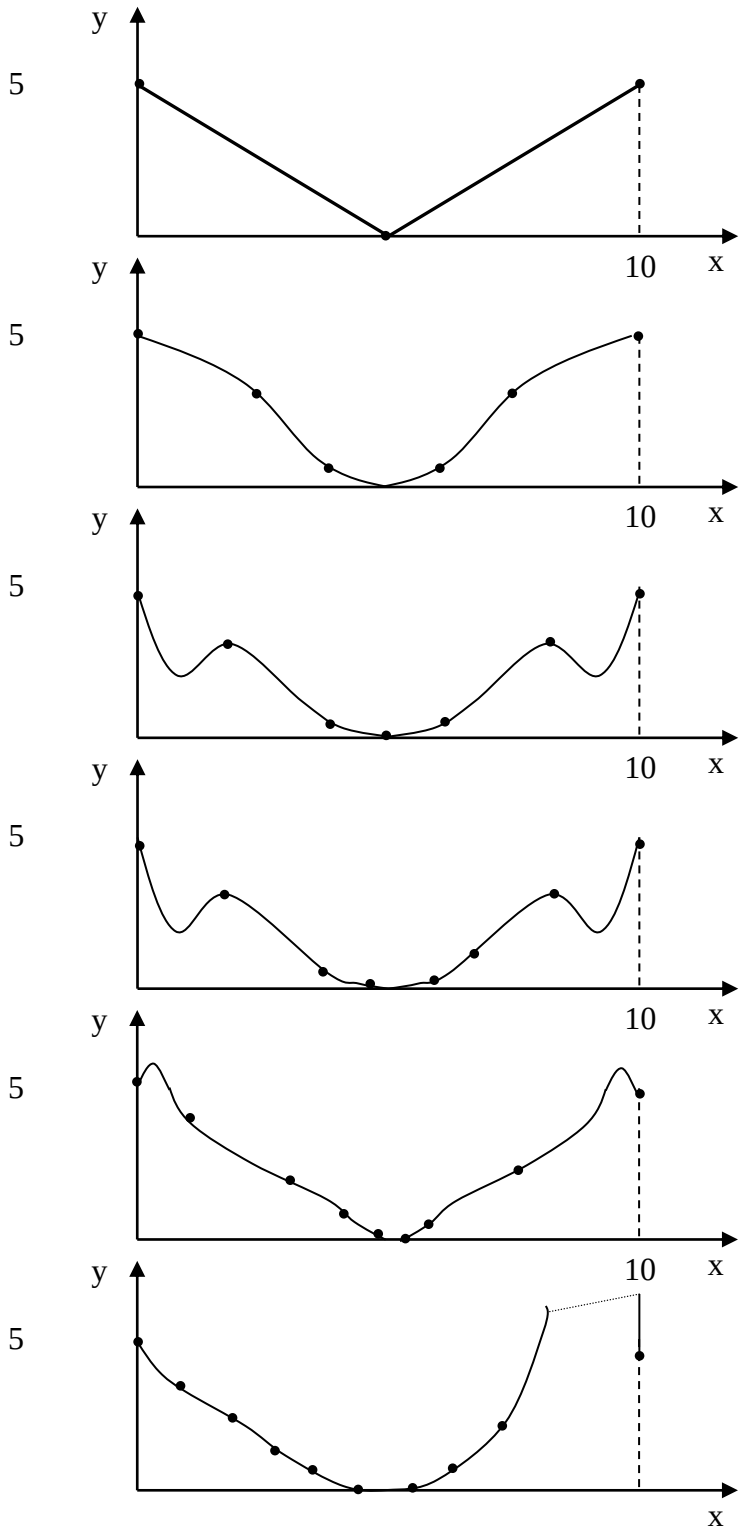
k = 1 до n

Для i = 0 до n ввести значення x_i, y_i

Для k = 1 до n

Для i = n до k з кроком –1

$y_i = (y_i - y_{i-1})/(x_i - x_{i-1})$
Тоді відповідно збережуться y_0, y_1', y_2'', y_3'''



Інтерполяція функції $y = |x - 5|$ з допомогою полінома Ньютона на 6-10 точках.

Підбір емпирічних формул

1. Характер експериментальних дослідних даних

При інтерполюванні функцій використовується відома умова $\varphi(x_i) = = f(x_i)$ – рівність значень інтерполяційного многочлена та даної функції у вузлах інтерполяції. Якщо $f(x_i)$ містить похибку, то апроксимуючий многочлен $\varphi(x_i)$ цю похибку повторить.

При обробці експериментальних даних, одержаних в результаті спостережень або вимірювань, потрібно мати на увазі похибки цих даних. Ці похибки можуть бути зумовлені недосконалістю вимірювального приладу, суб'єктивними причинами, різноманітними випадковими факторами.

Похибки експериментальних даних (ЕД) можна умовно розбити на 3 групи:

1) систематичні;
2) випадкові;
3) грубі.

Систематичні – дають, як правило, відхилення в одну сторону від істинного значення вимірювальної величини. Вони можуть бути сталими або закономірно змінюватись при повторі експерименту і їх причина та характеристики відомі. Систематичні похибки можуть бути викликані умовами експерименту (вологістю, температурою середовища і т. п.), дефектом вимірювального приладу, його поганим регулюванням (наприклад зміщення нуля) і т. п. Такі похибки усуваються наладкою апаратури або внесенням відповідних поправок.

Випадкові похибки – визначаються великим числом факторів, які не можуть бути усунуті або достатньо точно враховані при вимірюваннях або обробці результатів. Вони носять випадковий (несистематичний) характер, дають відхилення від середнього значення величини в різні сторони. Вони не можуть бути усунуті в експерименті. З точки зору теорії ймовірності математичне сподівання випадкової похибки дорівнює нулю.

Статистична обробка експериментальних даних дозволяє знайти значення випадкової похибки і довести її до деякого прийнятного рівня шляхом повторювання вимірювань.

Грубі похибки (помилки) явно спотворюють результат вимірювання. Вони надмірно великі і як правило зникають при повторі досліду. Вимірювання з такими похибками відкидаються і не враховується при остаточній обробці результатів вимірювань.

Таким чином, в ЕД завжди є випадкові похибки. Вони можуть бути зменшені шляхом багатократних повторних вимірювань. Однак для цього потрібні значні матеріальні та часові ресурси. Значно дешевше і швидше уточнені дані можна отримати шляхом спеціальної математичної обробки наявних результатів вимірювань (наприклад, статистична обробка дає значення розподілу похибок вимірювань, найбільш ймовірний діапазон зміни шуканої величини (довірчий інтервал) та інші параметри).

Ми розглянемо тільки визначення зв'язку між вхідними параметрами x та шуканою величиною y на підставі результатів вимірювань

2. Емпіричні формули

Маємо таблицю значень:

x_1	x_2	…	x_n
y_1	y_2	…	y_n

Необхідно знайти наближену залежність $y = f(x)$, значення якої при $x = x_i$ ($i=\overline{1,n}$), мало відрізняються від дослідних даних y_i. Наближена функціональна залежність $y = f(x)$, яка одержана на основі експериментальних даних, називається емпіричною формулою.

Одним із способів одержання емпіричних формул є метод найменших квадратів (МНК). Будем вважати, що тип емпіричної формули відомий (це, наприклад, пряма, парабола, многочлен чи інше) і її можна зобразити у вигляді

$$y = \varphi(x, a_0, a_1, \dots, a_m), \qquad (1)$$

де φ - відома функція;

$a_0, a_1, \dots, a_m$ – невідомі сталі параметри.

Задача полягає в тому, щоб визначити такі значення цих параметрів, при яких емпірична формула дає достатньо добре наближення таблично заданої функції.

Ідея МНК полягає в наступному. Запишемо суму квадратів відхилень для всіх точок x_i ($i=\overline{1,n}$)

$$S = \sum_{i=1}^{n} \left[\varphi(x_i, a_0, a_1, \dots, a_m) - y_i\right]^2 \qquad (2)$$

Параметри $a_0, a_1, \dots, a_m$ емпіричної формули (1) будем шукати з умови min функції $S = S(a_0, a_1, \dots, a_m)$. Оскільки тут параметри $a_0, a_1, \dots, a_m$ виступають в ролі незалежних змінних функції S, то її min знайдемо, прирівнюючи до нуля частинні похідні за цими змінними

$$\frac{\partial S}{\partial a_0} = 0; \quad \frac{\partial S}{\partial a_1} = 0; \quad \dots \quad ; \frac{\partial S}{\partial a_m} = 0; \qquad (3)$$

Розглянем випадок, коли за емпіричну функцію вибирають многочлен

$\varphi(x) = a_0 + a_1 x + a_2 x^2 + \dots + a_m x^m$ (4)

Тоді формула визначення суми квадратів відхилень S зобразиться так

$$S = \sum_{i=1}^{n} \left(a_0 + a_1 x_i + a_2 x_i^{\,2} + \dots a_m x_i^{\,m} - y_i\right)^2 \qquad (5)$$

Тоді система рівнянь для визначення $a_0, a_1, \dots, a_m$ з врахуванням (3) набере вигляду

$$\begin{aligned}
&\frac{\partial \mathrm{S}}{\partial \mathrm{a}_0} = 2\sum_{i=1}^{n} \left(a_0 + a_1 x_i + a_2 x_i^{\,2} + \dots a_m x_i^{\,m} - y_i\right) = 0 \\
&\frac{\partial \mathrm{S}}{\partial \mathrm{a}_1} = 2\sum_{i=1}^{n} \left(a_0 + a_1 x_i + a_2 x_i^{\,2} + \dots a_m x_i^{\,m} - y_i\right) x_i = 0 \\
&\text{– –} \qquad (6) \\
&\frac{\partial \mathrm{S}}{\partial \mathrm{a}_m} = 2\sum_{i=1}^{n} \left(a_0 + a_1 x_i + a_2 x_i^{\,2} + \dots a_m x_i^{\,m} - y_i\right) x_i^{\,m} = 0
\end{aligned}$$

Збираючи коефіцієнти при невідомих $a_0, a_1, \dots, a_m$, одержимо наступну систему рівнянь (2 перед знаком суми опускаєм – сталий множник, який не змінює коренів системи):

$$\begin{aligned}
&n a_0 + a_1 \sum_{i=1}^{n} x_i + a_2 \sum_{i=1}^{n} x_i^{\,2} + \dots + a_m \sum_{i=1}^{n} x_i^{\,m} = \sum_{i=1}^{n} y_i \\
&a_0 \sum_{i=1}^{n} x_i + a_1 \sum_{i=1}^{n} x_i^{\,2} + a_2 \sum_{i=1}^{n} x_i^{\,3} + \dots + a_m \sum_{i=1}^{n} x_i^{\,m+1} = \sum_{i=1}^{n} y_i x_i \\
&\text{– –} \qquad (7) \\
&a_0 \sum_{i=1}^{n} x_i^{\,m} + a_1 \sum_{i=1}^{n} x_i^{\,m+1} + a_2 \sum_{i=1}^{n} x_i^{\,m+2} + \dots + a_m \sum_{i=1}^{n} x_i^{\,2m} = \sum_{i=1}^{n} x_i^{\,m} y_i
\end{aligned}$$

Систему (7) можна записати в більш компактному вигляді:

$$\begin{aligned}
&c_0 a_0 + c_1 a_1 + c_2 a_2 + \dots + c_m a_m = d_0, \\
&c_1 a_0 + c_2 a_1 + c_3 a_2 + \dots + c_{m+1} a_m = d_1, \qquad (8)
\end{aligned}$$

$$c_m a_0 + c_{m+1} a_1 + c_{m+2} a_2 + \ldots + c_{2m} a_m = d_m ,$$

де $$c_j = \sum_{i=1}^{n} x_i^j , \qquad j = 0, 1, 2, \ldots, 2m \tag{9}$$

$$d_k = \sum_{i=1}^{n} x_i^{\,k} y_i , \qquad k = 0, 1, 2, \ldots, m \tag{10}$$

Поліном (4) степені $m < n$, де n − число пар x_i, y_i забезпечує апроксимацію таблично заданої функції $y_i(x_i)$ з мінімальною середньоквадратичною похибкою:

$$E = \sqrt{\frac{\left(\sum_{i=1}^{n} \varepsilon_i^{\,2} \right)}{(n+1)}} \tag{11}$$

Якщо $m = n$, то має місце звичайна інтерполяція, тобто

$$\varphi(x_i) = y_i$$

Зауваження щодо побудови програми .

Система (8) − це система лінійних алгебраїчних рівнянь відносно невідомих a_0, a_1, … , a_m. Коефіцієнти при невідомих одержуються за формулами (9) та (10). Для обчислення і зберігання коефіцієнтів c_j потрібен масив із $(2m+1)$ чисел, а для d_k − масив із $(m + 1)$ чисел, де m − степінь полінома, яка задається на початку роботи програми.

Потрібно ввести в циклі ($i = \overline{1, n}$) пари значень x_i, y_i, потім сформувати коефіцієнти при невідомих c_j та вільні члени d_k .

Одержану таким чином систему лінійних алгебраїчних рівнянь розв'язати методом Гауса (з частковим вибором головного елемента), одержуючи значення параметрів a_0, a_1, … , a_m апроксимуючого полінома $\varphi(x)$.

На практиці використовується поліноміальна апроксимація за МНК з автоматичним вибором степені полінома. Алгоритм наступний: задається початкове значення m, потім шукається коефіцієнти полінома a_0, a_1, … , a_m , за формулою (11) обчислюється середньоквадратична похибка і порівнюється із заданою E1. Якщо E > заданої, степінь m збільшується на 1 і все повторюється. Обчислення припиняється при E < E1.

Тема 3. Похибка інтерполяції. Збіжність процесу інтерполяції. Інтерполяційні сплайни.

Змістовий модуль 8. Чисельне розв'язання диференційних рівнянь.

Тема 1. Основні поняття. Диференційні рівняння з однокроковим методом. Метод Ейлера і Рунге-Кутта, схеми Рунге-Кутта другого і четвертого порядку.

Метод Ейлера

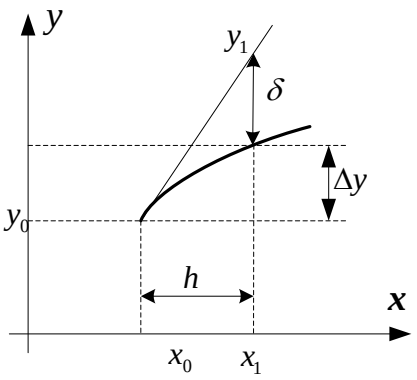
Однокрокові методи призначені для розв'язування диференціальних рівнянь першого порядку виду

$$\frac{dy}{dx} = f(x,y), y(x_0) = y_0 \qquad (6)$$

Метод Ейлера є найпростішим методом розв'язування задачі Коші. Він дозволяє інтегрувати ДР першого порядку. Точність його не велика.

h - настільки мале, що значення функції y мало відрізняється від лінійної функції

$tg\alpha$ - тангенс кута нахилу дотичної в точці x_0



$$y_1 = y_o + h \cdot tg\alpha = y_o + hy'(x_0) = y_0 + hf(x_o, y_0)$$

Тобто крива заміняється дотичними. Рух відбувається не по інтегральній кривій, а по відрізках дотичної .

Метод Ейлера базується на розкладі функції y' в ряд Тейлора в околі точки x_0

$$y(x_0 + h) = y(x_0) + hy'(x_0) + \frac{1}{2!}h^2 y''(x_0) + + \frac{h^3}{3!}y'''(x_0) + ... + \frac{h^p}{p!}y^{(P)}(x_0) + ...$$

$$y(x_i + \Delta x) = y(x_i) + y'(x_i)\Delta x_i + 0(\Delta x_i^2)$$

Якщо h мале, то, члени розкладу, що містять в собі h^2, h^3 і т.д. є малими високих порядків і ними можна знехтувати.

Тоді $y(x_0 + h) \approx y(x_0) + hy'(x_0) = y(x_0) + hf(x_o, y_o)$

Похідну $y'(x_0)$ знаходимо з рівняння (6), підставивши в нього початкову умову. Таким чином можна знайти наближене значення залежної змінної при малому зміщенні h від початкової точки. Цей процес можна продовжувати, використовуючи співвідношення.

$y_{n+1} = y_n + hy'(x_n) = y_n + hf(x_n, y_n)$,

роблячи як завгодно багато кроків.

Похибка методу має порядок h^2, оскільки відкинуті члени, що містять h в другій і вище степенях.

Недолік методу Ейлера - нагромадження похибок, а також збільшення об'ємів обчислень при виборі малого кроку h з метою забезпечення заданої точності.

В методі Ейлера на всьому інтервалі h тангенс кута нахилу дотичної приймається незмінним і рівним $y'(x_n)$. Очевидно, що це призводить до похибки, оскільки кути нахилу дотичної в точках x_n та $x_{n+1} = x_n + h$ різні. Точність методу можна суттєво підвищити, якщо покращити апроксимацію похідної.

Це можна зробити, якщо, наприклад, використати середнє значення похідної на початку та в кінці інтервалу.

Модифікований метод Ейлера

В модифікованому методі Ейлера (метод Ейлера з перерахунком) спочатку обчислюється значення функції в наступній точці за звичайним методом Ейлера.

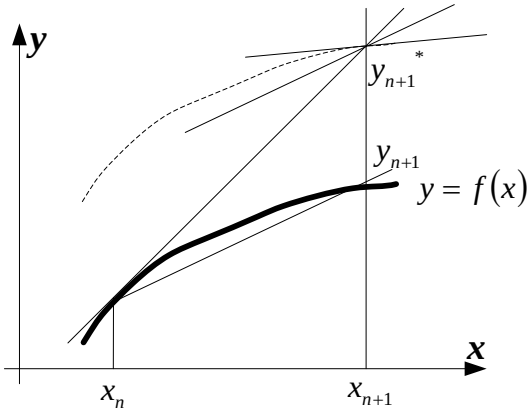
$$y^*_{n+1} = y_n + hf(x_n, y_n) \tag{9}$$

Воно використовується для обчислення наближеного значення похідної в кінці інтервалу $f(x_{n+1},\ y^*_{n+1})$.

Обчисливши середнє між цим значенням похідної та її значенням на початку інтервалу, знайдемо більш точне значення y_{n+1}:

$$y_{n+1} = y_n + \frac{1}{2}h\left[f(x_n, y_n) + f(x_{n+1}, y^*_{n+1})\right] \tag{10}$$

Цей прийом ілюструється на рисунку.



В обчислювальній практиці використовується також метод Ейлера-Коші з ітераціями:

1) знаходиться грубе початкове наближення (за звичайним методом Ейлера)

$$y^{(0)}_{n+1} = y_n + hf(x_n, y_n)$$

2) будується ітераційний процес

$$y^k_{n+1} = y_n + \frac{h}{2}\left[f(x_n, y_n) + f(x_{n+1}, y^{(K-1)}_{n+1})\right], k = 1,2,3... \tag{14}$$

Ітерації продовжують до тих пір, доки два послідовні наближення не співпадуть з заданою похибкою ε. Якщо після декількох ітерацій співпадіння нема, то потрібно зменшити крок h.

$$\left|y^{(K)}_{n+1} - y^{(K-1)}_{n+1}\right| < \varepsilon$$

Тобто в модифікованому методі Ейлера, в методі Ейлера-Коші з ітераціями спочатку (на першому етапі) знаходиться наближення для y_{n+1}, а потім воно вже коригується за формулами (10) або (14).

Метод Рунге – Кутта четвертого порядку

Метод Рунге-Кутта об'єднує ціле сімейство методів розв'язування диференціальних рівнянь першого порядку. Найбільш часто використовується метод четвертого порядку.

В методі Рунге-Кутта значення y_{n+1} функції y, як і в методі Ейлера, визначається за формулою

$$y_{n+1} = y_n + \Delta y_n \quad (1)$$

Якщо розкласти функцію y в ряд Тейлора і обмежитись членами до h^4 включно, то приріст Δy можна записати у вигляді

$$\Delta y = y(x+h) - y(x) = hy'(x) + \frac{h^2}{2!}y''(x) + + \frac{h^3}{3!}y'''(x) + \frac{h^4}{4!}y^{IV}(x) \quad (11)$$

Замість того, щоб обчислювати члени ряду за формулою (11) в методі Рунге-Кутта використовують наступні формули.

$$y_{n+1} = y_n + \frac{K_1 + 2K_2 + 2K_3 + K_4}{6}$$

$$K_1 = hf(x_n, y_n)$$

$$K_3 = hf\left(x_n + \frac{1}{2}h, y_n + \frac{1}{2}K_2\right)$$

$$K_4 = hf(x_n + h, y_n + K_3)$$

Це метод четвертого порядку точності.

Похибка на кожному кроці має порядок h^5. Таким чином метод Рунге-Кутта забезпечує значно вищу точність ніж метод Ейлера, однак вимагає більшого об'єму обчислень в порівнянні з методом Ейлера. Це досить часто дозволяє збільшити крок h.

Деколи зустрічається інша форма представлення методу Рунге-Кутта 4-го порядку точності.

$$y_{n+1} = y_n + \frac{h}{6}\cdot(K_1 + 2\cdot K_2 + 2\cdot K_3 + K_4)$$

$$K_1 = f(x_n, y_n)$$

$$K_2 = f\left(x_n + \frac{h}{2}, y_n + \frac{h}{2}\cdot K_1\right)$$

$$K_3 = f\left(x_n + \frac{h}{2}, y_n + \frac{h}{2}K_2\right)$$

$$K_4 = f\left(x_n + \frac{h}{2}, y_n + \frac{h}{2}K_3\right)$$

В більшості стандартних програм ЕОМ найчастіше використовується (схема) метод четвертого порядку (Рунге-Кутта).

Тема 2. Багатокрокові методи, метод прогнозу і корекції. Метод Адамса. Задачі Коші.

У цих методах для обчислення значення нової точки використовується інформація про декілька значень, що отримані раніше. Для цього використовуються дві формули: прогнозу і корекції. Алгоритм обчислення для всіх методів прогнозу і корекції однаковий та зображений на рисунку4.3. Вказані методи відрізняються лише формулами і не мають властивості "самостартування", оскільки вимагають знання попередніх значень. Перш ніж використовувати метод прогнозу і корекції, обчислюють початкові дані за допомогою будь-якого однокрокового методу. Часто для цього використовують метод Рунге – Кутта.

Обчислення виконують таким чином. Спочатку за формулою прогнозу та початковим значенням змінних знаходять значення $y_{n+1}^{(0)}$. Індекс (0) означає, що значення, яке прогнозується, є одним із послідовності значень y_{n+1} по мірі їх уточнення. За значенням $y_{n+1}^{(0)}$ за допомогою початкового диференціального рівняння (4.1.) знаходять похідну $y_{n+1}^{(0)'} = f\left(x_{n+1}, y_{n+1}^{(0)}\right)$, яка після цього підставляється у формулу корекції для обчислення уточненого значення $y_{n+1}^{(j+1)}$. В свою чергу, за $y_{n+1}^{(j+1)}$ знаходять похідну $y_{n+1}^{(j+1)'} = f\left(x_{n+1}, y_{n+1}^{(j+1)}\right)$. Якщо це значення не достатньо близьке до попереднього, то воно вводиться у формулу корекції і ітераційний процес продовжується. У випадку близькості значень похідних визначається y_{n+1}, яке і є остаточним. Після цього процес повторюється на наступному кроці, на якому обчислюється y_{n+2}.

Зазвичай при виведенні формул прогнозу і корекції розв'язок рівняння розглядають як процес наближеного інтегрування, а самі формули отримують за допомогою методів чисельного інтегрування.

Якщо диференціальне рівняння $y' = f(x, y)$ проінтегрувати в інтервалі значень від xn до xn+k , то результат матиме вигляд

$$y(x_{n+k}) - y(x_n) = \int_{x_n}^{x_{n+k}} f(x, y)dx$$

.

Цей інтеграл не можна обчислити безпосередньо, тому що y(x) – невідома функція. Вибір методу наближеного інтегрування і буде визначати метод розв'язання диференціальних рівнянь. На етапі прогнозу можна використовувати будь-яку формулу чисельного інтегрування, якщо до неї не входить попереднє значення $y'(x_{n+1})$.

В таблицю 4.1 зведені найбільш розповсюджені формули прогнозу і корекції. Для більшості методів прогнозу і корекції оцінюють похибку, користуючись таким співвідношенням:

$$\Delta \le \frac{1}{5}\left[y_n^{(0)} - y_n^{(j)}\right]$$

Мірою похибки слугує і є $y_n^{(j)}$, що входить до алгоритму рисунку 4.3.

Часто в довідниках приводяться більш точні формули для оцінки похибки багатокрокових методів.

При виборі величини кроку можна скористатися умовою:

$$h < \frac{2}{M_2},$$

де $M_2 = \left|\frac{\partial f}{\partial y}\right|_{max}$.

Виконання цієї умови необхідно для збіжності ітераційного процесу відшукання розв'язку.

Однак у багатьох практичних випадках складність оцінки величини M_2 приводить до того, що найбільш зручним для вибору кроку є спосіб, побудований на оцінці D у процесі обчислень

і зменшенні кроку, якщо похибка надто велика. При цьому необхідно враховувати, що оптимальне число ітерацій дорівнює двом.

Задача Коші

Задача Коші формулюється так:

Нехай задане ДР

$$\frac{dy}{dx} = f(x, y) \tag{5}$$

з початковими умовами $y(x_o) = y_0$. Потрібно знайти функцію $y(x)$, що задовольняє дане рівняння, та початкову умову. Для одержання чисельний розв'язку цієї задачі спочатку обчислюють значення похідної, а потім задаючи малий приріст $"x"$, переходять до нової точки $x_1 = x_0 + h$

Положення нової точки визначають за нахилом кривої, обчисленому з допомогою ДР. Таким чином, графік чисельного розв'язку являє собою послідовність коротких прямолінійних відрізків, якими апроксимується істинна крива $y(x)$. Сам чисельний метод визначає порядок дій при переході від даної точки кривої до наступної.

Існують дві групи методів розв'язування задачі Коші.

1. Однокрокові методи. В них для знаходження наступної точки на кривій $y(x)$ потрібна інформація лише про попередній крок. (Однокроковими є метод Ейлера та методи Руте-Кутта.)
2. Багатокрокові (або методи прогнозування та коригування).

Для знаходження наступної точки кривої $y(x)$ вимагається інформація більш ніж про одну з попередніх точок. До них належать методи Адамса, Мілна, Хеммінга.

Це чисельні методи розв'язування ДР. Вони дають розв'язок у вигляді таблиці значень.

Рекомендована література

Базова

1. Н.Б.Шаховська, Р.О.Голощук Алгоритми і структури даних. Навчальний посібник – «Магнолія 2006», Львів – 2014., -215 с.
2. **Боглаев Ю.П.** Вычислительная математика и программирование. -М.: Высшая школа, 1990. -245 с.
3. **Вирт Н.** Алгоритмы и структуры данных. - М: Мир, 1989. -360с.
4. **ВиртН.** Алгоритмы + структуры даннных = программы. - М., Мир, 1985.-308 с.
5. **Вирт Н.** Системное программирование: Введение. - М., Мир, 1977. - 403 с.
6. Чисельні методи в інформатиці : підруч. для студ. вищ. навч. закл. / Л. П. Фельдман, А. І. Петренко, О. А. Дмитрієва . - К. : BHV, 2006. - 480 с. : іл.
7. Методи обчислень : конспект лекцій для студентів механіко-математичного факультету / В. В. Попов. – К.: Видавничо-поліграфічний центр "Київський університет", 2012. – 303 с.
8. Ахо А., Хопкрофт Дж., Ульман Дж. Структуры данных и алгоритмы – М.: Изд. Дом «Вильямс», 2001. – 384с.
9. Ахо А., Хопкрофт Дж., Ульман Дж. Построение и анализ вычислительных алгоритмов. – М.: «Мир», 1979. - 536с.
10. **Д. Кнут.** Искусство программирования, т. I. Основные алгоритмы, 3-є изд. - М.: “Вильяме”, 2000. - 328 с.
11. **Д Кнут.** Искусство программирования, т.2. Получисленные алгоритмы, 3-є изд. - М.: “Вильяме”, 2000. - 390 с.
12. Д **Кнут.** Искусство программирования, т.З. Сортировка и поиск, 2-е изд. - М.: “Вильяме”, 2000. - 367 с.

Допоміжна

1. Браунси К. Основные концепции структур данных и реализация в С++. – М.: Изд. Дом «Вильямс», 2002. – 320с.
2. Проценко В.С. Техніка програмування мовою Сі: Навчальний посібник – К.: Либідь, 1993. – 224 с.
3. Гудман С. Хидетниеми С. Введение в разработку и анализ алгоритмов. - М.: "Мир", 1981. - 366 с.
4. Кнут Д. Искусство программирования для ЭВМ. т.3. Сортировка и поиск. М.:Мир, 1976. - 678 с.
5. Мейер Б., Бодуэн К. Методы программирования: В 2-х томах – М.: Мир, 1982. – 356+368с.

Інформаційні ресурси

На відповідних сайтах Internet по запиту в пошукових програмах.

Змістовий модуль 1. Базові поняття теорії алгоритмів. Поняття структури. Структурні та лінійні типи даних.

Тема 1. Визначення алгоритму, Способи описання та властивості, класи алгоритмів.

1.1. ВИЗНАЧЕННЯ АЛГОРИТМУ

За визначенням А.П.Єршова, інформатика - це наука про методи подання, накопичення, передавання та опрацювання інформації за допомогою комп'ютера. Що таке інформація? Вважається, що інформація - це поняття, яке передбачає наявність матеріального носія інформації, джерела і передавача інформації, приймача і каналу зв'язку між джерелом і приймачем інформації.

Основними в загальній інформатиці є три поняття: задача, алгоритм, програма. Відповідно, маємо три етапи в розв'язуванні задач (зазначимо, що, з точки зору інформатики, розв'язати задачу - це отримати програму, тобто, забезпечити можливість отримати рішення за допомогою комп'ютера): постановка задачі, побудова і обгрунтування алгоритму, складання і налагодження програми. Оскільки програма - об'єкт гранично формальний, а тому точний (можливо не завжди прозорий, навантажений неістотними із змістовної точки зору деталями, але недвозначний) то, пов'язані з нею об'єкти також мають бути точними. Алгоритм містить чіткий і ясний спосіб побудови результатів за точно вказаною в постановці задачі залежністю їх від наявних аргументів.

Відповідно до етапів маємо три групи засобів інформатики: специфікація, алгоритмізація і програмування.

Для специфікації задач в курсі застосовані засоби типу рекурентних співвідношень, рекурсивних визначень, а також прості інваріантні співвідношення, початкові й кінцеві умови.

Побудова алгоритму за точною постановкою задачі дає можливість його обгрунтування математичними методами. Більше того, існують класи задач, які дозволяють формальне перетворення специфікації в алгоритм. Вивченню деяких таких класів і відповідних методів відводиться важливе місце в нашому курсі.

Істотно, що, порівняно з програмою, алгоритм може перебувати на вищому рівні абстракції, бути вільним від тих або інших деталей реалізації, пов'язаних з особливостями мови програмування та конкретної обчислювальної системи. Засоби, прийняті для зображення алгоритмів, за традицією називають алгоритмічною мовою. До речі, так називалися також перші мови програмування високого рівня, наприклад, Алгол - це просто скорочення ALGOrithmic Language - алгоритмічна мова. Але, загалом, жодна мова програмування не може цілком замінити алгоритмічну мову, оскільки консервативна

Повинні існувати гарантії, що всі програми, складені вчора, в минулому році, десять років тому, не втратять значення ні сьогодні, ні завтра. Модифікація мови програмування призводить до небажаних наслідків: вимагає перероблення системи програмування, знецінює напрацьоване програмне забезпечення. У той же час алгоритмічна мова може створюватися спеціально для певної предметної області, певного класу задач або навіть окремої задачі. Вона може розвиватися навіть при створенні алгоритму, вбираючи в себе новітні результати.

Алгоритм - точне формальне розпорядження, яке однозначно трактує зміст і послідовність операцій, що переводять задану сукупність початкових даних в шуканий результат, або можна також сказати, що алгоритм - це кінцева послідовність загальнозрозумілих розпоряджень, формальне виконання яких дозволяє за скінченний час отримати рішення деякої задачі або будь-якої задачі з деякого класу задач.

Алгоритм — не скінченна послідовність команд, які треба викопати над вхідними даними для отримання результату.

Приклад 1.1. Обчислити *(x+y)/(a-b)*

$A=(A1,A2, A_3)$

A_1: *x+y*

A_2: *a-b*

A_3: A_1 / A_2

Слово алгоритм походить від algorithmi - латинської форми написання імені великого математика IX ст. Аль-Хорезмі, який сформулював правила виконання арифметичних дій.

Приклад 1.2. Розглянемо відому задачу про людину з човном (Л), вовком (В), козою (Кз) і капустою (Кп). Алгоритм її розв'язання можна подати так:

{ Л, В, Кз, Кп —> } - початковий стан,
пливуть Л, Кз, Кп
{ В —> Л, Кз, Кп } - 1-ий крок,
пливуть Л, Кз
{ Л, В, Кз —> Кп } - 2-ий крок,
пливуть Л, В
{ Кз —> Л, В, Кп } - 3-ій крок,
пливуть Л
{ Л, Кз —> В, Кп } - 4-ий крок,
пливуть Л, Кз
{—>Л, В, Кз, Кп } - кінцевий стан

1.2 СПОСОБИ ОПИСАННЯ АЛГОРИТМІВ

Існують такі способи описання алгоритмів:

- *словесний,*
- *формульний,*
- *графічний,*
- *алгоритмічною мовою.*

Перший спосіб *— це словесний опис алгоритму. Словесний опис потребує подальшої формалізації.*

Другий спосіб *- це подавання алгоритму у вигляді таблиць, формул, схем, малюнків тощо. Він є найбільш формалізованим та дозволяє описати алгоритм за допомогою системи умовних позначень.*

Третій спосіб *— запис алгоритмів за допомогою блок-схеми. Цей метод був запропонований в інформатиці для наочності подання алгоритму за допомогою набору* спеціальних блоків. Основні з цих блоків подані на рис. 1.1.

Четвертий спосіб - це мови програмування. Справа в тому, що найчастіше в практиці виконавцем створеного людиною алгоритму являється машина і тому він має бути написаний мовою, зрозумілою для комп 'ютера, тобто мовою програмування.

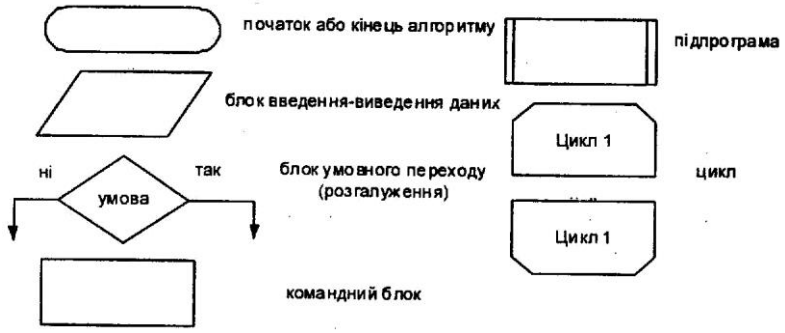


Рис. 1.1. Блоки для подання блок-схем.

1.2. ВЛАСТИВОСТІ АЛГОРИТМІВ

Розглянемо такі властивості алгоритмів: *визначеність, скінченність, результативність, правильність, формальність, масовість.*

Визначеність алгоритму. Алгоритм визначений, якщо він складається з допустимих команд виконавця, які можна виконати для деяких вхідних даних.

Приклад 1.3. Невизначеність виникне в алгоритмі (x+y)/(a-b), якщо в знамен-иику буце записано, наприклад, 92/92 (ділення на нуль неприпустиме).

Невизначеність виникне, якщо деяка команда буде записана не-правильно, бо така команда не належатиме до набору допустимих команд виконавця, (x+y)/(a-b).

Скінченність алгоритму. Алгоритм повинен бути скінченним - послідовність команд, які треба виконати, мусить бути скінченною. Кожна команда починає виконуватися після закінчення виконання попередньої. Цю властивість ще називають **дискретністю** алгоритму.

Приклад 1.4. Алгоритм (x+y)/(a-b) — скінченний. Він складається з трьох дій. Кожна дія, у свою чергу, реалізується скінченною кількістю елементар-них арифметичних операцій. Нескінченну кількість дій передбачає математичне правило перетворення деяких звичайних дробів, таких як 5/3, у нескінченні десяткові дроби.

Результативність алгоритму. Алгоритм результативний, якщо він дає результати, які можуть виявитися і невірнимим.

Наведені вище алгоритми є результативними. Прикладом нерезультативного алгоритму буде алгоритм для виконання обчислень, в якому пропущена команда виведення результатів на екран тощо.

Правильність алгоритму. Алгоритм правильний, якщо його виконання забезпечує досягнення мети.

Приклад 1.5. Наведені вище алгоритми є правильними. Помінявши місцями в алгоритмі з прикладу 1.2 будь-які дві команди, отримаємо неправильний алгоритм.

Формальність алгоритму. Алгоритм формальний, якщо його можуть виконати не один, а декілька виконавців з однаковими результатами. Ця властивість означає, що коли алгоритм *A* застосовують до двох однакових наборів вхідних даних, то й результати мають бути однакові.

Приклад 1.6. Наведені алгоритми задовольняють цю умову, їх можуть виконати багато виконавців.

Масовість алгоритму. Алгоритм масовий, якщо він придатний для розв'язування не однієї задачі, а задач певного класу.

Приклад 1.6. Алгоритм з прикладу 1.1 не є масовим. Алгоритм Маляр є масовим, оскільки може застосовуватись не тільки для зафарбовування якихось елементів, але й для певних задач на графах. Прикладами масових алгоритмів є загальні правила, якими користуються для множення, додавання, ділення двох багатозначних чисел, бо вони застосовні для будь-яких пар чисел. Масовими с алгоритми розв'язування математичних задач, описаних у загальному вигляді за допомогою формул, їх можна виконати для різних вхідних даних.

1.7. КЛАСИ АЛГОРИТМІВ

Основною оцінкою функції складності алгоритму f(n) є оцінка Θ .

Кажуть, що $f(n) = \Theta$ g(n)), якщо при$g > 0$ при $n > 0$ існують додатні c_1, c_2, n_0, такі, що

$$c_1 g(n) \le f(n) \le c_2 g(n)$$

при $n > n_0$. Тобто, можна знайти такі c_1, та c_2, що при достатньо великих n функція знаходитиметься між $c_1 g(n)$ та $c_2 g(n)$.

У такому випадку функція $g(n)$ є асимптотично точною оцінкою функції $f(n)$, оскільки за визначенням функція $f(n)$ не відрізняється від функціїg $g(n)$ з точністю до постійного множника.

Виділяють такі основні класи алгоритмів:

- ✓ *логарифмічні*: $f(n) = \Theta (\log_2 n)$;
- ✓ *лінійні* $f(n) = \Theta$ (n); якщо $n = 1$, то отримуємо константі алгоритми;
- ✓ *поліноміальні:* $f(n) = \Theta (n^m)$; тут m - натуральне число, більше від одиниці; при $m = 1$ алгоритм є лінійним;
- ✓ *експоненційні*: $f(n) = \Theta (a^n)$; a - натуральне число, більше від одиниці. Експоненційні алгоритми часто пов'язані з перебором різних варіантів розв'язку.

Для однієї й тієї ж задачі можуть існувати алгоритми різної складності. Часто буває і так, що повільніший алгоритм працює завжди, а швидший - лише за певних умов.

Будемо називати **часовою складністю задачі** часову складність найефективнішого алгортітму для її розв'язання.

Алгоритми без циклів і рекурсивних викликів мають константну складність. Якщо немає рекурсії та циклів, всі керуючі структури можуть бути зведені до структур константної складності. Отже, і весь алгоритм також характеризується константною складністю.

Визначення складності алгоритму в основному зводиться до аналізу циклів і рекурсивних викликів.

Приклад 1.7. Розглянемо алгоритм опрацювання елементів масиву. Нехай таким опрацюванням буде пошук заданого елемента.

```
For i=1 to N do
Begin
If a[i]=k then
…
End;
```

Складність цього алгоритму (N), оскільки тіло циклу виконується N разів, і складність тіла циклу рівна (1).

Якщо один цикл вкладений у інший і обидва цикли залежать від величини однієї і тієї ж змінної, то вся конструкція характеризується квадратичною складністю.

```
For i:=1 to N do
For j:=1 toN do
Begin
…
End;
```

Складність цієї програми (N^2).

Але, якщо заздалегідь відомо, що послідовність упорядкована за зростанням або за спаданням, можна застосувати інший алгоритм - алгоритм половинного ділення. Послідовність ділиться на дві рівні частини. Оскільки послідовність упорядкована, можна визначити, в якій частині міститься потрібний елемент. Після цього процедура повторюється: потрібна частина знову ділиться навпіл і т.д. Цей алгоритм є логарифмічним.

Дамо тепер визначення **складності класів задач P і NP. Клас P** складається 3S задач, для яких існують поліноміальні алгоритми розв'язання. ***Клас NP*** складають задачі, для яких існують поліноміальні алгоритми перевірки правильності рішення (точніше, якщо є розв'язок задачі, то існує деяка підказка, яка дозволяє за поліноміальний час отримати цю відповідь). Неформально кажучи, клас P складається із задач, які можна швидко розв'язати, а класс NP - зі задач, розв'язок яких можна швидко перевірити.

Наведемо приклад задачі класу P. Треба визначити, чи є у масиві дійсних чисел A[1..n] елемент зі значенням не меншим, ніж *k.* Очевидний у цьому випадку алгоритм перебирає всі елементи масиву за час (*n*).

Прикладом задачі класу NP є задача комівояжера (є множина міст та відділей між ними, мандрівний торговець мас відвідати усі міста, не заходячи у жодне двічі, з мінімальними витратами на дорору. Дійсно, якщо задано деякий маршрут завдовжки не більше ніж *k,* то за час*(n)* можна перевірити, що він дійсно має саме таку довжину, і тим самим переконатися у його існуванні. Для цього треба перебрати всі *n* переходів між містами, що містяться у маршруті, і додати їх довжини.

Очевидним є також включення P NP (для перевірки розв'язання задачі класу P досить розв'язати її поліноміальним алгоритмом).

Задача називається ***NP-повною,*** якщо вона належить класу NP і до неї за поліноміальний час можна звести будь-яку іншу задачу цього класу. Якщо якась NP- повна задача має поліноміальний алгоритм розв'язання, то всі NP-повні задачі можуть бути поліноміально розв'язані і, як наслідок, P=NP.

NP-повні задачі є найважчими у класі NP.

Експоненційні алгоритми та перебір

Експоненційні алгоритми часто пов'язані з перебором різних варіантів розв'язання.

Наведемо типовий приклад.

Приклад 1.8. Розглянемо задачу про виконуваність булевого виразу, яка формулюється так: **для** будь-якого булевого виразу від **/7** змінних знайти хоча б один набір значень змінних х,... x_n, при якому цей вираз приймає значення 1.

Типова схема розв'язування цієї задачі може мати такий вигляд: спочатку надаємо одне з двох можливих значень (0 або 1) першій змінній х,, потім другій і т.ін. Коли будуть розставлені значення всіх змінних, ми можемо визначити значення булевого виразу. Якщо він дорівнює 1, задача розв'язана. Якщо ні - треба повернутися назад і змінити значення деяких змінних.

Можна інтерпретувати цю задачу як задачу пошуку на певному дереві перебору. Кожна вершина цього дерева відповідає певному набору встановлених значень х ..., $x_к$. Ми можемо встановити змінну $x_ж$ в 0 або 1, тобто маємо вибір з двох дій. Тоді кожній із цих дій відповідає одна з двох дуг, які йдуть від цієї вершини. Вершина х, ..., $x_к$ має двох синів х, ..., $x_к$ 0 і х, ..., $x_к$ 1.

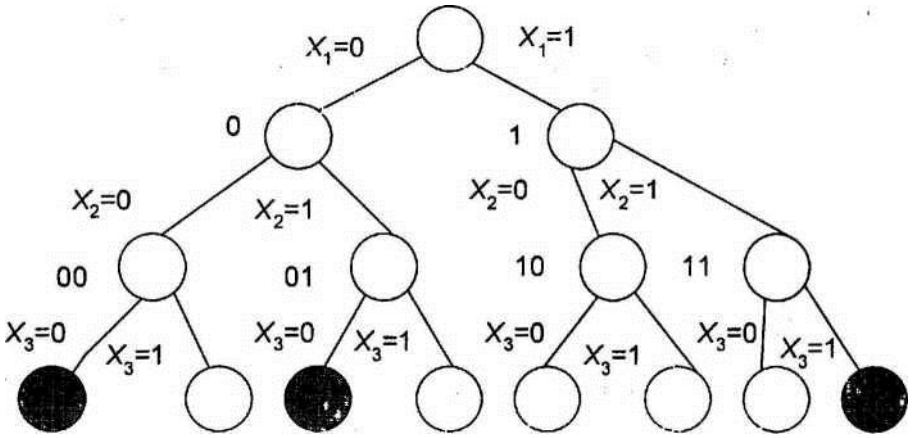


Рис 1.2. Пошук на дереві

Звертаємо увагу на те, що не кожен перевірний алгоритм є експоненційним. (Наприклад, алгоритм пошуку в масиві. Незважаючи на його перевірний характер, він є лінійним, а не експоненційним).

Зі зростанням розмірності будь-який поліпоміальний алгоритм стає ефективнішим, ніж будь-який експоИенційний. Дія лінійного алгоритму зросгання швидкодії комп'ютера в 10 разів дозволяє за той самий час розв'язати задачу, розмір якої в 10 разів більший. Для експоненційного алгоритму з основою 2 цей самий розмір можна збільшити лише на 3 одиниці.

Як правило, якщо для розв'язування якоїсь задачі є деякий пшіноміальний алгоритм, то часова оцінка цього алгоритму значно покращується.

Алгоритм із поверненнями назад

Метод перебору із поверненнями дозволяє розв'язувати практично незліченну множину задач, для багатьох з яких не відомі інші алгоритми. Незважаючи на таке велике різноманіття перебірних задач, в основі їх розв'язування є щось спільне, що дозволяє застосувати цей метод. Таким чином, перебір можна вважати практично універсальним методом розв'язування перебірних завдань. Наведемо загальну схему цього методу.

Розв'язування задачі методом перебору з поверненням будується конструктивно послідовним розширенням часткового розв'язування. Якщо на конкретному кроці таке розширення провести не вдається, то відбувається повернення до коротшого часткового розв'язування, і спроби його розширити продовжуються.

```
{пошук одного вирішення}
    procedure backtracking(k: integer); {k - номер ходу}
      begin
```

```
            {запис варіанту}
             if {рішення знайдене} then
                    {виведення рішення}
             else
             {перебір всіх варіантів }
 if { варіант задовольняє умови задачі} then
              backtracking(k+1); { рекурсивний виклик}
                 {стирання варіанту}
           end
begin
backtracking(l);
end.
```

Тема 2 Структура даних «масив», «множина», «таблиця», «стек», «черга».

3.1. ПОНЯТТЯ СТРУКТУРИ ДАНИХ ТИПУ «МАСИВ»

♦ Масив - послідовність елементів одного типу, який називається базовим. Математичною мовою масив - це функція з обмеженою областю визначення. Структура масивів однорідна. Для виділення окремого компонента масиву використається індекс. Індекс — це значення спеціального типу, визначеного як тип індексу певного масиву. Тому на логічному рівні СД типу «масив» можна записати так: type A = array [T1] of T2, де Л - базовий тип масиву, 72 - тип індексу.

Якщо D_n - множина значень елементів типу 71, D_n - множина значень елементів типу 72, то A: $D_{T|}$ ® D_n (відображення).

Кардинальне число Car(T) структури типу 7 - це множина значень, які може приймати задана структура типу Т. Кардинальне число характеризує об'єм пам'яті, необхідний такій структурі.

Для масиву А: Car (A) = [Car(72)] Car(71).

Масив може бути одновимірнгіч (вектором), та багатовимірним (наприклад, двовимірною таблицею), тобто таким, де індексом є не одне число, а кортеж (сукупність) із декількох чисел, кількість яких співпадає з розмірністю масиву.

У переважній більшості мов програмування масив є стандартною вбудованою структурою даних.

Отже, з вищенаведеного сформулюємо такі властивості масиву:

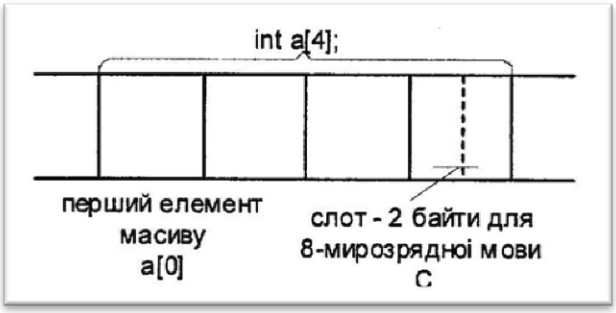
- ✓ *усі елементи масиву мають той самий тин;*
- ✓ *кожний компонент має свій номер у послідовності (індекс) і відрізняється ним від інших елементів (ідентифікується);*
- ✓ *множина індексів (індексова множина) скінченна й зафіксована в означенні масиву і ід час виконання програми не змінюється;*
- ✓ *можливість опрацювання компонента, або його доступність, не залежить від його місця в послідовності (елементи рівнодоступні).*

3.2. НАБІР ДОПУСТИМИХ ОПЕРАЦІЙ ДЛЯ СД ТИПУ «МАСИВ»

Над масивом можна виконувати такі операції:

- Операція доступу (доступ до елементів масиву - прямий; від розміру структури операція не залежить).
- Операція присвоювання.
- Операція ініціалізації (визначення початкових умов).

На фізичному рівні СД типу «масив» є неперервною ділянкою пам'яті елементів однакового об'єму. Ділянка пам'яті, необхідна для одного елемента, називається ***слотом.***



Var B: A {визначаємо змінну *B* як змінну типу «масив *A»}*;

p <i <g, де ***p*** - індекс першого елемента масиву, ***g*** - індекс останнього елемента масиву, *i* - індекс елемента.

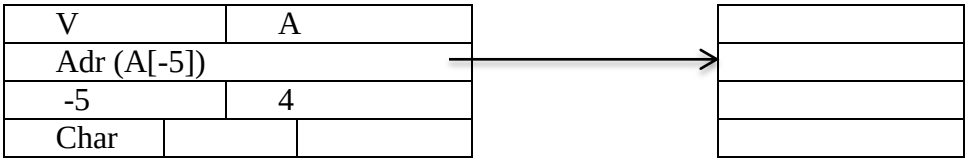
3.3. ДЕСКРИПТОР СД ТИПУ «МАСИВ»

Нерідко фізичній структурі ставиться у відповідність дескриптор (заголовок), що ,і і /ить загальні відомості про задану фізичну структуру. Дескриптор також

зб рігається, як і структура, в пам'яті. Загалом дескриптор являє собою структуру т .пу «запис».

Стосовно до СД типу «масив», дескриптор містить такі компоненти: ім'я масиву, умовне позначення заданої структури, адресу першого елемента масиву, індекси нижньої й верхньої границь масиву, тип елемента масиву, розмір слота.

Наприклад, для наступного описування масиву: var *A:* array [-5 .. 4] of Char дескриптор буде виглядати так:



Для СД типу «масив» розмір дескриптора не залежить від розмірності масиву. При кожній операції доступу використовується вся інформація дескриптора. Наприклад, поля границі зміни індексу використовуються при обробці виняткових операцій.

3.4. ЕФЕКТИВНІСТЬ МАСИВІВ

Масиви ефективні при звертанні до довільного елемента, яке відбувається за постійний час (0(1)), однак такі операції як додавання та видалення елемента, потребують часу 0 (н), де п - розмір масиву. Тому масиви переважно використовуються для зберігання даних, до елементів яких відбувається довільний доступ без додавання або видалення нових елементів, тоді як для алгоритмів з інтенсивними операціями додавання та видалення, ефективнішими є зв'язані списки.

Інша перевага масивів, яка є досить важливою - це можливість компактного збереження послідовності їх елементів в локальній області пам'яті (що не завжди вдається, наприклад, для зв'язаних списків), що дозволяє ефективно виконувати операції з послідовного обходу елементів таких масивів.

Масиви є дуже економною щодо пам'яті структурою даних. Для збереження 100 цілих чисел у масиві треба рівно у 100 разів більше пам'яті, ніж для збереження одного числа (плюс, можливо, ще декілька байтів). У той же час, усі структури даних, які базуються на вказівниках, потребують додаткової пам'яті для збереження самих вказівників разом із даними. Однак, операції з фіксованими масивами ускладнюються тоді, коли виникає необхідність додавання нових елементів у вже заповнений масив. Тоді його слід розширювати, що не завжди можливо і для таких задач слід використовувати зв'язані списки, або динамічні масиви.

У випадках, коли розмір масиву є досить великий і використання звичайного звертання за індексом стає проблематичним, або великий відсоток його комірок не використовується, треба звертатися до асоціативних масивів, де проблема індексування великих об'ємів інформації вирішується оптимальніше. В асоціативному масиві замість числових індексів

використовуються ключі будь-яких типів. Дані в асоціативному масиві так само можуть бути різнотипними. Така структура також відома як «хеш» або як «словник». Це лише відображення «назва-значепня», як показано нижче:

```
h = {1 =>2, «2» => «4»}
print hash,»\n»
print hash[1],»\n»
print hash[«2»],»\n»
print hash[5],»\n»
```

З тої причини, що масиви мають фіксовану довжину, треба дуже обережно ставитися до процедури звертання до елементів за їхнім індексом, тому що намагання звернутися до елемента, індекс якого перевищує розмір такого масиву (наприклад, до

3.6. СД ТИПУ «МНОЖИНА»

❖ Множина — *скінчений набір елементів одного типу, для яких не важливий порядок слідування і жоден з елементів не може бути два рази включений. Така СД визначається конструкцією type T = set of T0, де T0 - вбудований або раніше визначений тип даних (базовий тип). Значеннями змінних типу T є множини елементів типу T0 (зокрема, порожні множини).*

Кардинальне число множини (потужність) рівне кількості її елементів.

Набір допустимих операцій для СД типу «множина»: «*» - перетин множин, «+» - об'єднання множин, «-»-різниця множин, «in» - перевірка належності до множини елемента базового типу.

Дескриптор СД типу «множина» не відрізняється від дескриптора СД типу «масив». Подамо програму, яка виводить усі цифри, що не входять у десятковий запис числа.

Для цього скористаємося множиною s, що міститиме усі цифри.

```
Vars:set of 0..9;
   n,ost,i:integer; {n - число, яке треба проаналізувати}
begin write('lnput number');
     readln(n);
     s:=[0,1 ,2,3,4,5,6,7,8,9]; {включили у множину всі цифри}
     while n>0 do
      {виділяємо цифри числа методом ділення його на 10}
      begin
        {визначаємо остачу від ділення}
         ost:=n mod 10;
         n:=n div 10;
         if (ost in s) then s:=s-[ost] {здійснюємо операцію різниці}
      end;
{виведемо всі цифри, що не належать до запису числа, за зростанням}
    for i:=0 to 9 do
     if i in s then write (i,',')
    end.
```

Найчастіше множини використовуються для формування набору елементів, що зустрічаються у масивах лише один раз.

3.8. СД ТИПУ «ТАБЛИЦЯ»

❖ Таблиця — *послідовність записів, які мають ту саму організацію. Такий окремий запис називається* елементом таблиці. *Найчастіше використовується простий запис. Отже, таблиця - це агрегація елементів. Якщо послідовність записів впорядкована щодо певної ознаки, то така таблиця називається* впорядкованою, *інакше - таблиця* невпорядкована.

Класифікацію СД типу таблиця подано на рис. 3.1.

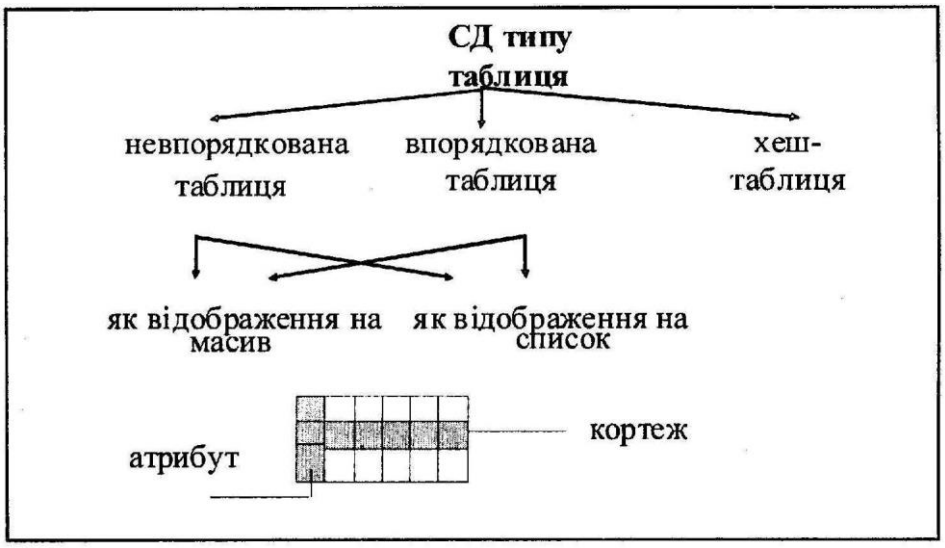


Рис. Класифікація СД типу «таблиця»

Якщо один елемент *d*, то *кортеж*-це $<d_v\ d_v\ ....\ d>$, причому DT_t O *d.* Множина значень елементів типу *T* (множина допустимих значень СД типу «таблиця») буде визначатися за допомогою прямого декартового добутку:

$DT = DT_1$ **r** DT_2 r ... DT_n, *причому* DT_i $<d_1, d_2, d_n>$...

Сам елемент таблиці можна подати у вигляді двійки <*K,V*>, де *K* - ключ, а *V*— тіло елемента. Ключем може бути різна кількість полів, які визначають цей елемент. Ключ використовується для операції доступу до елемента, тому що кожний із ключів унікальний для заданого елемента. Отже, таблиця є сукупністю двійок <*K, V*>.

На логічному рівні елемент СД типу «таблиця» описуванняється так (приклад на мові Паскаль):

```
Type Element = record Key: integer;
{опис інших полів} end;
```

При реалізації таблиці як відображення на масив її опис виглядає так:

```
Tabl = array [0 .. N] of Element.
```

Під час виконання програми кількість елементів може змінюватися. Структура, у якій змінюється кількість елементів під час виконання програми, називається *динамічною.* Якщо розглядати динамічну структуру як відображення на масив, то така структура називається *напівапатичною.*

Перед тим як визначити операції, які можна виконувати над таблицею, розглянемо **класифікацію операцій.**

Конструктори — операції, які створюють об'єкти розглянутої структури. *Деструктори* - операції, які руйнують об'єкти розглянутої структури. Ознакою цієї операції є звільнення пам'яті.

Модифікатори - операції, які модифікують відповідні структури об'єктів. До них належать динамічні й напівстатичні модифікатори.

Спостерігачі — операції, у яких елементом (вхідним параметром) є об'єкти відповідної структури, а повертають ці операції результати іншого типу. Отже, операції- спостерігачі не змінюють структуру, а лише подають інформацію про неї.

Ітератори - оператори доступу до вмісту частини об'єкта у певному порядку. **Набір допустимих операцій для СД типу «таблиця» подаємо нижче**

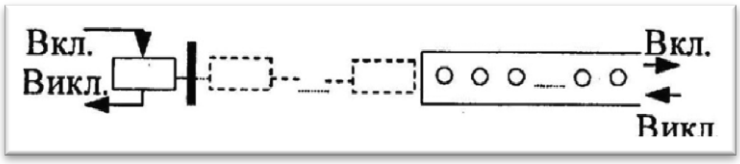
- *Операція ініціалізації (конструктор).*
- *Операція включення елемента в таблицю (модифікатор).*
- *Операція виключення елемента з таблиці (модифікатор).*

Операції-предикати:

- *таблиця порожня / таблиця не порожня (спостерігач),*
- *таблиця переповнена / таблиця не переповнена (спостерігач).*
- *читання елемента за ключем (спостерігач).*

3.9. СД ТИПУ «СТЕК»

❖ ***Стек*** *- це послідовність, у якій включення й виключення елемента здійснюється з однієї сторони послідовності (вершини стека). Так само здійснюється й операція доступу. Структура функціонує за принципом LIFO (останній, що прийшов, обслуговується першим). Умовні позначення стека зображені на рис*



а) відображеня на масив б) відображення на список

Рис. 3.2. Організація стека.

При реалізації стека розглядаються стек як відображення на масив і стек як відображення на список.

Відображення на масив передбачає оголошення звичайного масиву та змінної, значення якої дорівнюватиме значенню індексу елемента, що відіграватиме роль «голови» (елемента, на який вказуватиме вказівник):

```
int a[100]; // оголошення стеку
    int n=0; // дійсна кількість елементів у стека
    int current=99; // індекс «голови», діє принцип LIFO
    void pop () // функція видобування (вилучення) елемента зі стека
    {
        if (n!=0)
        {
              current++; // зсунули «голову» на один елемент вперед
            printf('‘%d%, a[current]);
            n—;          // зменшили кількість елементів
        }
    }
    void push 0 // функція додавання елемента до стеку
    {
        if (п<99)
        {
            current—;
            //додали «голову», зсунувши індекс на один елемент вперед
```

```
            scanf("%d%, &a[current]);
            n++;            // збільшили кількість елементів
        }
    }
```

Відображення на список передбачає оголошення динамічної структури. Перевагою такого відображення є відсутність обмежень на максимальну кількість елементів у стеку, але, водночас, передбачає підтримку складнішої вказівникової структури:

```
struct stack {
            int el;         // значення елемента
    struct stack *next;} st // адреса наступного елемента st 'head, *p1, *p2; //вказівник на
 «голову», допоміжні вказівники st* push(int a; st *cur)
    // функція додавання елемента,
    //a - значення, яке треба внести у стек //cur-вершина («голова») стека {st *p;
        // якщо стек порожній if(!cur)
        {
            //          створюємо          вершину
          cur=(st*)malloc(sizeof(st));
            cur->el=a;
            cur->next=NULL;
            return (cur);
        }
        else
        {
            // створюємо новий елемент, який стане вершиною стека
            p=(st*)malloc(sizeof(st));
            p->ei=a;
            p->next=cur;
            return (p);
        }
    }
    st* pop()          // видалення вершини стека
    {st *p;
        if(head) // якщо в стеку є елементи
         { .// вершиною стає наступний елемент
              p=head->next;
              //знищуємо «голову»
              free(head);
              return (p);
         }
     else return (NULL);
     }
```

Сукупність операцій, що визначають структуру типу «стек», подана нижче.

- ✓ *Операція ініціалізації.*
- ✓ *Операція включення елемента в стек.*
- ✓ *Операція виключення елемента зі стека.*
- ✓ *Операція перевірки: стек порожній / стек не порожній.*
- ✓ *Операція перевірки: стек переповнений / стек не переповнений (ця операція характерна для стека як відображення на масив).*
- ✓ *Операція читання елемента (доступ до елемента).*

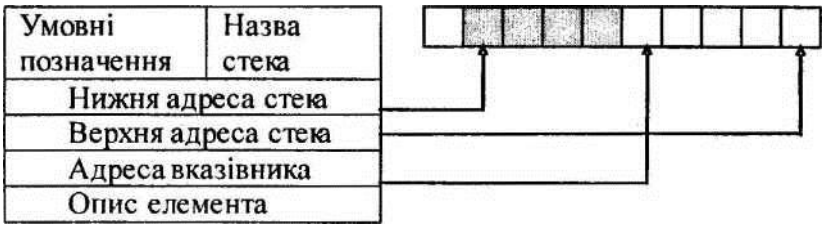
Є дві модифікації стека:

- ➢ *вказівник перебуває на вершині стека, показуючи на перший порожній елемент;*
- ➢ *слот;*
- ➢ *вказівник вказує на перший заповнений елемент.*

3.9.1. Дескриптор СД типу «стек»

Дескриптор СД типу «стек» містить:

- *адреси початку та кінця стека,*
- *адресу вказівника,*
- опис елементів



3.9.2. Області застосування СД типу «стек»

Стек використовується при перетворенні рекурсивних алгоритмів у нерекурсивні. Зокрема, за допомогою стека можна модифікувати алгоритм сортування Хоора.

Стек використовується при розробленні компіляторів.

Стеки вплинули й на архітектуру комп'ютера, послужили основою для стекових машин. У такого комп'ютера акумулятор виконаний у вигляді стека, що дозволяє розширити спектр безадресних команд, тобто команд, що не вимагають явного задання адрес операндів. Наслідком використання стека є збільшення швидкості опрацювання.

СД ТИПУ «ЧЕРГА»

❖ **Черга** - послідовність, у яку включають елементи **з** одного боку, а виключають - з іншого. Структура функціонує за принципом FIFO (надійшовший першим, обслуговується першим). Умовне позначення черги подане на рис 3.3.

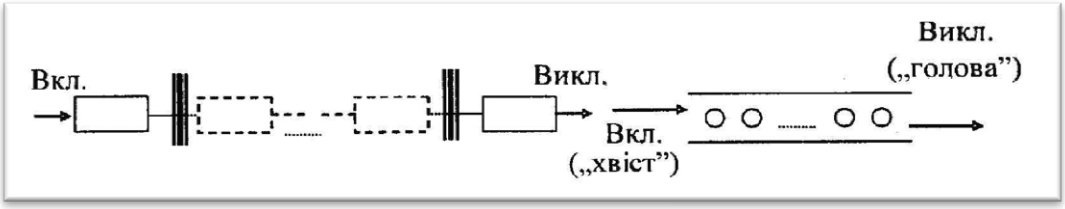


Рис. СД типу «черга».

При реалізації черги розглядаються черга як відображення на масив (напівстатична реалізація) і черга як відображення на список.

Відображення на масив:

```
int a[100]; // оголошення черги
int n=0; // індекс останнього елемента у черзі
int current=0; H індекс «голови», діє принцип FIFO
void pop () // функція видобування (вилучення) елемента з черги
{
    if (n!=0)
    {
        current++; // зсунули «голову» на один елемент вперед printf("%d%, a[current]);
    }
}
void push () // функція додавання елемента до черги
{
    if (leurrent) // черга порожня
```

```
        {
            scanf("%d%, &a[current]);
        }
        else if (n<99)
        {                   // збільшили кількість елементів
            n++;            // додали елемент у кінець черги
            scanf("%d%, &a[п]);
        }
    }
    Відображення на список: struct cherga {
            int el;                 //значення елемента
    struct charga *next;} ch // адреса наступного елемента ch 'head, *p1, *p2;      //вказівник на
 «голову», допоміжні вказівники
ch* push(int a; ch *cur)
// функція додавання елемента,
//a - значення, яке треба внести у чергу // сиг - останній елемент черги {ch *p;
// якщо черга порожня
if(!cur)
{           // створюємо вершину
cur=(ch*)mal!oc(sizeof(ch));
cur->el=a;
cur->next=NULL;
retum(cur);
}
else
{           // створюємо новий елемент та додаємо його у кінець
p=(ch*)mailoc(sizeof(ch)); p->el=a; cur->next=p; return (p);
}
}
ch* pop() // видалення вершини черги {ch *p;
if(head) // якщо в черзі є елементи
{           .// вершиною стає наступний елемент
p=head->next;
// знищуємо «голову»
free(head);
return(p);
}
else return (NULL);
}
```

Сукупність операцій, що визначають структуру типу «черга» подана нижче.

- ✓ *Операція ініціалізації.*
- ✓ *Операція включення елемента в чергу.*
- ✓ *Операція виключення елемента із черги.*
- ✓ *Операція перевірки: черга порожня / черга не порожня.*
- ✓ *Операція перевірки: черга переповнена / черга не переповнена.*

Області застосування СД типу «черга»

Черга використається при передаванні даних з оперативної у вторинну пам'ять (при цьому відбувається процедура буферизації: накопичується блок і передається у вторинну пам'ять). Наявність буфера забезпечує незалежність взаємодії процесів між виробником і споживачем (рангування задач користувача). Задачі розділяються за пріоритетами:

- *задачі, розв 'язувані в режимі реачьного часу (вищий пріоритет) (черга I);*

- *задачі, розв'язувані в режимі розділення часу (черга 2);*
- *задачі, розв 'язувані в пакетному режимі (фонові задачі) (черга 3).*

Доступ до елементів черги здійснюється послідовно.

Контрольні запитання до модуля 1.

21. *Опишіть структури даних типу «масив».*
22. *Перерахуйте набір допустимих операцій для структури даних типу «масив».*
23. *Поняття дескриптора. Приклад.*
24. *Дескриптор структури даних типу «масив».*
25. *Структури даних типу «запис» (прямий декартовий добуток).*
26. *Структури даних типу «таблиця».*
27. *Класифікація структур даних типу «таблиця».*
28. *Класифікація операцій над структурами даних типу «таблиця».*
29. *Набір допустимих операцій для структури даних типу таблиця».*
30. *Структури даних типу «стек».*
31. *Сукупність операцій, що визначають структуру типу «стек».*
32. *Дескриптор структури даних типу «стек».*
33. *Області застосування структури даних типу «стек».*
34. *Структури даних типу «черга».*
35. *Сукупність операцій, що визначають структуру типу «черга».*
36. *Дескриптор структури даних типу «черга».*
37. *Області застосування структури даних типу «черга».*
38. *Області застосування СД типу «дек».*

Тести для закріплення матеріалу

14. Перерахувати допустимі операції над масивами:

а) операція доступу;
б) операція розіменування;
в) операція присвоєння;
г) операція індексування;
д) операція ініціалізації.

15. Перерахувати дані, що містить дескриптор масиву:

а) ім'я;
б) умовне позначення;
в) адреса першого елемента;
г) адреса останнього елемента;
д) індекс першого елемента;
е) індекс останнього елемента.

16. Перерахувати операції над множинами:

а) перетин;
б) транспонування;
в) об'єднання;
г) різниця;
д) сума;
е) перевірка належності.

17. Дати визначення структур даних типу «запис»:

а) послідовність елементів, які, в загальному випадку, можуть бути одного типу;
б) послідовність елементів, які, в загальному випадку, можуть бути різного типу;
в) послідовність декількох множин елементів;
г) послідовність декількох масивів.

18. Перерахувати допустимі операції над записами:

а) операція доступу;
б) операція розіменування;
в) операція присвоєння;

г) операція індексування;
д) операція ініціалізації'.

19. Типи таблиць:

а) невпорядкована таблиця;
б) впорядкована таблиця;
в) умовно-впорядкована таблиця;
г) хеш-таблиця;
д) відображення на множину;
е) відображення на масив; є) відображення на список.

20. ***За визначенням вибрати операцію над таблицею: операція, у якої вхідним параметром є об'єкти відповідної структури, вона повертає результати іншого типу:***

а) конструктори;
б) деструктори;
в) модифікатори;
г) спостерігачі;
д) ітератори.

21. За визначенням вибрати операцію над таблицею: опера тор доступу до вмісту об'єкту частинами у певному порядку:

а) конструктори;
б) деструктори;
в) модифікатори;
г) спостерігачі;
д) ітератори.

22. За визначенням вибрати операцію над таблицею: операція, яка руйнує об'єкти розглянутої структури:

а) конструктори;
б) деструктори;
в) модифікатори;
г) спостерігачі;
д) ітератори.

23. За визначенням вибрати операцію над таблицею: операція, яка створює об'єкти розглянутої структури:

а) конструктори;
б) деструктори;
в) модифікатори;
г) спостерігачі;
д) ітератори.

24. Перерахувати допустимі операції над таблицями:

а) операція ініціалізації;
б) операція присвоєння;
в) операція включення елемента в таблицю;
г) операція виключення елемента з таблиці;
д) операції-предикати;
е) операція порівняння;
є) читання елемента за ключем.

25. Принцип LIFO діє для:

а) черги;
б) стека;
в) списку;
г) слота.

26. Принцип FIFO діє для:

а) черги;

б) стека;
в) списку;
г) слота.

Змістовий модуль 2. Зв'язаний розподіл пам'яті. Хешування даних. Нелінійні структури даних: дерева і граф.

Тема 1. Зв'язаний розподіл пам'яті. Хешування даних. Хеш-функція, алгоритми хешування, динамічне хешування.

4.1. СД ТИПУ ВКАЗІВНИК

Вказівний тип займає проміжне положення між скалярними й структурними типами: з одного боку значення вказівного типу є атомарним (неподільним), а з іншого, ці типи визначаються через інші (у тому числі й структурні) типи.

Type <тип вказівиика> = ^Л <тип об'єкту, що вказується>

(цей тип дозволяє використати базовий тип перед описом)

Type PtrType = ^BaseType;

BaseType = record

x, y: real;

end;

Var A: PtrType; {A - змінна статичного типу, значенням якої є адреси розташування в пам'яті конкретних значень заданого типу}

B: BaseType;

C: ^PtrType;

Змінній A можна присвоїти адресу якоїсь змінної, для чого використаємо унарну операцію взяття вказівника: A = B.

На фізичному рівні вказівник займає два слоти: у першому слоті перебуває адреса сегмента, у другому - адреса зсуву.

Операції над вказівним типом:

4) операція порівняння на рівність: **=** *(рівність, якщо співпадають адреси сегмента й зсуву);*

5) операція порівняння на нерівність: <>;

6) операція доступу: B.X = B.X + ***C.***

Серед всіх можливих вказівників виділяється один спеціальний вказівник, що нікуди не вказує. Тобто у пам'яті виділяється одна адреса, у яку не записується жодна

На це місце в пам'яті й посилається такий порожній або "нульовий" вказівник, що позначається **nil** на мові Паскаль або NULL на мові С. Вказівник **nil** вважається константою, сумісною з будь-яким вказівним типом, тобто це значення можна присвоювати будь-якому вказівному типу.

4.2. СТАТИЧНІ Й ДИНАМІЧНІ ЗМІННІ

4.2.1. Відмінності між статичними та динамічними змінними

Дотепер ми розглядали змінні, які розміщаються в пам'яті відповідно до цілком певних правил, а саме, для локальних змінних, описаних у підпрограмах, пам'ять приділяється при виклику підпрограми; при виході з неї ця пам'ять звільняється, а самі змінні припиняють існування. Глобальним змінним програми пам'ять приділяється на початку її виконання; ці змінні існують протягом усього періоду роботи програми. Іншими словами, розподіл пам'яті у всіх цих випадках відбувається повністю автоматично. Змінні, пам'ять під які розподіляється подібним чином, називаються **статичними.**

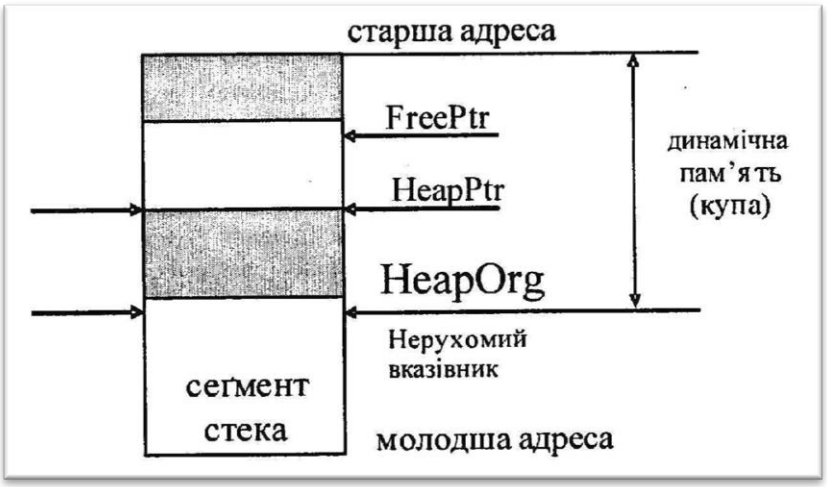


Рис. Розподіл памяті для динамічних змінних

New (<ім'я постання>): point; — процедура призначена для створення динамічних змінних певного типу (інакше кажучи, для відведення пам'яті в купі для зберігання значень динамічної змінної).

Dispose (<ім я посилання>): point; — процедура використовується для звільнення пам'яті, відведеної за допомогою процедури **New.**

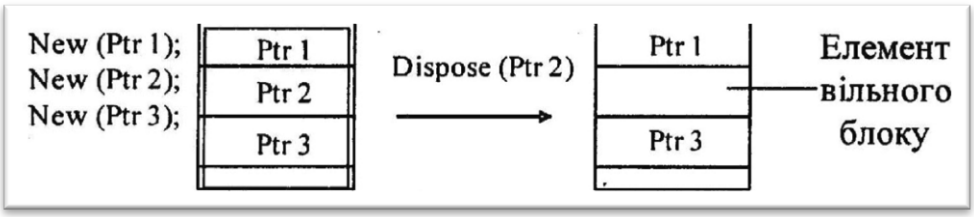
MaxAvail: longint; - функція повертає максимальний розмір (у байтах) безперервної вільної ділянки купи. Застосування цієї процедури необхідне для контролю динамічної пам'яті при реалізації операції включення.

Наприклад: if MaxAvail > SizeOf (BaseType) then {генеруємо об'єкт}

Створимо три об'єкти, які розташуються в пам'яті послідовно (без фрагментарності), а потім знищимо другий об'єкт. У результаті виникне фрагментарність, якої треба уникати.

MemAvail: longint; — функція повертає загальну кількість вільної пам'яті.

Щоб перевірити, є чи фрагментарність, треба порівняти результати застосування функцій *MaxAvail* ***і*** *MemAvail* - вони повинні збігатися.



4.3. КЛАСИФІКАЦІЯ СД ТИПУ «ЗВ'ЯЗНИЙ СПИСОК»

Усі вищерозглянуті структури, що реалізувалися як відображення на масив, мають певні недоліки, тобто вони малоефективні при розв'язуванні деяких задач. До таких недоліків можна віднести наступне.

- ✓ Точно невідомо, скільки елементів буде мати певна структура, тобто не можна зробити оцінку.
- ✓ Якщо ми розглядаємо якусь послідовність елементів у послідовній пам'яті

x_1 x_2.x_n і необхідно включити який-небудь новий елементну цю послідовність, то

ми повинні здійснити масову операцію зсуву всіх елементів, що перебувають за тим елементом послідовності, після якого ми хочемо включити новий елемент. Після цього вставимо цей новий елемент **x** на місце, що звільнилося. Отже, тут проявляється властивість фізичної суміжності

Рис. Додавання нового елемента в список

Позбутися фізичної суміжності можна, якщо елементи будуть мати не тільки дані, але й вказівники. У цьому випадку елементи можуть бути хаотично розкидані по оперативній пам'яті, а логічна послідовність буде забезпечуватися одним або декількома вказівниками.

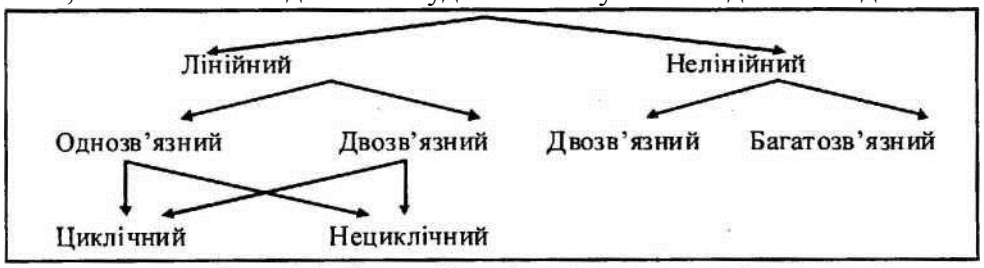


Рис. 4.3. Класифікація зв'язних списків.

5.1. ПОНЯТТЯ ХЕШ-ФУНКЦІЇ

Для прискорення доступу до даних у таблицях можна використовувати попереднє впорядковування таблиці відповідно до значень ключів.

При цьому можуть бути використані методи пошуку у впорядкованих структурах даних, наприклад, метод половинного розподілу, що істотно скорочує час пошуку даних за значенням ключа. Проте при додаванні нового запису вимагається перевпорядкувати таблицю. Втрати часу на повторне впорядковування таблиці можуть значно перевищувати виграш від скорочення часу пошуку. Тому для скорочення часу доступу до даних у таблицях використовується так зване **випадкове впорядковування** або **хешування.** При цьому дані організовуються у вигляді таблиці за допомогою хеш- функції А, яка використовується для обчислення адреси за значенням ключа (рис. 5.1).

Ідеальною хеш-функціоо є така хеш-функція, яка для будь-яких двох неоднакових ключів повертає неоднакові адреси.

Підібрати таку функцію можна у випадку, якщо всі можливі значення ключів наперед відомі.

Така організація даних має назву «досконале хешування». У разі наперед невизначеної множини значень ключів і обмеженої довжини таблиці підбір досконалої функції важко здійснити. Тому часто використовують хеш-функції, які не гарантують виконання умови.

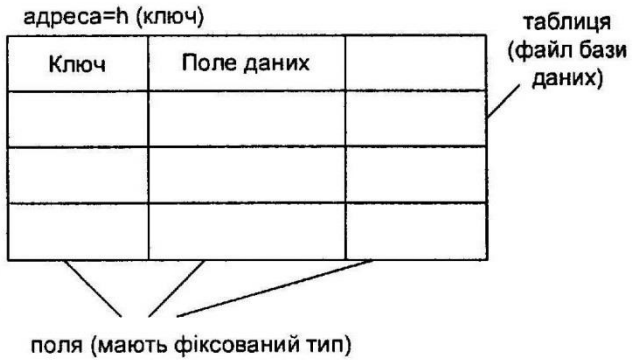


Рис. Структура хеш-таблиці

Приклад 5.1. Розглянемо приклад реалізації недосконалої хеш-функції на мові TurboPascal. Припустимо, що ключ складається із чотирьох символів. При цьому таблиця має діапазон адрес від 0 до 10000.

```
Function hash (key: string[4]): integer; varf: longint; begin
f:=ord (key[1 ]) - ord (key[2]) + ord (key[3]) -ord (key[4]);
{обчислення функції за значенням ключа} f:=f+255*2;
{ поєднання початку області значень функції з початковою адресою хеш-таблиці (a=1)}
f:=(f*10000) div (255*4);
```

{ поєднання кінця області значень функції з кінцевою адресою
хеш-таблиці (a=10 000)}
hash:=f
end;

При заповненні таблиці виникають ситуації, коли для двох неоднакових ключів функція обчислює одну і ту саму адресу. Даний випадок має назву **«колізія», а** такі ключі називаються **ключами-синонімами.**

5.2. АЛГОРИТМИ ХЕШУВАННЯ

Для визначеності вважатимемо, що хеш-функція **h{K)** має не більше як **М** різних значень і, що ці значення задовольняють умову

0 < *h(K) <M*

для всіх ключів **К.**

Теоретично неможливо так визначити хеш-функцію, щоб вона створювала випадкові дані з невипадкових реальних файлів. Але на практиці неважко зробити достатньо хорошу імітацію випадковості, використовуючи прості арифметичні діїРозглянемо, наприклад, випадок десятизначних ключів на десятковому комп'ютері. Сам собою напрошується наступний спосіб вибору хеш-функції: встановити Мрівним, скажімо, 1000, а як **h(K)** узяти три цифри, вибрані приблизно з середини 20-значного добутку **К*К.** Здавалося б, це повинно давати досить рівномірний розподіл значень між 000 і 999 з незначною ймовірністю часткою колізій. Насправді, експерименти з реальними даними показали, що такий метод **«серединних квадратів»** непоганий за умови, що ключі не містять багато лівих або правих нулів підряд.

З'ясувалося, проте, що існують надійніші йі простіші способи хеш-функцій.

Метод ділення особливо простий: використовується остача частка від ділення на **М**

h(K)=K mod M

У цьому випадку, очевидно, що деякі значення Мбудуть кращі за інших.Наприклад, якщо М-парне число, то значення h{K) буде парним при парному К, інакше—непарним; часто це приводить до значних зсувів даних. Зовсім погано брати А/рівним розрядності машинного слова, оскільки тоді h(K) дає нам праві значущі цифри К (К mod М не залежить від інших цифр). Аналогічно, Мне має бути кратним 3, бо буквенні ключі, що відрізняються один від одного лише регістром, могли б дати значення функції, різниця між якими кратна 3. (Причина криється в тому, що 10« mod 3= **4п** mod 3= 1.) Взагалі ми хотіли б уникнути значень М, які діляться на **rk** ± **a,** де **k** і **a** —невеликі числа, а **г** - «основа системи числення» для множини літер, що використовуються (зазвичай **г** =64, 256 і 100), оскільки остача від ділення на такі значення Мвиявляється суперпозицією цифр ключа.

Мультиплікативиа схема хешування полягає в заданні послідовність випадкових цілих чисел за формулою

X_{+1} AX(mod **M).**

Для машинної реалізації найзручнішим є **М= 2^g,** де *g* - розрядність машинного слова. Алгоритм подаємо нижче.

1 .Вибрати $\mathbf{X_O}$ - довільне непарне число.

2. Визначити коефіцієнт **X** = 8/ ± 3, де / - довільне ціле додатне число.

3.Знайти добуток ЯX_0 що містить не більше 2**g** значущих розрядів.

4. Взяти **g** молодших розрядів в якості $\mathbf{X_г}$

5.Знайти дріб х, **=XJ2^g** в інтервалі (0,1).

- Присвоїти А' - $\mathbf{X_{\{}}$**.**
- Повторити з п. 3.

Хороша хеш-функція повинна задовольняти дві вимоги:

39. її обчислення має бути дуже швидким;

40. вона повинна мінімізувати число колізій.

Властивість (а) частково залежить від особливостей машини, а властивість (Ь)- від характеру даних. Якби ключі були дійсно випадковими, можна було б просто виділити декілька бітів і використовувати їх для хеш-функції, але на практиці, щоб задовольнити (Ь), майже завжди

потрібна функція, залежна від усіх бітів.

5.3. ДИНАМІЧНЕ ХЕШУВАННЯ

5.3.1. Означення динамічного хешування

Описані вище методи хешування є статичними, тобто спочатку виділяється деяка хеш-таблиця, під її розмір підбираються константи для хеш-функції. На жаль, це не надається для завдань, у яких розмір бази даних часто змінюється. У міру зростання бази даних можна

- ✓ *користуватися початковою хеги-функцією, втрачаючи продуктивність через зростання колізій;*
- ✓ *вибрати хеш-функцію «із запасом», що спричинить невиправдані втрати дискового простору;*
- ✓ *періодично змінювати функцію, перераховувати всі адреси; це забирає дуже багато ресурсів і виводить з ладу базу на деякий час.*

Існує техніка, що дозволяє динамічно змінювати розмір хеш-структури. Це - динамічне хешування. Хеш-функція генерує так званий псевдоключ, який використовується лише частково для доступу до елементу. Іншими словами, генерується досить довга бітова послідовність, яка має бути достатня для адресації всіх потенційно можливих елементів. У той час, як при статичному хешуванні було б треба дуже велику таблицю (яка зазвичай зберігається в оперативній пам'яті для прискорення доступу), тут розмір зайнятої пам'яті прямо пропорційний кількості

елементів в базі даних. Кожен запис в таблиці зберігається не окремо, а в якомусь блоці ("bucket"). Ці блоки збігаються з фізичними блоками на пристрої зберігання даних. Якщо в блоці немає більше місця, щоб вміщати запис, то блок ділиться на два, а на його місце ставиться вказівник на два нові блоки.

Завдання полягає в тому, щоб побудувати бінарне дерево, на кінцях гілок якого б)ли б вказівники на блоки, а навігація здійснювалася б на основі псевдоключа. Вузли дерева можуть бути двох видів: вузли, які показують на інші вузли або вузли, які показують на блоки. Наприклад, нехай вузол має такий вигляд, якщо він показує на

Zero	Null
Bucket	Вказівник
One	Null

Якщо ж він вказуватиме на два інші вузли, то він матиме такий вигляд:

Zero	Адреса a
Bucket	Null
One	Адреса b

Спочатку є тільки вказівник на динамічно виділений порожній блок. При додаванні елемента обчислюється псевдоключ, і його біти по черзі використовуються для визначення місця розташування блоку.

5.3.2. Розширюване хешування

Розширюване хешування близьке до динамічного. Цей метод також передбачає зміну розмірів блоків у міру зростання бази даних, але це компенсується оптимальним використанням місця. Оскільки за один раз розбивається не більш як один блок, накладні витрати досить малі.

Замість бінарного дерева розширюване хешування передбачає список, елементи якого посилаються на блоки. Самі ж елементи адресуються за деякою кількостю і бітів псевдоключа. При пошуку береться / бітів псевдоключа і через список (сііgес- Югу) знаходиться адреса шуканого блоку. Додавання елементів відбувається складніше. Спочатку виконується процедура, аналогічна до пошуку. Якщо блок неповний, додається запис у нього і в базу даних. Якщо блок заповнений, він розбивається на два, записи перерозподіляються за описаним вище алгоритмом. У цьому випадку можливе збільшення числа бітів, необхідних для адресації. Тоді

розмір списку подвоюється і кожному новому створеному елементу присвоюється вказівник, який містить його батько. Отже, можлива ситуація, коли декілька елементів показують на один і той самий блок. Зауважимо, що за одну операцію додавання перераховуються значення не більш, ніж одного блоку. Видалення здійснюється за таким самим алгоритмом, тільки навпаки. Блоки, відповідно, можуть бути склеєні, а список - зменшений у два рази.

Отже, основною перевагою розширюваного хешування є висока ефективність, яка не знижується при збільшенні розміру бази даних. Окрім цього, розумно витрачається місце на пристрої зберігання даних, оскільки блоки виділяються тільки під реальні дані, а список вказівників на блоки має розміри, мінімально необхідні для адресації заданої кількості блоків. За ці переваги розробник розплачується додатковим ускладненням програмного коду.

Тема 2. Визначення дерева, «бінарне дерево», алгоритми проходження дерев углиб та вшир.

6.1. ДЕРЕВО

6.1.1. *Визначення дерева*

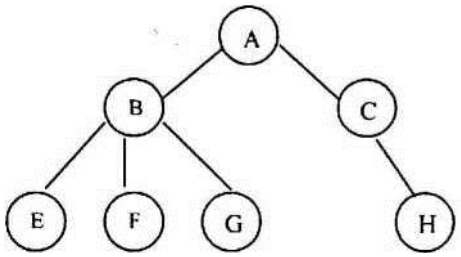
❖ ***Дерево*** *— скінченна непорожняТ, гцо складається з одного й більше вузлів таких, що виконуються наступні умови:*

✓ *є один спеціально позначений вузол, який називається коренем дерева;*

✓ *інші вузли (крім кореня) містяться в т>=0 попарно не пересічних множинах $T_г$ $T_у$ .... Т , кожна з яких у свою чергу, є деревом. Дерева $T_Г$ T_2.. T_m називаються піддеревами такого кореня.*

Дерева зображаються такими способами:

- *графічно,*
- *за допомогою множин,*
- *як модифікація багатозв язних списків.*



A: T_1 = {B, E, F, G}
T_2 = {C, H}
B: T_{11} = {E}
T_{12} = {F}
T_{13} = {G}
C: T_{21} = {H}

а) графічний *б) за допомогою множин*

Рис. Способи зображення дерева.

Якщо підмножини $\mathbf{T_x}$, $\mathbf{T_г}$, ..., $\mathbf{T_т}$ упорядковані, то дерево називають **упорядкованим.** Якщо два дерева вважаються рівними й тоді, шли вони відрізняються порядком, то такі дерева називаються **орієнтованими деревами.** Кінцева множина непересічних дерев називається **лісом** (рис. 6.2).

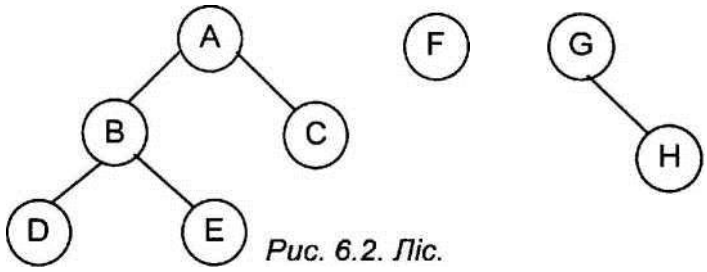


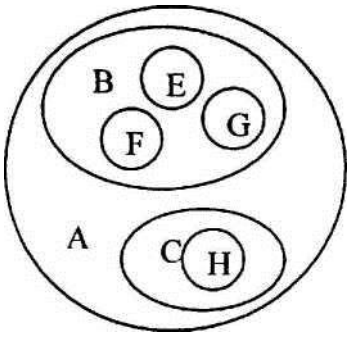
Рис. 6.2. Ліс.

6.1.2. Бінарне дерево

Бінарне дерево - скінченна множина елементів, що може бути порожньою, яка складається з кореня й двох непересічних бінарних дерев, причому піддерева впорядковані: ліве піддерево й праве піддерево.

ь с> * о ь

Кількість підмножин для заданого вузла називається **ступенем вузла.** Якщо така кількість дорівнює нулю, то вузол є листом. Максимальний ступінь вузла в дереві - **ступінь дерева. Рівень вузла** - довжина шляху від кореня до розглянутого вузла. Максимальний рівень дерева - висота дерева.



Структуру дерева можна зображати й за допомогою способів, поданих на рис 6.3.

Вкладені множини

Дужкова форма: (А (В (Е) (її) (в)) (С (Н)))

Десяткова форма Дьюі: А—І;

В-І.І; С— 1.2;

Е-1.1.1; Р~1.1.2; С—1.1.3; Н-1.2.1

Рис. 6.3. Способи подання дерев.

6.1.3. Подання дерев у зв'язній пам'яті комп'ютера

Розрізняють три основні способи подання дерев у зв'язній пам'яті: стандартний, інверсний, змішаний. Розглянемо ці способи для дерева зображеного на рис. 6.4.

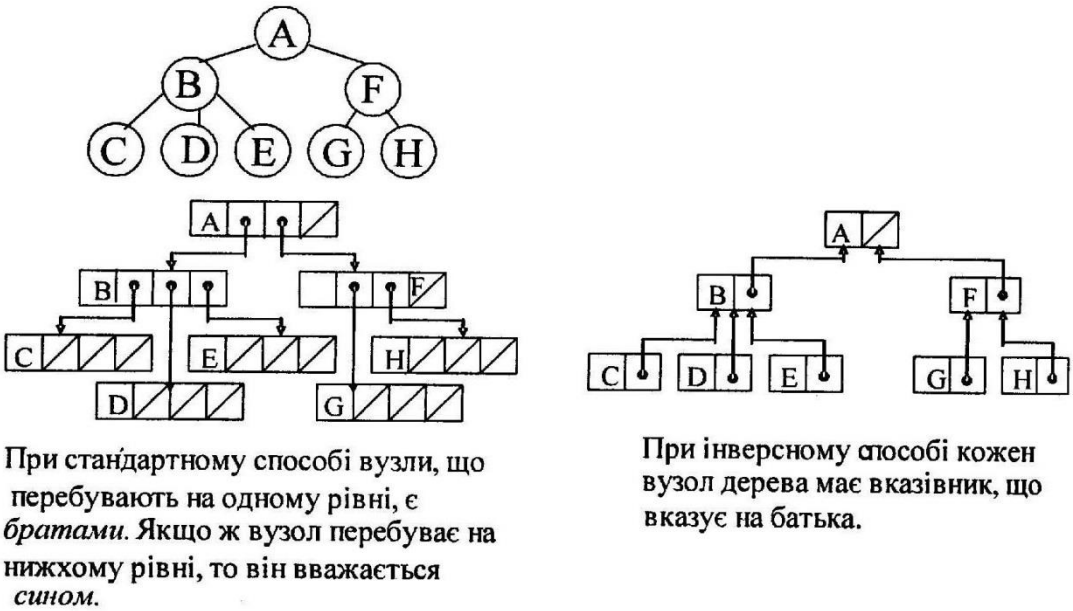


Рис. Стандартний та інверсний способи подання дерев

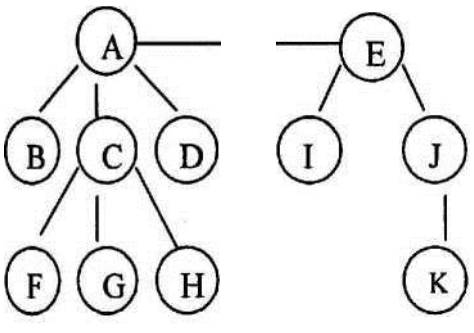
Якщо ж говорити про змішаний спосіб подання дерева у зв'язній пам'яті, то тут, як видно з назви, кожний вузол включає вказівники, що вказують як на синів, так і на батька.

Функція побудови дерева стандартним способом: struct btree // структура дерева {

```
        int el;
        struct btree *l,*r;
    }; // вказівники на лівого та правого сина //вказівники на корінь
  дерева та поточний елемент struct btree *root, *cur;
//функція додавання елемента у дерево
struct btree * insert (struct btree *c, int k, struct btree * new1)
        {
struct btree *p Д int a;
        if(!c) //поточний елемент є порожнім (відсутнім)
        {
            c=(struct btree*) malloc (sizeof(struct btree));
```

```
            c->el=k;
            c->l =NULL;
            c->r =NULL;
            //якщо задано значення a, то дерево вже містить вузли // і ми здійснюємо прив'язку створенного елемента
            // як лівого (a=1) або правого (a=2) сина if (a==1) new1->l=c; if (a==2) new1->r=c;

        return (c);
    }
    else
    //спускаємося далі по дереву
    {
    new1=c;
        if(c->el>k)
        //переходимо до лівого сина
        {
            c=c->I; a=1; p=insert (c,k,new1);
        }
        else
        //переходимо до правого сина
        {
            c=c->r; a=2; p=insert (c,k,new1);
        }
    }
}
void main()
//виклик функції побудови дерева
{
int          kIn=5,l;          scanf("%d",&k);
root=insert(NULL,k,cur); for(i=2;i<=n;i++)
    {
        scanf("%d",&k);
        cur=insert(root,k,cur);
    }
}
```



6.1.4. Алгоритми проходження дерев углиб і вшир

При проходженні вглиб зображеного дерева, список його вершин, записаних у порядку їхнього відвідування, буде виглядати так

A, B, C, ¥, *в,* H, *B,* E, I, *5,* K.

Алгоритм проходження дерева вглиб

спорожній стек S>;

спройти корінь і включити його в стек S>;

while <стек S не порожній> do

begin

{нехай P - вузол, що перебуває у вершині стека S} if <не всі сини вузла P пройдені»

then спройти старшого сина й включити його в стек S> else begin

свиключити з вершини стека вузол P>; if <не всі брати вершини P пройдені»

then спройти старшого брата й включити його в стек S>

end;
end;

При проходженні зображеного дерева вшир (по рівнях), список його вершин, записаних у порядку їхнього відвідування, буде таким:

A, B, C, D, E, F, G, H, I, J, K.

Алгоритм проходження дерева вширину: свзяти дві черги 01 і 02»; спомістити корінь у чергу 01»; while <01 або 02 не порожня» do begin

if <01 не порожня» then

{P - вузол, що перебуває в голові черги 01} begin

свиключити вузол із черги 01 і пройти його»; спомістити всі вузли, що відносяться до братів цього вузла P, у чергу 02»; end

else <01=02; *02=0*

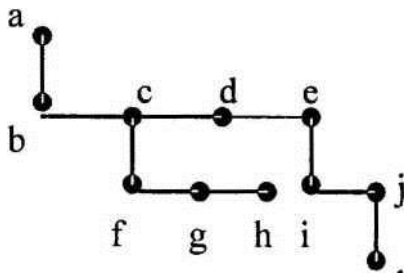
end;

6.*1.5. Подання дерев у вигляді бінарних*

Між деревами загального виду (вузол дерева може мати більше двох синів) і бінарними деревами існує взаємно однозначна відповідність, тому бінарні дерева часто використають для подання дерев загального виду.

Для такого подання використають наступний алгоритм:

- зображуємо корінь дерева;
- по вертикалі зображуємо старшого сина цього кореня;
- по горизонталі вправо від цього вузла подаємо всіх його братів;
- пп. I, 2, 3 повторюємо для всіх його вузлів.

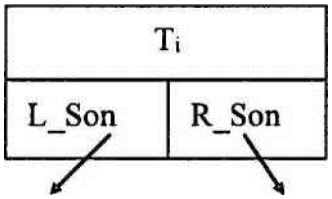


if n=0 then Tree:=nii else begin nl:=n div 2; nr:=n - nl - 1;
read (x); New (NewElement); with NewElement do begin Data:=x;
L_Son:=Tree (nl);
R_Son:=Tree (nr); end;
Tree:=NewElement;
end;
end;

Ефективність рекурсивного визначення полягає в тому, що воно дозволяє за допомогою кінцевого висловлення визначити нескінченну множину об'єктів.

6.1.5 Застосування бінарних дерев в алгоритмах пошуку

В однозв'язному списку неможливо використати бінарні методи, вони можуть використовуватися тільки в послідовній пам'яті. Однак, якщо використати бінарні дерева, то в такій зв'язній структурі можна одержати алгоритм пошуку зі складністю $0(\log_2 \mathbf{N})$. Таке дерево реалізується в такий спосіб; для будь-якого вузла дерева із ключем Г всі ключі в лівому піддереві повинні бути менші від Г, а в правому - більше **Т.** У дереві пошуку можна знайти місце кожного ключа, рухаючись, починаючи від кореня й переходячи на ліве або праве піддерево, залежно від значення його ключа. З **п** елементів можна організувати бінарне дерево (ідеально збалансоване) з висотою не більшою ніж $\log_2 N$, що визначає кількість операцій порівняння при пошуку.



< Ti > Ti

```
Function Search (x: integer; t: ElPtr): EIRtr;
{поле Data замінимо на поле Key} varf: boolean; begin f:=false;
while (t<>nil) and not f do if x=t^A. Key then f:=true else
ifx>t^A. Key then t:=t^A. R_Son else
t:=t^A. L_Son;
Search:=t;
end;
```

Якщо одержимо, що значення функції = nil, то ключа зі значенням ху дереві не знайдено.

```
Function Search (x: integer; t: ElPtr): ElPtr; begin
   S^A.key:=x; while t^A.key<>x do if
x>t^A.key then
          t:=t^A. R_Son else t:=t^A. L_Son;
   Search:=t;
```

6.1.8. Застосування бінарних дерев

1. **Дерево Хафмана** - двійкове дерево, листям якого є символи алфавіту, в кожній вершині якого зберігається частота символу (для листа) або сума частот його двох нащадків (називатимемо це число вагою вершини). При цьому виконується властивість: якщо відстань від кореня до вершини А"більша, ніж до вершини У, то вага **Х**не перевищує вагу У. Один із способів застосування дерев Хафмана - **алгоритми кодування (архівування) інформації (детальніше** див. розділ 10.3.3).

Код змінної довжини символа (змінний код у дереві Хафмана) - послідовність бітів, яку отримуємо при проходженні по ребрах від кореня до вершини із цим символом, якщо зіставити кожному ребру значення I або 0. У дереві Хафмана ребрам, що виходять із вершини до її нащадків присвоюються різні значення (вважатимемо далі, що ребро з I - ліве,

появи символів, будується двійкове дерево Хафмана, з нього отримуються відповідні кожному символу коди змінної довжини. Нарешті, знову здійснюється проходження початковим файлом, при цьому кожен символ замінюється на свій код у дереві. Отже, статичному алгоритму потрібні два проходи по файлу-джерелу, щоб закодувати дані. Статичний алгоритм:

```
{Ініціалізуємо двійкове дерево}
InitTree;
For i:=0 to Length(CharCount)
{Ініціалізуємо масив частот елементів алфавіту}
CharCount[i]:=0;
{Запускаємо перше проходження файлу}
While not EOP(Файл-джерело)
Begin
{Читаємо активний символ} i^eacI(Файл-джерело, C);
{Збільшуємо частоту його появи}
CharCount[C]= CharCount[C]+1;
End;
{За отриманими частотами будуємо двійкове дерево}
ReBuildTree;
WriteTree (Файл-приймач) {Записуємо у файл двійкове дерево} {Запускаємо друге проходження файлу}
While not EOP(Файл-джерело)
Begin
Read(Файл-джерело, C); {Читаємо поточний символ}
{Запускаємо другий прохід файлу} code=FindCodelnTree(C);
write( Файл-приймач, code); {Записуємо послідовність бітів у файл}
```

End

Процедура InitTree ініціал ізує двійкове дерево, онулюючи значення, ваги, і вказівники на батька і нащадків активного листка. Далі виконується перший прохід по джерелу даних з метою перевірки частотності символів, результати якого запам’ятовуються у масив CharCount. Розмірність цього масиву повинна дорівнювати кількості елементів алфавіту джерела. У загальному випадку, для стискання заданого комп’ютерного файлу його довжина повинна становити 256 елементів. Потім, відповідно до отриманого набору частот ,будуємо двійкове дерево ReBuildTree. Під час другого проходу переконуємо джерело відповідно до дерева і записуємо одержані послідовності бітів у стиснутий файл. Для того, щоб мати можливість відновити стиснуті дані, необхідно в отриманий файл зберегти копію двійкового дерева WriteTree.

Динамічний алгоритм:

{Ініціалізуємо двійкове дерево}

InitTree;

For i:=0 to Length(CharCount)

{Ініціалізуємо масив частот елементів алфавіту}

CharCount[i]:=0;

{Запускаємо проходження файлу}

While not EOP(Файл-джерело)

Begin

КеасІ(Файл-джерело, С); {Читаємо активний символ}

If С немає в дереві then {Перевіряємо чи зустрічався символ раніше} Code:=Kofl "порожнього” символу & Asc(C)

{Якщо ні, то запам’ятовуємо код порожнього листка і ASCII код активного символу}

{Інакше запам'ятовуємо код активного символу} else

code:=Kofl C;

{Записуємо послідовність бітів у файл} write(Файл-приймач, code);

{Оновлюємо двійкове дерево символом}

ReBuildTree(C);

End;

Динамічний алгоритм дозволяє реалізувати однопрохідну модель стискання. Не знаючи реальної ймовірності появи символів у початковому файлі, програма поступово змінює двійкове дерево, з кожним символом, що зустрічається, збільшуючи частоту його появи е дереві та перебудовуючи зв’язки у самому дереві. Проте, стає очевидним, що, вигравши в кількості проходжень початковим файлом, ми втрачаємо у стисканні, оскільки у статичному алгоритмі реальні частоти символів були відомі із самого початку і довжини кодів цих символів ближчі до оптимальних, тоді як динамічний метод, вивчаючи джерело, поступово доходить до його реальних частотних характеристик. Але, оскільки динамічне двійкове дерево постійно модифікується новими символами, немає необхідності запам’ятовувати їх частоти заздалегідь - при розархівуванні програма, отримавши з архіву код символа, так само відновить дерево, як вона це робила при стисканні, і збільшить на одиницю частоту символа.

2.Будь-який *алгебраїчний* вираз містить змінні, числа, знаки операцій і дужки можна подавати *у вигляді бінарного дерева* (виходячи з бінарності цих операцій). При цьому знак операції міститься в корені, перший операнд - у лівому піддереві, а другий операнд- у правому. Дужки при цьому опускаються. У результаті всі числа й змінні виявляться в листках, а знаки операцій - у внутрішніх вузлах.

Розглянемо приклад: 3 (x - 2) + 4.

Звичною формою виразів є інфіксна, коли знак бінарної 4 операції записується між позначеннями операндів цієї операції, наприклад, х-2. Розглянемо запис знаків операцій після позначень операндів, тобто постфіксний запис, наприклад, х 2 -. Такий запис має також назву зворотного польського, оскільки його запропонував польський логік Ян Лукасевич.

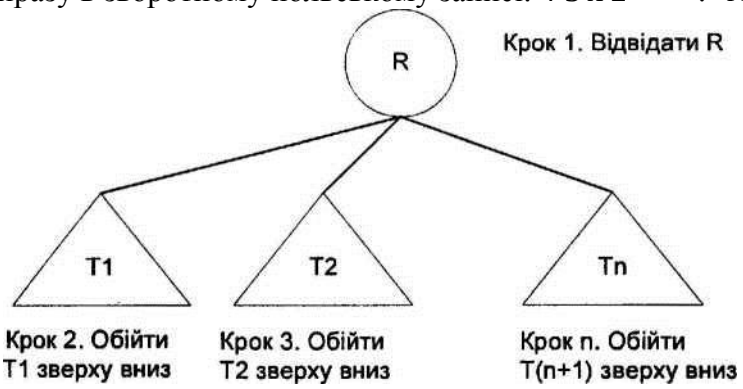
Сформулюємо правило обчислення значень виразу у інфіксному записі: вираз проглядається справа наліво, виділяються перші 2 операнди перед операцією, ця операція виконується, і її

результат — це новий операнд.

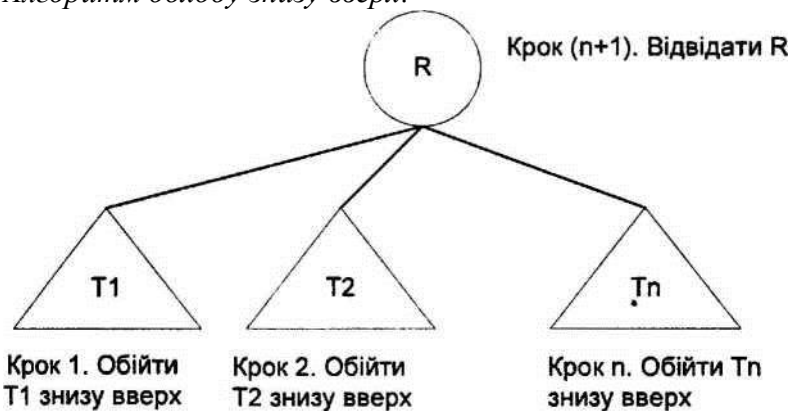
Правило обчислення значення виразу в зворотному польському записі: вираз проглядається зліва направо, виділяються перші 2 операнди перед операцією, ця операція виконується, і її результат - це новий операнд.

Запис виразу в інфіксній формі: 4 + 3 * x - 2.

Запис виразу в зворотному польському записі: 4 3 x 2 - * + . Алгоритм обходу зверху вниз:



Алгоритм обходу знизу вверх:



3. Класична програма із класу інтелектуальних: ***побудова дерев рішень.*** У цьому випадку не листкові (внутрішні) вузли містять предикати - запитання, відповіді на які можуть приймати значення “так” або “ні”. На рівні листків перебувають об’єкти (альтернативи, за допомогою яких розпізнаються програми). Користувач одержує запитання, починаючи з кореня, і, залежно від відповіді, спускається або на ліве піддерево, або на праве. Таким чином він виходить на той об’єкт, що відповідає сукупності відповідей на зппитання.

6.2. ВИДИ БІНАРНИХ ДЕРЕВ

6.2.1. Збалансоване дерево

У програмуванні *збалансоване дерево* в загальному розумінні цього слова - *це* такий різновид бінарного дерева пошуку, яке автоматично підтримує свою висоту, тобто кількість рівнів вершин під коренем, мінімальною. Ця властивість є важливою тому, *що* час виконання більшості алгоритмів на бінарних деревах пошуку пропорційний *до* їхньої висоти, і звичайні бінарні дерева пошуку можуть мати досить велику висоту в тривіальних ситуаціях. Процедура зменшення (балансування) висоти дерева виконується за допомогою трансформацій, відомих як обернення дерева, в певні моменти часу (переважно при видаленні або додаванні нових елементів).

Більш строге визначення збалансованих дерев було дане Г.Адельсон-Вельським та Є.Ландісом. *Ідеально збалансованим деревом* за Адельсон-Вельським та Ландісом є таке, у якого для кожної вершини різниця між висотами лівого та правого піддерев не перевищує одиниці. Однак, така умова доволі складна для виконання на практиці і може вимагати значної перебудови дерева при додаванні або видаленні елементів.

Тому було запропоноване менш строге визначення, яке отримало назву умови *АВЛ(AVL)-*

збалансованості і говорить, що бінарне дерево є збалансованим, якщо висоти лівого та правого піддерев різняться не більше ніж на одиницю. Дерева, що задовольняють такі умови, називаються AVL-деревами. Зрозуміло, що кожне ідеально збалансоване дерево є також АВЛ-збалансованим, але не навпаки.

Наведемо програму додавання та знищення елемента у збалансованому дереві. Type node - record {запис для визначення дерева}

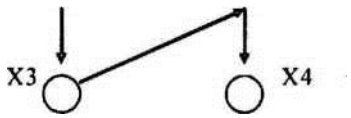
```
Key: integer;
Left, right: ref;
Bal: -1 ..1;                    {показує різницю висоти гілок дерева}
End;
procedure search(x: integer; var p: ref; varh: boolean);
var p1, p2: ref; {h = false}
begin
if p = nil then
begin                           {вузла немає у дереві; включити його}
new(p); h := true; with pл do begin
key := x; count := 1; left := nil; right := nil; bal := 0; end end else
ifx<pA.key then begin
search(x, pMeft, h);
if h then                       {виросла ліва гілка}
case pA.bal of
1: begin pA.bal := 0; h := false end ;
0: pA.bal :=-1;
-1: begin {балансування} p1:= pMeft; if p1A.bal = -1 then {однократний правий поворот} begin
//
```

Тема 3. Поняття графу, алгоритми проходження графу вглиб та вшир, топологічне сортування,

пошук мостів.

7.1. ПОНЯТТЯ ГРАФУ

❖ Граф — *двійка О = (X,U), де X - множина елементів (вершин, вузлів), а U - бінарне відношення на множині X* (U∩Xx *X). Якщо |X| = n, то граф є скінченим. Елементи и називають або дугами, або ребрами.*



X = {XI, X2, X3, X4}
XxX = {(X1,X1), ...,(X4,X4)}
U = {(XI,X2), (XI,X3), (X2,X3), (X2,X4)}=> $UcXxX = X^2$

Якщо (X, X) - впорядкована пара, то такий граф називається орієнтованим (орграфом,), а елементи *1!* називаються дугами.

Якщо (X, X) = (X, X), то граф - неорієнтований (неорграф), а елементи *и* називаються ребрами.

Я: II —> Я - якщо задано таку функцію, то граф є зваженим, де *Я* - множина дійсних чисел, тобто відображення дуги на число.

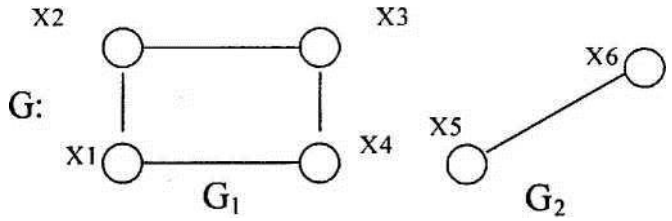
Загалом: 0 *17я X x X*.

Для орієнтованого графу кількість ребер, що входять у вузол, називається напівступенем вузла, кількість ребер, що виходять з вузла - напівступенем результату. Кількість вхідних та вихідних ребер може бути довільною, у тому числі і нульовою. Граф без ребер називається нуль-графом.

Мультиграфом називається граф, що має паралельні (що сполучають одні і ті ж вершини) ребра, інакше граф називається простим.

Компонентами орграфу є: дуга, шлях, контур. Шлях - така послідовність дуг, у якій кінець кожної попередньої дуги збігається з початком наступної. Контур - кінцевий шлях, у якому початкова вершина збігається з кінцевої. Граф, у якому є контур, назвається циклічним. Контур одиничної довжини називають петлею.

Компонентами неорграфу є, відповідно: ребро, ланцюг, цикл. Ланцюг-безперервна послідовність ребер між парою вершин неорієнтованого графу. Неорієнтований граф називають зв'язним, якщо будь-які дві його вершини можна з'єднати ланцюгом. Якщо ж граф - незв'язний, то його можна розбити на підграфи. Наприклад:



Слабка зв'язність - орграф замінюється неорієнтованим графом, який, у свою чергу, є зв'язним. **Однобічна зв'язність** - це така зв'язність, коли між двома вершинами існує шлях в одну або в іншу сторону. **Сильна зв'язність** — це зв'язність, коли між будь-якими двома вершинами існує шлях в одну й в іншу сторону.

Отже, багатозв'язна структура має такі властивості:

- на кожен елемент (вузол, вершину) може бути довільна кількість посилань;
- кожен елемент може мати зв'язок з будь-якою кількістю інших елементів;
- кожна зв'язка (ребро, дута) може мати напрямок і вагу.

Дерево - зв'язний граф без циклів.

Ліс (або ациклічний граф) - неограф без циклів (може бути і незв'язним).

Контур (каркас) зв'язного графу - дерево, що містить всі вершини графу. Визначається неоднозначно.

Редукція графу - контур з найбільшим числом ребер.

Цикломатичне (або циклічний ранг) число графу £=ш-я+с, де *п* - кількість вершин, *т* - кількість ребер, *с* - кількість компонент зв'язності графу.

Коциклічний ранг (або коранг) *Е*-п-с.*

Неограф Сг є лісом тоді і тільки тоді, коли *$(С)^{2}-0.*

Неограф *О* має єдиний цикл тоді і тільки тоді, коли 6ХС)=1.

Контур неографу має 5^4 ребер.

Ребра графа, що не входять в контур, називаються хордами.

Цикл, що виходить при додаванні до контуру графу його хорди, називається фундаментальним щодо цієї хорди.

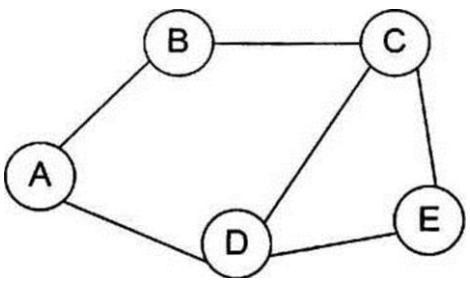
7.2. ПОДАННЯ ГРАФУ В ПАМ'ЯТІ КОМП'ЮТЕРА

Графічний спосіб подання (якщо граф невеликий).

Використання матриць. Матриця легко описуванняється, й при аналізі характеристик графу можна використати алгоритми лінійної алгебри. Також використається подання графа у зв'язній пам'яті, у тому випадку, якщо значна кількість елементів у матриці дорівнює нулю (матриця не заповнена).

Одним із матричних способів подання графу є **матриця суміжності.** Нехай задано граф *Є = (X, Ц),\X\ = n.* Маємо матрицю *А* розмірності *пхп,* що називається *матрицею суміжності*, якщо елементи її визначаються так:

$$a_{ij} = \begin{cases} 1, (X_i X_j) \in U \\ 0, (X_i X_j) \notin U \end{cases}$$



Приклад 7.1. Нехай маємо граф, поданий рис. 7.1

Рис. Приклад графу

Йому відповідає така матриця суміжності:

	A	в	с	D	E
A	0	1	0	1	0
B	0	0	1	1	0
C	0	0	0	1	1
D	I	0	1	0	1
E	0	0	1	1	0

Розглянемо застосування матричної алгебри для визначення характеристик графу. Вираз о $_{ik}L$ a_{t} означає, що між вузлами *і* і у є дві дуги, що проходять через вузол к, якщо значення виразу дорівнює True.

n

Вираз $У^{a}$*ік* ^{Aa}*к і* означає, що завжди є шлях між цими вузлами довжиною 2, якщо вираз є істинним.

А Ь *А* - $A^{(7>}$ - логічні операції заміняються арифметичними. Тоді кожний елемент $a_{ч}$ буде подавати інформацію про те, є чи шлях з / в у (ї,у = 1, 2

Вираз $A^{(n)} = A^{(п\sim 11}$ Ь *А* означає, чи є шлях довжиною п між різними вузлами і. По діагоналі буде характеристика, чи є цикли (контури) у матриці.

небудь шлях між вузлами *і* та j. Алгоритм обчислення заданого виразу:

- *Р = А;*
- повторити 3, 4 (А=1,2
- повторити4 для/= 1,2, ...,*n*\
- повторити $P_{u} = P_{i\,i}$ V $(P_{ік}$ Ь $P_{к})$,] = 1, 2,..., *n*...

У зв'язній пам'яті найчастіше подання графу здійснюється за допомогою структур

суміжності. Для кожної вершини множини *X* задається множина *M(X.)* відповідно до дуг її послідовників (якщо це орграф) або сусідів (для неорграфу). Отже, структура суміжності графу *Є* буде списком таких множин: < $M(X_1)$, $M(X_2)$,..., $M(X_n)$> для всіх його вершин.

Приклад Нехай маємо граф, поданий рис. 7.2 (вузли позначаємо у вигляді цифр: 1,2,..., и):

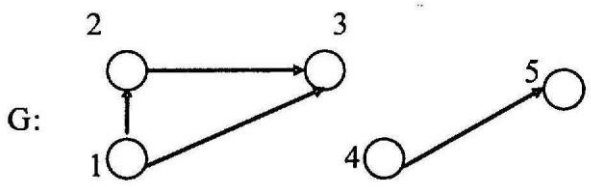


Рис. Подання графу

Для неорграфу:
1:2,3;
2:1,3;
3:1,2;
4:5;
5:4.
Для орграфу:
1:2;
2:3 3:1 4:5 5:-.

Структуру суміжності можна реалізувати масивом з *n* лінійно зв'язаних списків:

Подання графу може вплинути на ефективність алгоритму. Часто запис алгоритмів на графах задається в термінах вершин і дуг, незалежно від подання графу. Наприклад, алгоритм визначення кількості послідовників вершин: С (Л0=О, *S* — кількість дуг.

S = 0;
∀ x ∈ X виконати:
початок
C(x)=0;
∀ t ∈ M(x) виконати: C(x) = C(x) + 1;
S = S + C(x);
кінець;

7.3. АЛГОРИТМИ ПРОХОДЖЕННЯ ГРАФУ

Алгоритм проходження може бути використаний як алгоритм пошуку, якщо вузлами графу є елементи таблиці.

Маємо граф $G = (X, U)$, $X= \{x_1,x_2,...,x_n\}$... Кожне проходження можна розглядати як певну послідовність. Максимальна кількість проходжень (перестановок) – *n!*.

7.3.1. Алгоритм проходження графу вглиб

Для пояснення принципу проходження графу вглиб скористаємося графом, поданим на рис. 7.3.

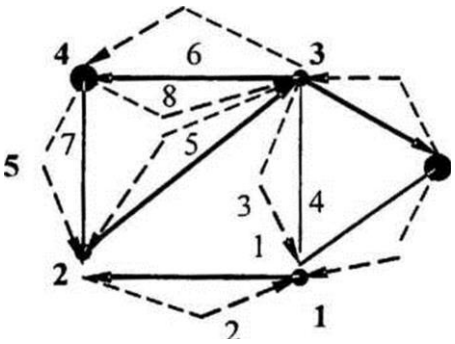


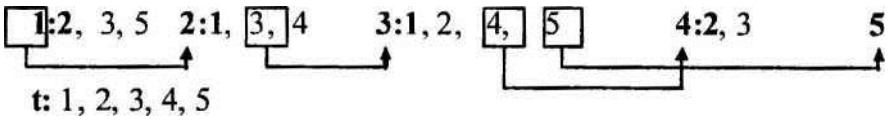
Рис. Демонстрація проходження графу вглиб.

Цифри на ребрах графу позначають кроки відвідування. Будемо розрізняти відвідувані (¬) і не відвідувані (¬) вершини. Кожна вершина обходиться два рази. Якщо всі суміжні вершини пройдені, то повертаємося у попередню вершину.

Проходження графу вглиб здійснюється за такими правилами:

- перебуваючи у вершині *x,* треба рухатися до будь-якої іншої, раніше не відвіданої вершини (якщо така знайдеться), одночасно запам'ятовуючи дугу, по якій ми вперше потрапили до цієї вершини;
- якщо з вершини *x* ми не можемо потрапити до раніше не відвіданої вершини або такої взагалі немає, то ми повертаємося у вершину *z,* з якої вперше потрапили до x, і продовжуємо обхід вглиб з вершини *z.*

Визначимо списки суміжності для кожної вершини графу *G:*



Якщо граф G зв'язний, то описаний процес визначає проходження (обхід) графу *G.* Якщо ж граф *G* не зв'язний, то проходимо тільки одну з компонентів графу *G*, що містить початкову вершину. Якщо граф *G* є незв'язним, то для одержання повного обходу необхідно досягати такого результату у кожному зв'язному компоненті. За допомогою цього методу можна визначити кількість компонентів.

Для кожного вибору початкової вершини у зв'язному графі може бути отримане єдине проходження.

Алгоритм проходження графу вглиб буде виглядати так:

```
procedure ОБХІД-ВГЛИБ(p: вершина);
begin
відвідати вершину p;
for all q from множини вершин, суміжних до p, do if q ще не відвідана then
ОБХІД-ВГЛИБ(д) end end end; begin
        for all p from множини вершин G do
    if p ще не відвідувалась then ОБХІД-ВГЛИБ(p) end end end.
```

На мові Паскаль рекурсивна процедура проходження вглиб виглядає так:

```
procedure dfs(v:integer);
var
i: integer; begin
    used[v]:=true; {відзначити вершину як відвідану} for i:=1 to n do
        {якщо між вершинами є зв'язок та вершина не відвідана} if
     (a[v,i]=1)and(not used[i]) then
    dfs(i); {викликаємо процедуру з цією вершиною}
end;
```

Нерекурсивна функція проходження графу вглиб виглядає так:

```
procedure dfs(v: integer);
var
i: integer; found: boolean; begin
    used[v]:=true; {відзначили вершину як відвідану} inc(c); {збільшуємо
 кількість занесених у стек вершин}“ st[c] := v; {занесли у стек}
{доки стек не порожній} while c > 0 do begin
                v := st[c]; {беремо вершину з голови стеку} found := false; {шляху не знайдено}
                {проходимо по усіх вершинах з метою пошуку шляху з обраної
                вершини}
                for i := 1 to n do
                    if a[v, i] and not used [i] then begin
                        found ;= true; {знайшли шлях} break;
                        end;
                   {якщо шлях знайдений} if found then
                 begin
                        used [i] := true; inc(c); {додається у стек}
                    st[c] := i; p[i]:=v; end
                 else {вилучаємо вершину зі стека} dec(c); end;
end;
```

Змістовий модуль 3. Алгоритми пошуку. Загальна класифікація та принципи роботи.

Тема 1. Загальна класифікація алгоритмів пошуку. Лінійний пошук, двійковий (бінарний) пошук елементів в масиві, пошук методом Фібоначчі. М-блоковий пошук.

АЛГОРИТМИ ПОШУКУ

Одна з тих дій, які найбільш часто зустрічаються в програмуванні – пошук. Існує декілька основних варіантів пошуку, і для них створено багато різноманітних алгоритмів.

Задача пошуку – відшукати елемент, ключ якого рівний заданому „аргументу пошуку”. Отриманий в результаті цього індекс забезпечує доступ до усіх полів виявленого елемента.

1.1. Послідовний (лінійний) пошук

Найпростішим методом пошуку елемента, який знаходиться в неврегульованому наборі даних, за значенням його ключа є послідовний перегляд кожного елемента набору, який продовжується до тих пір, поки не буде знайдений потрібний елемент. Якщо переглянуто весь набір, і елемент не знайдений – значить, шуканий ключ відсутній в наборі. Цей метод ще називають методом повного перебору.

Для послідовного пошуку в середньому потрібно *N/2* порівнянь. Таким чином, порядок алгоритму – лінійний – *O(N)*. Якщо елемент знайдено, то він знайдений разом з мінімально можливим індексом, тобто це перший з таких елементів. Рівність $i=N$ засвідчує, що елемент відсутній.

Єдина модифікація цього алгоритму, яку можна зробити, – позбавитися перевірки номера елементу масиву в заголовку циклу ($i<N$) за рахунок збільшення масиву на один елемент у кінці, значення якого перед пошуком встановлюють рівним шуканому ключу – *key* – так званий „бар’єр”.

Якщо нема додаткових вказівок про розташування необхідного елемента, то природнім є послідовний перегляд масиву із збільшенням тієї його частини, де бажаного елемента не знайдено. Такий метод називається лінійним пошуком. Умови закінчення пошуку наступні.

5. *Елемент знайдений, тобто a = x.*
6. *Весь масив проглянувши і збігу не знайдено.*
7. *Це дає нам лінійний алгоритм:*

8. Звертаємо увагу, що якщо елемент знайдений, то він знайдений разом Із мінімально можливим індексом, тобто це перший з таких елементів. Очевидно, що закінчення циклу здійсниться, оскільки на кожному кроці значення *i* збільшується, і, отже, воно досягне за скінченну кількість кроків межі *N;* фактично ж, якщо збігу не було, це відбудеться після *N* кроків.

```
Int LinearSearch (int[] a, int N, int L, int R, int Key)
{
    for (int I = L;i<= R; i++)
if (a[i] = = Key)
return (i);
    return (-1 ); // елемент не знайдений
}
```

8.3. ДВІЙКОВИЙ (БІНАРНИЙ) ПОШУК ЕЛЕМЕНТА В МАСИВІ

Якщо у нас є масив, що містить впорядковану послідовність даних, то дуже ефективний двійковий пошук.

Змінні *Lb* і *Ub* містять, відповідно, ліву і праву межі відрізка масиву, де міститься потрібний елемент. Починаємо завжди з дослідження середнього елементу відрізка. Якщо шукане значення менше від середнього елемента, ми переходимо до пошуку у верхній половині відрізка, де всі елементи менші від тільки що перевіреного. Іншими словами, значенням *Ub* стає *(M*- 1) і на наступній ітерації ми працюємо з половиною масиву. Отже, в результаті кожної перевірки ми удвічі звужуємо область пошуку.

Блок-схема бінарного пошуку подана на рис. 8.1.

Функція, що реалізує бінарний пошук, має такий вигляд:

```
int BinarySearch (int a, int Lb, int Ub, int Key)
{
int M;
do
M = Lb + (Ub-Lb)/2;
//шукаємо середину
//відрізку
if (Key < a[M])
Ub = —M; //переходимо у ліву частину else if (Key > a[M])
Lb = ++M; //переходимо у парву частину else
return (M); if (Lb > Ub)
return (-1); //не знайдено while (1);
}
```

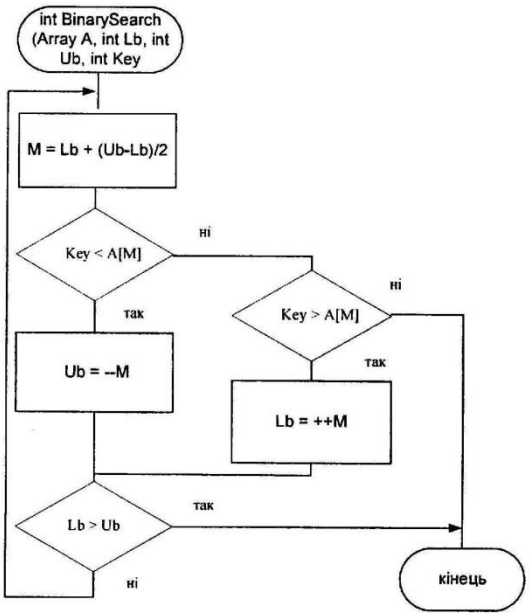


Рис. Блок-схема функції бінарного пошуку за параметром

8.4. ПОШУК МЕТОДОМ ФІБОНАЧЧІ

Він працює швидше, ніж бінарний пошук, оскільки замість операцій ділення, що застосовуються у попередньому методі, використовує операції додавання та віднімання. Суть методу полягає у визначенні наступного елемента для порівняння з числами Фібоначчі (звідси і назва). Зменшення індекса означає перехід по попереднього числа, збільшення - перехід до наступного.

Числа Фібоначчі формуються на основі додавання двох попередніх чисел, де перше і друге число дорівнюють 1. Тобто, можна скласти таку послідовність чисел:

1,1,2, 3, 5, 8,13,21,34...

Блок-схема пошуку методом Фібоначчі подана на рис. *int find_fibo(int strPar)*

```
int resFind, a[10];
int q=Fibonachi(1), p=Fibonachi(2), i=Fibonachi(3);
//функція, що визначає число Фібоначчі за номером
 int FindResult = -1;
do {
if(a[i]= strPar)
{
        FindResult = i; curSelRow = i+1; resFind =
    a[FindResult]; return (resFind);
}
else
    if (if(a[i]> strPar)
 if(q==0)
        {
            resFind = NULL; return (resFind);
        }
        else
            {i = i-q; p=q; q=p-q;} //перерахунок наступного числа
            else
              if(p==1)
            {
            resFind = NULL;
            return (resFind);
            }
            else
```

```
            {
            i=i+q; p=p-q; q=q-p;
            }
        }
        while(resFind);
            if(FindResult == -1)
                resFind = NULL;
        return (resFind);
}
```

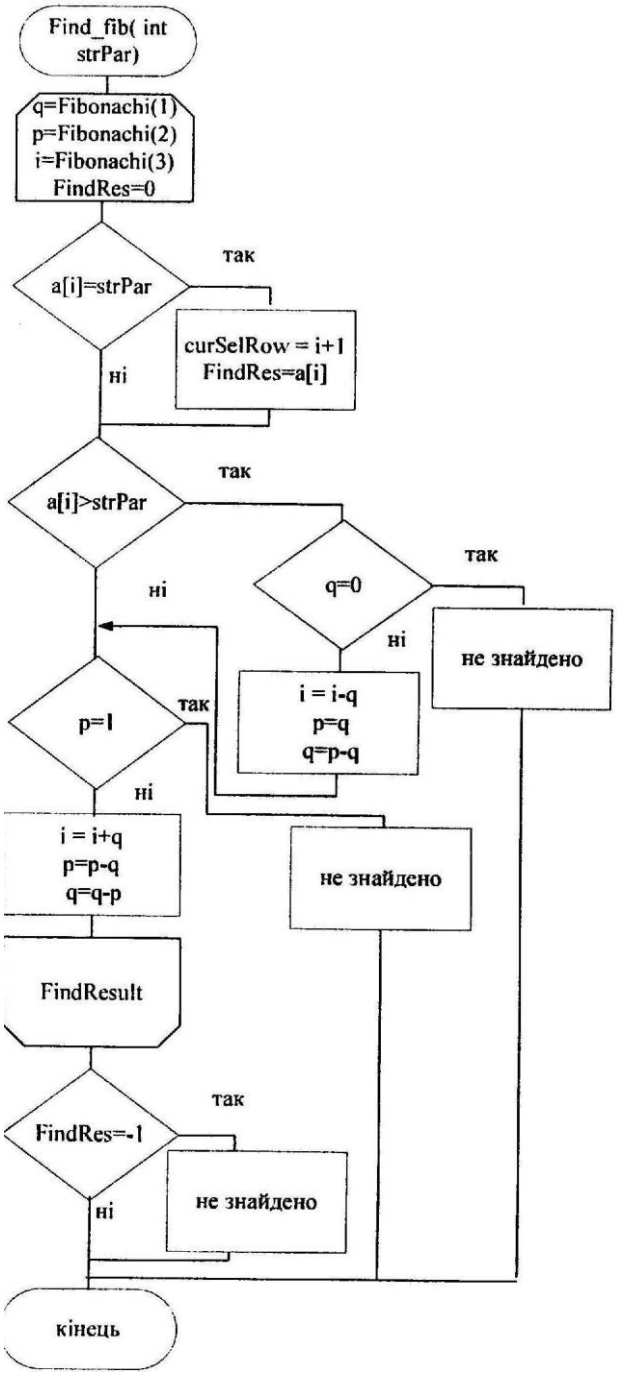


Рис. Блок-схема функції пошуку Фібоначчі за параметром.

Алгоритм пошук може бути значно ефективнішим, якщо дані будуть впорядковані.

Іншим, відносно простим, методом доступу до елемента є метод бінарного (дихотомічного) пошуку, який виконується в явно впорядкованій послідовності елементів.

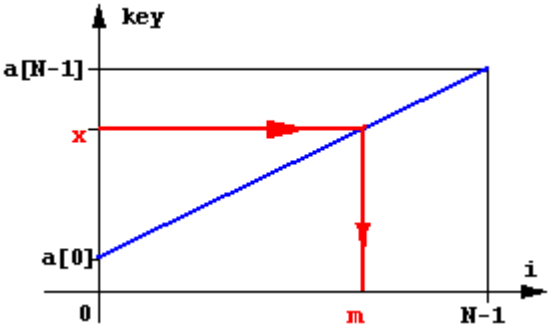
Оскільки шуканий елемент швидше за все знаходиться „десь в середині”, перевіряють саме середній елемент: *a[N/2]==key*? Якщо це так, то знайдено те, що потрібно. Якщо *a[N/2]<key*, то значення *i=N/2* є замалим і шуканий елемент знаходиться „праворуч”, а якщо *a[N/2]>key*, то „ліворуч”, тобто на позиціях *0…i*.

Для того, щоб знайти потрібний запис в таблиці, у гіршому випадку потрібно *log2(N)* порівнянь. Це значно краще, ніж при послідовному пошуку.

Максимальна кількість порівнянь для цього алгоритму рівна *log2(N)*. Таким чином, приведений алгоритм суттєво виграє у порівнянні з лінійним пошуком.

Відомо декілька модифікацій алгоритму бінарного пошуку, які виконуються на деревах.

1.2. Метод інтерполяції



Якщо немає ніякої додаткової інформації про значення ключів, крім факту їхнього впорядкування, то можна припустити, що значення *key* збільшуються від *a[0]* до *a[N-1]* більш-менш „рівномірно”. Це означає, що значення середнього елементу *a[N/2]* буде близьким до середнього арифметичного між найбільшим та найменшим значенням. Але, якщо шукане значення *key* відрізняється від вказаного, то є деякий сенс для перевірки брати не середній елемент, а „середньо-пропорційний”.

Вираз для поточного значення *i* одержано з пропорційності відрізків:

$$\frac{a[e]-key}{key-a[b]}=\frac{e-i}{i-b}$$

В середньому цей алгоритм має працювати швидше за бінарний пошук, але у найгіршому випадку буде працювати набагато довше.

1.3. Метод „золотого перерізу”

Деякий ефект дає використання так званого „золотого перерізу”. Це число φ, що має властивість:

$$\varphi-1=\frac{1}{\varphi};\quad \varphi^2-\varphi-1=0;\quad \varphi_{1,2}=\frac{1\pm\sqrt{5}}{2}.$$

Доданий корінь $\varphi=\frac{1+\sqrt{5}}{2}=1,61803398875\ldots$ і є золотим перерізом.

Згідно цього алгоритму відрізок слід ділити не навпіл, як у бінарному алгоритмі, а на відрізки, пропорційні φ та 1, в залежності від того, до якого краю ближче *key*.

1.4. Алгоритми пошуку послідовностей

Даний клас задача відноситься до задачі пошуку слів у тексті. Одним з найпростіших методів пошуку є послідовне порівняння першого символу з символами масиву. Якщо наявний збіг, тоді порівнюються другі, треті,... символи аж до повного збігу рядка *s* з частиною вектору такої ж довжини, або до незбігу у деякому символі. Тоді пошук продовжується з наступного символу масиву та першого символу рядку.

Існує варіант удосконалення цього алгоритму – це починати пошук після часткового збігу не з наступного елементу масиву, а з символу, наступного після тих, що переглядалися, якщо у рядку немає фрагментів, що повторюються.

Д. Кнут, Д. Моріс і В. Пратт винайшли алгоритм, який фактично потребує лише *N* порівнянь навіть в самому поганому випадку. Новий алгоритм базується на тому, що після часткового збігу початкової частини слова з відповідними символами тексту фактично відома пройдена частина тексту і можна „обчислити” деякі відомості (на основі самого слова), за допомогою яких потім можна швидко переміститися текстом.

Основною відмінністю КМП-алгоритму від алгоритму прямого пошуку є здійснення зсуву слова не на один символ на кожному кроці алгоритму, а на деяку змінну кількість символів. Таким чином, перед тим як виконувати черговий зсув, потрібно визначити величину зсуву. Для підвищення ефективності алгоритму необхідно, щоб зсув на кожному кроці був би якомога більшим.

Якщо *j* визначає позицію в слові, в якій міститься перший символ, який не збігається (як в алгоритмі прямого пошуку), то величина зсуву визначається як *j-D*. Значення *D* визначається як розмір самої довшої послідовності символів слова, які безпосередньо передують позиції *j*, яка повністю збігається з початком слова. *D* залежить тільки від слова і не залежить від тексту. Для кожного *j* буде своя величина *D*, яку позначимо d_j.

Так як величини d_j залежать лише від слова, то перед початком фактичного пошуку можна обчислити допоміжну таблицю *d*; ці обчислення зводяться до деякої попередньої трансляції слова. Відповідні зусилля будуть оправдані, якщо розмір тексту значно перевищує розмір слова (*M*<<*N*). Якщо потрібно шукати багатократні входження одного й того ж слова, то можна користуватися одними й тими ж *d*.

КМП-пошук дає справжній виграш тільки тоді, коли невдачі передувала деяка кількість збігів. Лише у цьому випадку слово зсовується більше ніж на одиницю. На жаль, це швидше виняток, ніж правило: збіги зустрічаються значно рідше, ніж незбіги. Тому виграш від практичного використання КМП-стратегії в більшості випадків пошуку в звичайних текстах досить незначний. Метод, який запропонували Р. Боуєр і Д. Мур в 1975 р., не тільки покращує обробку самого поганого випадку, але й дає виграш в проміжних ситуаціях.

БМ-пошук базується на незвичних міркуваннях – порівняння символів починається з кінця слова, а не з початку. Як і у випадку КМП-пошуку, слово перед фактичним пошуком трансформується в деяку таблицю. Нехай для кожного символу *x* із алфавіту величина *dx* – відстань від самого правого в слові входження *x* до правого кінця слова. Уявимо, що виявлена розбіжність між словом і текстом. У цьому випадку слово відразу ж можна зсунути праворуч на *dpM-1* позицій, тобто на кількість позицій, швидше за все більше одиниці. Якщо символ, який не збігся, тексту в слові взагалі не зустрічається, то зсув стає навіть більшим, а саме зсовувати можна на довжину всього слова.

Варто сказати, що запропоновані методи пошуку послідовностей можна модифікувати таким чином, щоб у кожному рядку пошук йшов не до кінця кожного рядка, а на кількість шуканих символів менше, бо слово *s* не може бути розташоване у кінці одного рядка та на початку наступного.

Тема 2. Метод обчислення адреси, інтерполяційний пошук в масиві, бінарний пошук, пошук в

таблиці, прямий пошук рядка.

8.6. МЕТОДИ ОБЧИСЛЕННЯ АДРЕСИ

Нехай у кожному з *M* елементів масиву *T* міститься елемент списку (наприклад, ціле позитивне число). Якщо є деяка функція *H(V),* що обчислює однозначно по елементі *V* його адресу - ціле позитивне число з інтервалу [0,*M*-1],то *V* можна зберігати в масиві *T* з номером *H(V)* тобто *V=T(H(V)).* При такому збереженні пошук будь-якого елемента відбувається за постійний час, не залежний від *M.*

Масив T називається масивом хешування, а функція H—функцією хешування (див. розділ 2).

При конкретному застосуванні хешування зазичай є визначена область можливих значень елементів списку *V* і деяка інформація про них. На основі цього вибирається розмір масиву хешування *M* і будується функція хешування. Критерієм для вибору *M* і *H є* можливість їхнього ефективного використання.

Нехай треба зберігати лінійний список з елементів $K_1,K_2,..,K_n$, таких, що при $K_i=K_j$

mod(K_i,26)= mod(K_j,26). Для збереження списку виберемо масив хешування T(26) із простором адрес 0-25 і функцію хешування $H(V)$= =mod(V,26). Масив T заповнюється елементами $T(H(K_i)) = K_i$ і $T(j)$=0, якщо $j \in$ у ($H(K_1)$, $H\{K_2),..,H(K_n)$).

Пошук елемента Vу масиві T із присвоюванням Z його індексу, якщо V міститься в T, чи -1, якщо V не міститься в T, здійснюється так:

```
int t[26],v,z,i;
        i=(int)fmod((double)v,26.0);
        if(t[i]==v) z=i;
         else z=-1;
```

Додавання нового елемента V у список з поверненням у Z індексу елемента, де він буде зберігатися, реалізується фрагментом

```
z=(int)fmod((doule)v,26.0);
t[zj=v;
```

а виключення елемента V зі списку присвоєнням

```
t[(int)fmod((double)v,26)3=0;
```

Тепер розглянемо складніший випадок, коли умова $K_i=K_j$ $H(K_{i)}=H(K_j)$ не виконується. Нехай V— множина можливих елементів списку (цілі позитивні числа), у якому максимальна кількість елементів дорівнює 6. Візьмемо M= 8 і як функцію хешування виберемо функцію $H(V)=mod(V,8)$.

Припустимо, що B=8, причому $H(K_1)=5$, $H(K_2)=5$, $H(K_3)=5$, $H(K_4)=5$, $H(K_5)=5$,тобто $H(K_2)=H(K_4)$ *хоча* $K=K_4$. Така ситуація, як показано у попередніх розділах, називається колізією, і в цьому випадку при заповненні масиву хешування треба метод для її дозволу. Зазвичай вибирається перша вільна комірка за власною адресою. Для нашого випадку масив Г[8] може мати вигляд

$$T=<0,K_5,0,K_2,K_4, K_1, K_3,0>$$

При наявності колізій ускладнюються всі алгоритми роботи з масивом хешування. Розглянемо роботу з масивом T[100], тобто з простором адрес від 0 до 99. Нехай кількість елементів N не більш 99, тоді в Т завжди буде хоча б один вільний елемент дорівнює нулю. Для оголошення масиву використовуємо оператор

```
int static t[ 100];
```

Додавання в масив Т нового елемента Z із занесенням його адреси в I і числа елементів у N виконується так:

```
i=h(z);
        while (t[i]!=0 && t[i]!=z) if
    (i==99) i=0; else i++;
        if (t[i]!=z) t[i]=z, n++;
```

Пошук у масиві Т елемента Z із присвоєнням I індекса Z, якщо Z є в Т, чи =-1, якщо такого елемента немає, реалізується в такий спосіб:

```
i=h(z);
        while (t[i]!=0 && t[i]!=z) if
    (i==99) i=0; else i++; if (t[i]==0) i=-
    1;
```

При наявності колізій виключення елемента зі списку шляхом позначення його як порожнього, тобто $t[i]=0$, може привести до помилки. Наприклад, якщо зі списку B виключити елемент K_2, то одержимо масив хешування $T=<0,K_5,0,K_2,K_4,K_5,K_3,0>$, у якому неможливо знайти елемент K_4, оскільки $H(K_4)=3$, *а* $T(3)=0$. У таких випадках при виключенні елемента зі списку можна записувати в масив хешування деяке значення, що не належить області значень елементів списку і не рівне нулю. При роботі з таким масивом це значення буде вказувати на те, що треба переглядати із середини комірок.

Перевага методів обчислення адреси полягає в тому, що вони найшвидші, а недолік у том}' що порядок елементів у масиві Т не збігається з їх порядком у списку, крім того, досить складно здійснити динамічне розширення масиву Т.

8.7. ІНТЕРПОЛЯЦІЙНИЙ ПОШУК ЕЛЕМЕНТА В МАСИВІ

Якщо відомо, що ключ лежить між K_1 і K_u то наступний пошук доцільно здійснювати не всередині впорядкованого масиву, а на відстані $(u-1)(K-K_1)/(K_u-K_1)$ від 1, припускаючи, що ключі є числами, що зростають приблизно в арифметичній прогресії.

Інтерполяційний пошук працює за log log *N* операцій, якщо дані розподілені рівномірно. Як правило, він використовується лише на дуже великих таблицях, причому робиться декілька кроків інтерполяційного пошуку, а потім на малому відрізку використовується бінарний або послідовний варіант.

```
Int interpolationSearch(int[] a, int toFind, int high)
{
// повертає індекс елемента зі значенням toFind або -1, якщо такого // елемента нема int low =
0;
      //high - кількість елементів масиву int mid;
while (a[low] < toFind && a[high] >= toFind)
{
          mid = low + ((toFind - a[low]) * (high - low)) / (afhigh] - a[low]); if (a[mid] <
      toFind) low = mid + 1 ; else if (a[mid] > toFind) high = mid - 1; else
              return mid;
}
if (a[low] == toFind) return low; else
          return -1 ; // Not found
}
```

8.8. БІНАРНИЙ ПОШУК ІЗ ВИЗНАЧЕННЯМ НАЙБЛИЖЧИХ ВУЗЛІВ

У ряді випадків (зокрема, в завданнях інтерполяції) доводиться з'ясовувати, де по відношенню до заданого впорядкованого масиву дійсних чисел розташовується задане дійсне число. На відміну від пошуку в масиві цілих чисел, задане число в цьому випадку найчастіше не співпадає ні з одним із чисел масиву, і вимагається знайти номери елементів, між якими це число може бути розміщене.

Одним з найшвидших способів цього є бінарний пошук, характерна кількість операцій якого має порядок $log_2(n)$, де *n* - кількість елементів масиву. При численних зверненнях до цієї процедури кількість операцій буде рівна m $log_2(n)$ (*m* - кількість обходів). Прискорення цієї процедури можна добитися за рахунок збереження попереднього результату операції і спроб пошуку при новому обігу в найближчих вузлах масиву з подальшим розширенням області пошуку у разі неуспіху.

При цьому в найгірших випадках кількість операцій буде більшою (приблизно у 2 рази) в порівнянні з бінарним пошуком, але, зазвичай, при ***m,*** значно більшою, ніж $log_2(n)$, вдається довести порядок кількості операцій до **т,** тобто зробити її майже незалежною від розміру масиву.

Завдання ставиться так. Заданий впорядкований масив дійсних чисел *array* розмірності ***n,*** значення *value,* що перевіряється, і початкове наближення вузла *old.* Вимагається знайти номер вузла *res* масиву *array*, такий, що *array* [*res*] <=*value*<*array*[*res* +1].

Алгоритм працює таким чином.

10. Визначається, чи лежить значення value за межами масиву array. У разі *value*<*array[0]* повертається -1, у разі *value*>*array[n-]]* повертається п-1.

11. Інакше перевіряється: якщо значення old лежить за межами індексів масиву (тобто *old*<0 або *old>0*, то переходимо до звичного бінарного пошуку, встановивши ліву межу *left*=0, праву *right*=*n*-1.

12. Інакше переходимо до з'ясування меж пошуку. Встановлюється *leß=right=old, inc*=\ - інкремент пошуку.

13. Перевіряється нерівність *vahie*>=*array*[*old*]. При його виконанні переходимо до

наступного пункту (5), інакше до пункту (7).

14. Права межа пошуку відводиться далі: *right~right+inc.* Якщо *right*>=n-1, то встановлюється *right*= n-1 і переходимо до бінарного пошуку.

15. Перевіряється *value>=array[righf\.* Якщо ця нерівність виконується, то ліва межа переміщається на місце правої: *left=right, inc* множиться на 2, і переходимо назад на (5). Інакше переходимо до бінарного пошуку.

16. Ліва межа відводиться: *left=left-inc.* Якщо *left*<=0, то встановлюємо *left*=0 і переходимо до бінарного пошуку.

17. Перевіряється *value<array[leß*]. При виконанні права межа переміщується на місце лівої: *right=left,* inc множиться на 2, переходимо до пункту (7). Інакше до бінарного пошук.

18. Проводиться бінарний пошук у масиві з обмеженням індексів *left* і *right.* При цьому кожного разу інтервал скорочується приблизно в 2 рази (у залежності від парності різниці), поки різниця між *left* і *right* не досягне 1. Після цього повертаємо *left* як результат, одночасно присвоюючи *old=left.*

```
int fbin_search(float value,int *old,float ‘array,int n)
{
register int left, right;
/* перевірка позиції за межами масиву 7 if(value < array[0]) retum(-1);
    if(value >= array[n-1]) return(n-l);
    процес розширення області пошуку. Перевіряємо валідність початкового
наближення 7 if(*o!d>=0 && *old<n-1)
        {
            register int inc=1; left = right =
         *old; if(value < arrayfold])
            {
                /* область розширюють вліво 7 while(1)
                {
                    left -= inc; if(left <= 0)
                    {
                        left=0;break;

                        }
                    if(array[left] <= value) break; right=left;
                 inc«=1;
                    }
                }
                else
                {
                    /* область розширюють вправо */ while(1)
                    {
                        right += inc; if(right >= n-1)
                        {
                            right=n-1 ;break;
                        }
                    }
                    if(array[right] > value) break; left=right; inc«=1;
                }
            /* початкове наближення погане- за область пошуку
          приймається весь масив */ else {
                left=0;right=n-1 ;
```

```
        }
        /* це алгоритм бінарного пошуку необхідного інтервалу */ while(left<right-1 )
        {
            register int node=(left+right)»1 ;
         if(value>=array[node]) left=node; eise
         right=node;
        }
    /* повертаємо знайдену ліву межу, обновивши старе
  значення результату */ retum(*old=left);
}
```

8.9. ПОШУК У ТАБЛИЦІ

Пошук у масиві іноді називають пошуком у таблиці, особливо, якщо ключ сам є складовим об'єктом, таким, як масив чисел або символів. Часто зустрічається саме останній випадок, коли масиви символів називають рядками або словами. Рядковий тип визначається так:

String =array[0..Mv1] of char

відповідно визначається і відношення порядку для рядків x і y:

x = *y,* якщо xj = yj для *0* face=«Symbol» =< j < *M*

x < *y* якщо xi < yi для 0 face=«Symbol» =< i < *M i* xj = yj

для 0 face=«Symbol» =< j < i

Щоб встановити факт збігу, необхідно встановити, що всі символи порівнюваних пядків відповідно рівні один іншому. Тому порівняння складових операндів зводиться .іошуку частин, що неспівпадають, тобто до пошуку на нерівність. Якщо нерівних частин не існує, то можна говорити про рівність. Припустимо, що розмір слів достатньо малий, скажімо, менше ніж ЗО. В цьому випадку можна використовувати лінійний пошук і діяти так.

Для більшості практичних застосувань бажано виходити з того, що рядки мають змінний розмір. Це припускає, що розмір вказується в кожному окремому рядку. Якщо виходити з раніше описаного типу, то розмір не повинен перевершувати максимального розміру М. Така схема достатньо гнучка і придатна для багатьох випадків, в той самий час вона дозволяє уникнути складнощів динамічного розподілу пам'яті. Найчастіше використовуються два такі подання розміру рядків.

Розмір неявно вказується шляхом додавання кінцевого символа, більше цей символ ніде не вживається. Зазвичай, для цієї мети використовується недрукований символ із значенням 00h. (Для подальшого важливо, що це мінімальний символ зі всієї множини символів.)

Такий прийом має ту перевагу, що розмір явно доступний, а недолік - тому, що цей розмір обмежений розміром множини символів (256).

У подальшому алгоритмі пошуку віддається перевага першій схемі. У цьому випадку порівняння рядків виконується так: i:=0;

while (x[i]=y[ij) and (x[i]<>00h) do i:=i+1;

Кінцевий символ працює тут як бар'єр.

Тепер повернемося до завдання пошуку в таблиці. Він вимагає вкладених пошуків, а саме: пошуку по рядках таблиці, а для кожного рядка послідовних порівнянь між компонентами.

Тема 3. Алгоритми: Ахо-Корасика, Моріса-Прата, Кнута, рабіна-Карпа, Боуера-Мура, Хорспула. Порівняння методів пошуку.

8.11. АЛГОРИТМ АХО-КОРАСИК

Алгоритм Ахо-Корасик - алгоритм пошуку підрядків в рядку, створений Альфредом Ахо і Маргарет Корасик. Алгоритм реалізує пошук множини підрядків із словника в даному рядку. Час роботи пропорційний Θ *(M+N+K)* , де *N*-довжина рядка - зразка, *M* — сумарна довжина рядків словника, а *K* - довжина відповіді, тобто сумарна довжина входжень слів із словника в рядок-зразок. Тому сумарний час роботи може бути квадратичним (наприклад, якщо в рядку 'ааааааа' ми шукаємо слова 'а', 'аа', 'ааа' ...).

```
q:=0;
        for i := 1 to m do begin
                while g (q, T [i]) = -1 do q : = f(q);
        q := g (q. т [i]);
        if out (q) * 0 then write(i), out (q); end;
```

8.12. АЛГОРИТМ МОРІСА-ПРАТА

Цей алгоритм модифікує прямий пошук, збільшуючи розмір зсуву, запам'ятовуючи одночасно частини тексту, що співпадають зі зраз ком.це дозволяє уникнути непотрібних порівнянь і збільшує швидкість пошуку.

Розглянемо порівняння на позиції *i,* де зразокх[0, *m* - 1] порівнюється зі частиною тексту *y*[*i, i* + *m*- 1]. Припустимо, що перша розбіжність відбулася між y[i + j] і x[j] де 1 <*j* < *m.*

Введемо поняття префікс-функції.

> ❖ ***Префікс-функцією*** *називається функція, що повертає найбільший префікс рядка. Префіксом рядка називається підрядок, який одночасно є і суфіксом (закінченням рядка). Так, для рядка «аавпа» префік-функція поверне символ «а», а для рядка «ааввсваа» - підрядок «аав».*

При зсуві можна очікувати, що префікс зразка «співпаде з якимсь суфіксом підслова тексту *и.* Найдовший такий префікс - межа *и* (він зустрічається на обох кінцях и). Це приводить нас до наступного алгоритму: нехай *mp_next*[*j*] - довжина межі x[0,j - 1]. Тоді після зсуву ми можемо відновити порівняння з місця y[i + y] і *x*[*j-mp*_next[j]] без втрати можливого місцезнаходження зразка. Таблиця *mp next* може бути обчислена за (*m)* перед самим пошуком. Максимальна кількість порівнянь на один символ - *m.* void PRE_MP(char *x, int m, int mp_next[])

```
{
        inti.j;
        i=0;
        j=mp_next[0]=-1; while (i < m
    )
        {
              while (j > -1 && x[i] != x[j])
            j=mp_next[j];
            mp_next[++i]=++j;
        }
}
void MP( char *x, char *y, int n, int m)
{
        int i, j, mp_next[XSIZE];
        PRE_MP(x, m, mp_next); i=j=0;
        while (i < n )
        {
            while (j > -1 && x[j] != y[i])
         j=mp_next[j]; i++; j++;
            if (j >= m)
            {
                ouTPUT(i-j);
                j = mp_next[j];
            }
        }
}
```

8.13. АЛГОРИТМ КНУТА, МОРІСАІ ПРАТТА

Приблизно в 1970 р. Д. Кнут, Д. Моріс і В. Пратт винайшли алгоритм (КМП), що фактично вимагає тільки У порівнянь навіть в найгіршому випадку. Новий алгоритм грунтується на тому міркуванні, що після часткового співпадіння початкової частини слова з відповідними

символами тексту фактично відома пройдена частина т" **-у** і можна визначити деякі відомості (на основі самого слова), за допомогою яких потім можна швидко просунутися по тексту. Приведений приклад пошуку слова ABCABD показує принцип роботи такого алгоритму. Символи, що порівнювалися, тут підкреслені. Зверніть увагу: при кожному неспівпадінні пари символів слово зсувається на усю пройдену відстань, оскільки менші зсуви не можуть привести до повного співпадіння.

ABCABCABAABCABD
ABCABD
ABCABD
ABCABD
ABCABD
ABCABD

Основною відмінністю КМП-алгоритму від алгоритму прямого пошуку є здійснення зсуву слова не на один символ на кожному кроці алгоритму, а на деяку змінну кількість символів. Отже, перш ніж здійснювати черговий зсуву, необхідно визначити величину зсуву. Для підвищення ефективності алгоритму необхідно, щоб зсув на кожному кроці був якомога більшим.

Якщо j визначає позицію в слові, що містить перший символ, що не співпала (як в алгоритмі прямого пошуку), то величина зсуву визначається як *j-D***.** Значення *D* визначається як розмір найдовшої послідовності символів слова, які безпосередньо передують позиції у, і яка повністю співпадає з початком слова. *D* залежить тільки від слова і не залежить від тексту. Для кожного *j* буде своя величина *D,* яку позначимо *d.*

Оскільки величини d_j залежать тільки від слова, то перед початком фактичного пошуку можна обчислити допоміжну таблицю *d.* Відповідні зусилля будуть виправданими, якщо розмір тексту значно перевищує розмір слова ***(M*N)*.** Якщо треба шукати багато входжень одного і того ж слова, то можна користуватися одними і тими самим d. Точний аналіз КМП-пошуку як і сам його алгоритм, вельми складний. Його винах - .ьаодять, що треба порядку M+N порівнянь символів, що значно краще,ніж M*N порівнянь із прямого пошуку. Вони так само відзначають таку позитивну властивість, як показник сканування, який ніколи не повертається назад, тоді як при прямому пошуку після неспівпадіння перегляд завжди починається з першого символа слова і тому може включати символи, які раніше вже були видимими. Це може привести до негативних наслідків, якщо текст читається з вторинної пам'яті, адже в цьому випадку повернення обходиться дорого. Навіть при введенні буфера може зустрітися таке велике слово, що повернення перевищить місткість буфера.

```
Varn: longint;
T: array[1 ..40000] of char;
S : array[1..10000] of char;
P: array[1 ..10000] of word; {масив, в якому зберігається значення префікс-
функції}
    i,k: longint; m : longint;
Procedure Prefix; {процедура, яка визначає префікс-функцію}
Begin
        P[1]:=0; {префікс рівний нулеві} к:=0;
          for i:=2 to m do begin
        while (k>0) and (S[k+1]<>S[i]) do
              k:=P[k]; {отримаємо значення функції з попередніх розрахунків} if S[k+1]=S[i]
           then k:=k+1;
           P[i]:=k; {присвоєння префікс-функції} end;
End;
```

8.14. АЛГОРИТМ РАБІНА-КАРПА

Ідея, запропонована Рабіном і Карпом**,** має на увазі поставити у відповідність кожному рядку деяке унікальне число і замість того, щоб порівнювати самі рядки, порівнювати числа, що набагато швидше. Проблема в тому, що шуканий рядок може бути довгим, рядків у тексті теж

вистачає. А оскільки кожному рядку треба поставити у відповідність число, то і чисел має бути багато, а отже, числа будуть великими (порядку D_m, це D - кількість різних символів), і працювати з ними буде так само незручно.

```
VarT : array[1 ..40000] of 0..9;
        S : array[1..8]of 0..9; i j : longint; n,m: longint;
    v,w: longint; {v - число, яке характеризує рядок, що шукається, w характеризує
  рядок довжини m у тексті} k ; longint;
const D : longint = 10; {кількість різних символів}
Begin
            v:=0;
            w:=0;
            for i:=1 to m do begin
       v:=v*D+S[i]; {визначення v, рядок поданий як число} w:=w*D+T[i]; {
    визначення початкового значення w} end; k:=1;
            for i:=1 to m-1 do
{k необхідне для багатократного визначення w і рівне Dm-1}
        k:=k*D;
            for i:=m+1 to n+1 do begin
                if w=v then {якщо числа рівні, то рядки співпадають} write In
             (‘УРА’); if i<=n then
                        w:=d*(w-k*T[i-m])+T[i]; { визначення нового значення w}
        end;
End.
```

Цей алгоритм виконує лінійне проходження по рядку **(т** кроків) і лінійне проходження по всьому тексту **(п** кроків), отже, спільний час роботи є 0 (**п+т**). Цей час лінійно залежить від розміру рядка і тексту, отже, програма працює швидко. Але який інтерес працювати тільки з короткими рядками і цифрами? Розробники алгоритму придумали, як поліпшити цей алгоритм без особливих втрат у швидкості роботи. Як ви помітили, ми ставили у відповідність рядку його числове подання, але виникала проблема великих чисел. Її можна уникнути, якщо проводити всі арифметичні дії з модулями якогось простого числа (постійно брати залишок від ділення на це число). Таким чином знаходиться не само число, що характеризує рядок, а його залишок від ділення на якесь просте число. Тепер ми ставимо число у відповідність не одному рядку, а цілому класу, але оскільки класів буде досить багато (стільки, скільки різних залишків від ділення на це просте число), то додаткову перевірку доведеться виконувати рідко.

```
V:=0;
w:=0;
        for i:=1 to m do { визначення v та w} begin
        v:=(v*D+ord(S[i])) mod P; {ord перетворює символ у число} w:=(w*D+ord(T[i])) mod
      P; end;
k:=1;
for i:=1 to m-1 do
            k:=k*D mod P; {k маємо значення Dm-1 mod P}
        for i:=m+1 to n+1 do begin
        if w=v then {якщо числа рівні, то рядки належать до одного класу, перевірка на
      співпадіння} begin J>0;
                        while (j<m) and (S[j+1]=T[i-m+j]) do j:=j+1;
                        if j=m then {кінцева перевірка}
                        writeln(‘yPA’);
                end;
        if i<=n then
                w:=(d*(w+P-(ord(T[i-m])*k mod P))+ord(T[i])) mod P;
            end.
```

Змістовий модуль 4. Алгоритми сортування. Загальна класифікація та принципи роботи. Жадібні алгоритми.

Тема 1. Алгоритми сортування основні поняття. Методи внутрішнього сортування. Метод простого включення, сортування шляхом підрахунку, метод Шелла, обмінне сортування,

сортування вибором.

9.1. МЕТОДИ ВНУТРІШНЬОГО СОРТУВАННЯ

У загальній постановці завдання сортування ставиться таким чином. Є послідовність однотипних записів, одне з полів яких вибране як ключове (далі ми називатимемо його ключем сортування). Тип даних ключа повинен включати операції порівняння («=», «>», «<», «>=» і «<=»). Завданням сортування є перетворення початкової послідовності в послідовність, що містить ті самі записи, але у порядку зростання (або спадання) значень ключа. Метод сортування називається стійким, якщо при його застосуванні не змінюється відносне положення записів із рівними значеннями ключа.

Розрізняють сортування масивів записів, розташованих в основній пам'яті (внутрішнє сортування), і сортування файлів, що зберігаються в зовнішній пам'яті і не вміщуються повністю в основній пам'яті (зовнішнє сортування). Для внутрішнього і зовнішнього сортування потрібні істотно різні методи.

Природною умовою, що висувається до будь-якого методу внутрішнього сортування є те, що ці методи не повинні вимагати додаткової пам'яті: всі перестановки

з метою впорядкування елементів масиву мають вироблятися в межах того ж масиву. Мірою ефективності алгоритму внутрішнього сортування є кількість необхідних порівнянь значень ключа (С) і кількість перестановок елементів **(М).**

Зазначимо, що оскільки сортування засноване тільки на значеннях ключа і ніяк не зачіпає поля записів, що залишилися, можна говорити про сортування масивів ключів.

9.1.1. Метод простого включення

Нехай є масив ключів a[1], a[2],..., a[n]. Для кожного елемента масиву, починаючи з другого, здійснюється порівняння з елементами з меншим індексом (елемент a[i] послідовно порівнюється з елементами a[i-1], a[i-2]...) і до тих пір, поки для чергового елемента a[j] виконується співвідношення a[j]> a[i], a[i] і a[j] міняються місцями. Якщо вдається зустріти такий елемент a[j], що a[j]<= *a*[*i*], або якщо досягнута нижня межа масиву, здійснюється перехід до опрацювання елемента a[/+1] (поки не буде досягнута верхня межа масиву) (рис.).

Можна побачити, що в кращому разі (коли масив вже впорядкований) для виконання алгоритму з масивом із п елементів буде треба n-1 порівняння і 0 пересилань. У гіршому разі (коли масив впорядкований у зворотному порядку) буде треба n*(n-1)/2 порівнянь і стільки ж пересилань.

Отже, можна оцінювати складність методу простих включень як $\Theta (n^2)$.

Можна скоротити кількість порівнянь, що використовуються в методі простих включень, якщо скористатися тим фактом, що при опрацюванні елемента масиву a[*i*] елементи a[1], a**[**2], ..., a[i-1] вже впорядковані, і скористатися для пошуку елементу, з яким має виконуватись перестановка, методом двійкового розподілу. У цьому випадку оцінка кількості необхідних порівнянь стає Θ (nlog n).

Зазначимо, що оскільки при виконанні перестановки потрібне зсування на один елемент декількох елементів, то оцінка кількості пересилань

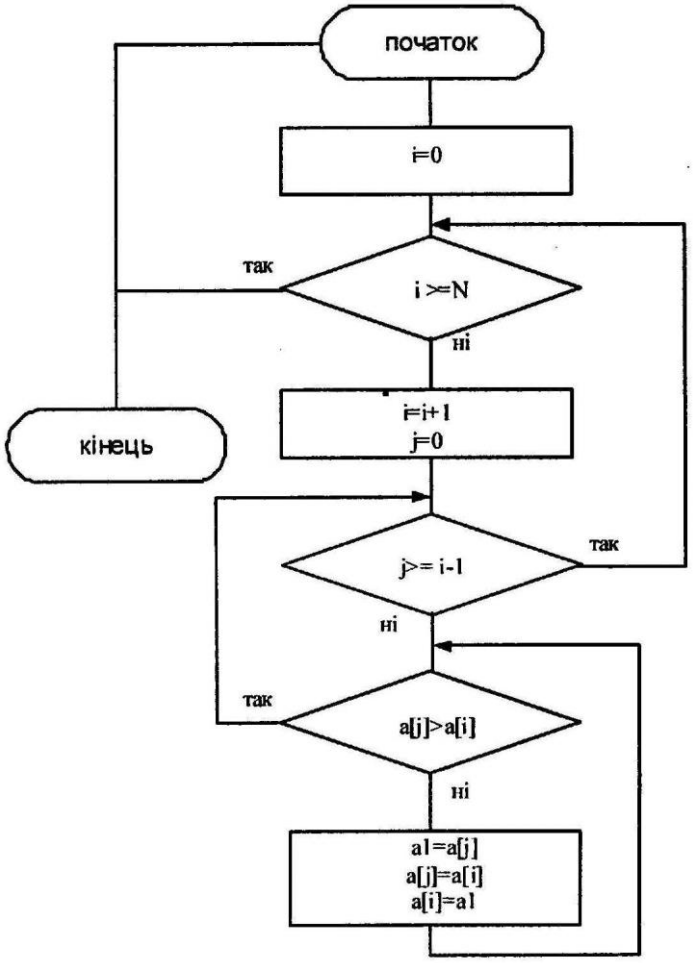


Рис. Блок-схема сортування методом включення

```
Void sort_vstavka()
{
    int;  int a[50], a1;
    int k[50];
    i=0;
    while(i<50)
    {
        for  (j=i-1; j>=0; i—)
        {
            if(a[j]>a[i])
            {
                a1=a[j];
                a[j]=a[i];
                    a[i]=a1; i—;
            }
        }
        i++;
    }
}
```

Таблиця Приклад сортування методом простого включення

Початковий стан масиву	8 23 5 65 44 33 1 6
Крок 1	8 23 5 65 44 33 1 6
Крок 2	8 5 23 65 44 33 1 6
	5 8 23 65 44 33 1 6
Крок 3	5 8 23 65 44 33 1 6
Крок 4	5 8 23 44 65 33 1 6
Крок 5	5 8 23 44 33 65 1 6
	5 8 23 33 44 65 1 6
Крок 6	5 8 23 33 44 1 65 6
	5 8 23 33 1 44 65 6
	5 8 23 1 33 44 65 6
	5 8 1 23 33 44 65 6
	5 18 23 33 44 65 6
	1 5 8 23 33 44 65 6
Крок 7	1 5 8 23 33 44 6 65
	1 5 8 23 33 6 44 65
	1 5 8 23 6 33 44 65
	15 8 6 23 33 44 65
	15 6 8 23 33 44 65

9.1.3. Сортування шляхом підрахунку

Кожний елемент порівнюється зі всіма іншими; остаточно положення елементів визначаються після підрахунку кількості менших ключів.

Треба знайти перестановку *p*(1)*p(2)...p(n),* таку, що

Kp(I)<=Kp(2)<=... <Kp(n).

Метод заснований на тому щоу'-кпюч впорядкованої послідовності перевищує рівно *j-1* інших ключів. Ідея полягає в тому, щоб зрівняти попарно всі ключі і підрахувати, скільки з них менші від кожного окремого ключа.

Спосіб виконання поставленого завдання такий:

((порівняти *Kj* з *Ki)* при 1 <=*j*<*i*) при 1<*i*<=*N*.

```
Void sort_pidr()
{
        inti.j; int a[50], a1[50]; int k[50];
            for (i=0; i<50; i++) k[i]=0;
        for (i=0; i<50-1; I++) for(j=i+1; j<50;
    j++) if(a[i]<aO])
                    ВД++;
        else k[j]++; for (i=0; i<50; i++)
    a1[k[i]]=a[i]; for (i=0; i<50; i++)
    a[i]=a1[i];
```

9.1.2. Метод Шелла

Подальшим розвитком методу сортування з включеннями є сортування методом Шелла, яке інакше називається сортуванням включеннями з відстанню, що зменшується.

Метод полягає в тому, що таблиця, яка впорядковується, розділяється на групи елементів, кожна з яких упорядковується методом простих включень. У процесі впорядкування розміри таких груп збільшуються доти, поки всі елементи таблиці не ввійдуть у впорядковану групу. Вибір чергової групи для сортування і її розташування всередині таблиці здійснюється так, щоб можна було використовувати попередню впорядкованість. Групою таблиці називають послідовність елементів, номери яких утворять арифметичну прогресію з різницею А (А називають кроком групи). На початку процесу впорядкування вибирається перший крок групи $A_{|3}$ що залежить від розміру таблиці. Шелл запропонував брати

h_j=[*n*/2], а h_i=*h*((*i*-1)/2).

У більш пізніх роботах Хіббард показав, що для прискорення процесу доцільно визначити крок h_1, за формулою:

h_1=2**k+1 , де 2**k < n <= 2**(k+1).

Після вибору h_1 методом простих включень впорядковуються групи, що містять елементи з номерами позицій i, $i+h_1$, $i+2*h_1$, ... , $i+m_i$,$*h_1$.

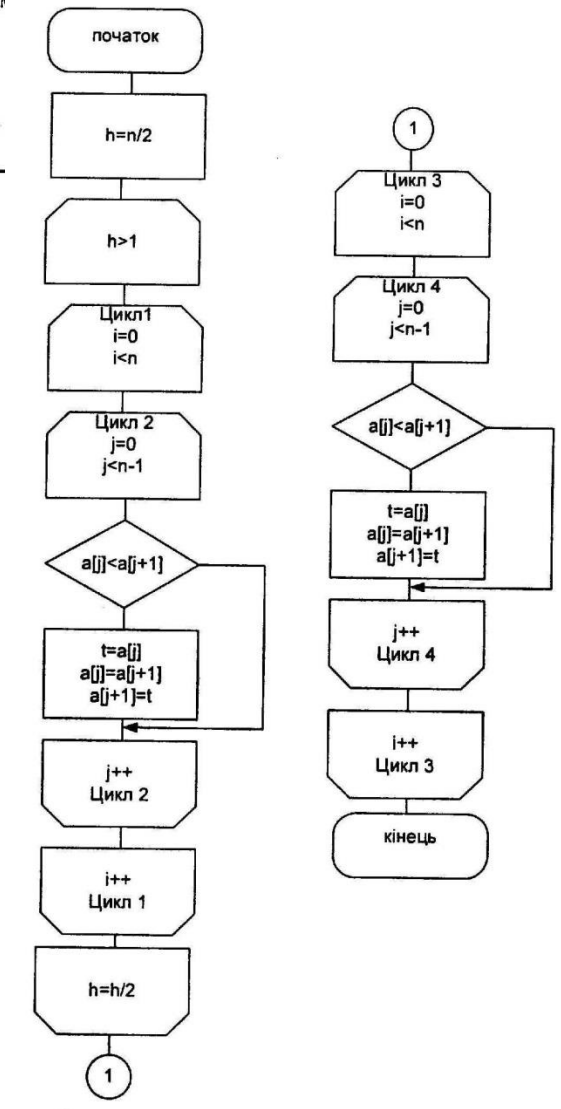
При цьому i= 1,2,...h_1; m[i] - найбільше ціле, що задовольняє нерівність $m[i]*h <= n$.

Потім вибирається крок $h_2<h_1$, і впорядковуються групи, що містять елементи з номерами позицій. Ця процедура з усе зменшуваними кроками продовжується доти, поки черговий крок $h[l]$ стане рівним одиниці ($h_1>h_2>.. .>h_n$,). Цей останній етап являє собою впорядкування всієї таблиці методом включень. Але оскільки вихідна таблиця впорядковувалася окремими групами з послідовним об'єднанням цих груп, то загальна кількість порівнянь значно менша, ніж n /4, необхідна при методі включень. Кількість операцій порівняння пропорційна $n*(\log_2(n))**2$.

Застосування методу Шелла до масиву, використаного в наших прикладах, показано в таблиці

Таблиця Приклад сортування методом Шелла

Початковий стан масиву	8 23 5 65 44 33 1 6
Фаза 1 (сортуються елементи, відстань між якими чотирьом)	8 23 5 65 44 33 1 6
	8 23 5 65 44 33 1 6
	8 23 1 65 44 33 5 6
	8 23 1 6 44 33 5 65
Фаза 2 (сортуються елементи, відстань між якими двом)	1 23 8 6 44 33 5 65
	1 23 8 6 44 33 5 65
	1 23 8 6 5 33 44 65
	1 23 5 6 8 33 44 65
	1 6 5 23 8 33 44 65
	1 6 5 23 8 33 44 65
	1 6 5 23 8 33 44 65
Фаза 3 (сортуються елементи, відстань між якими одном[illegible])	1 6 5 23 8 33 44 65
	[illegible] 44 65
	[illegible] 44 65
	[illegible] 44 65
	[illegible] 44 65
	[illegible] 44 65
	[illegible] 44 65



У загальному випадку алгоритм Шелла переформулюється для природно заданої послідовності з ї відстаней між елементами A,, A,, ..., A(, для яких виконуються умови A, = 1 і A(/+1)< A. При правильно підібраних складність алгоритму Шелла є 0 («(1.2)), що істотно менша за складність простих алгоритмів сортування. Блок-схема методу Шелла подана на рис. У ньому використано змінні: a[] - массив, який необхідно відсортувати, t - допоміжна змінна того ж типу, що і масив, n - розмірність масиву.

Рис. Блок-схема методу Шелла.

Тема 2. Сортування поділом (Хоара), за допомогою дерева, пірамідальне сортування, сортування злиттям, методи порозрядного сортування.

9.1.6. Сортування поділом (Хоара)

Метод сортування поділом був запропонований Чарльзом Хоаром 1962 року. Цей метод є розвитком методу простого обміну і настільки ефективний, що його почали називати «методом швидкого сортування - Quicksort».

Основна ідея алгоритму полягає в тому, що випадковим чином вибирається деякий елемент масиву x, після чого масив є видимим зліва, поки не зустрінеться елемент $a[i]$ такий, що $a[i]>x$**,** а потім масив є видимим справа, поки не зустрінеться елемент $a[j]$ такий, що $a[j]<x$. Ці два елементи міняються місцями, і процес перегляду, порівняння і обміну продовжується, поки ми не дійдемо до елемента **x.** У результаті масив виявиться розбитим на дві частини - ліву, в якій значення ключів будуть менші від **x,** і праву із значеннями ключів, більшими від x. Далі процес рекурсивно продовжується для лівої і правої частин масиву до тих пір, поки кожна частина не міститиме лише один елемент. Зрозуміло, що, як завжди, рекурсію можна замінити ітераціями, якщо запам'ятовувати відповідні індекси масиву. Прослідкуємо цей процес на прикладі нашого стандартного масиву

Таблиця Приклад швидкого сортування

Початковий стан масиву	8 23 5 65 |44| 33 1
Крок 1 (як x вибирається a[5])	8 23 5 6 4433 1 65
	8 23 5 6 1 33 44
Крок 2 (у підмасиві a[1], a[5] як x вибирається	8 23 |5| 6 1 33 44
	1 23 5 6 8 33 44

a[3])	1 5 23 6 8 33 44
Крок 3 (у підмасиві a[3], a[5] як x вибирається a[4])	1 5 23 |6| 8 33 44
	1 5 8 6 23 33 44
Крок 4 (у підмасивах a[3], a[4] вибирається a[4])	1 5 8 |6| 23 33 44
	1 5 6 8 23 33 44

Алгоритм недаремно називається швидким сортуванням, оскільки для нього оцінкою кількості порівнянь і обмінів є Θ (*n* log *n*)**.** Насправді, в більшості утиліт, що виконують сортування масивів, використовується саме цей алгоритм.

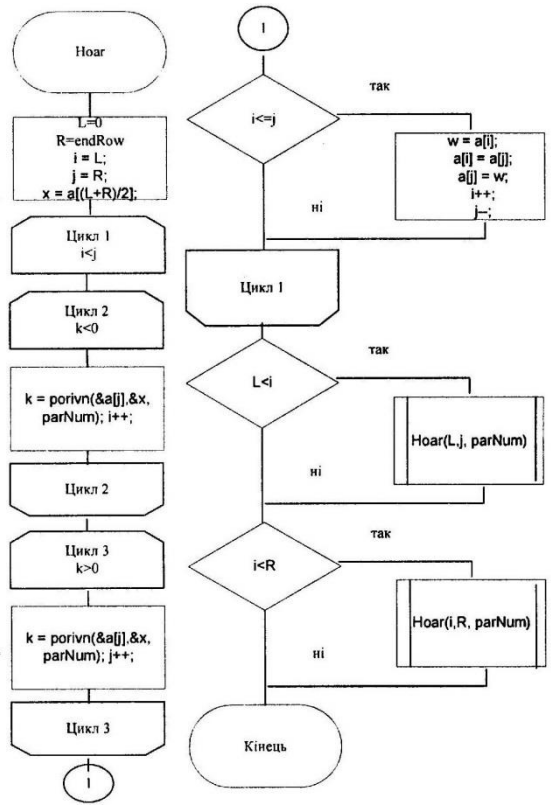


Рис. Блок-схема рекурсивного методу Хоара

18.1.7. Сортування за допомогою дерева

Почнемо з простого методу сортування за допомогою дерева, при використанні якого будується двійкове дерево порівняння ключів, Побудова дерева починається з листя, яке містить всі елементи масиву. З кожної сусідньої пари вибирається найменший елемент, і ці елементи утворюють наступний (ближчий до кореня рівень дерева). Із кожної сусідньої пари вибирається найменший елемент і т.ін., поки не буде побудований корінь, тобто найменший елемент масиву.

Приклад двійкового дерева показано на рис. 9.5.

Отже, ми вже маємо якнайменше значення елементів масиву. Щоб одержати наступний по величині елемент, спустимося від коріння по шляху; що веде до листка з якнайменшим значенням.

У цій листковій вершині ставиться фіктивний ключ з «нескінченно великим» значенням, а у всі проміжні вузли, що займалися якнайменшим елементом, вноситься якнайменше значення з вузлів - безпосередніх нащадків (рис. 9.6). Процес продовжується доти, поки всі вузли дерева не будуть заповнені фіктивними ключами.

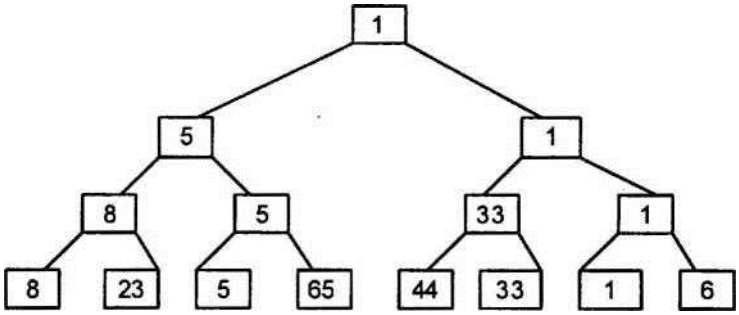


Рис. Перший крок

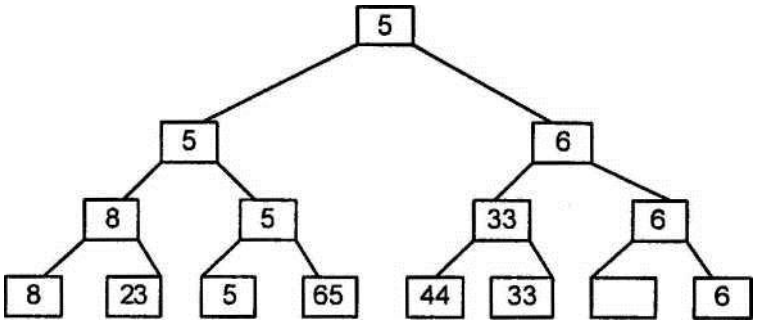


Рис. Другий крок

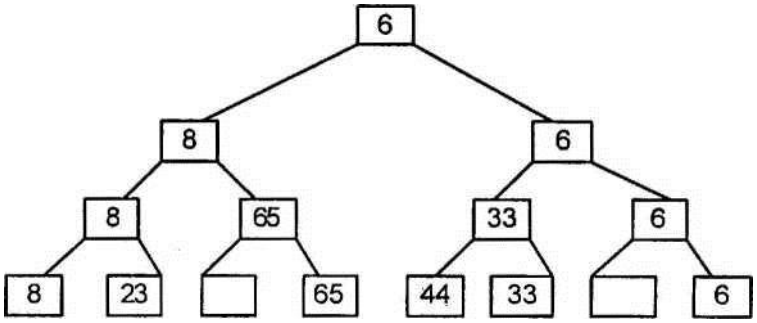


Рис. Третій крок

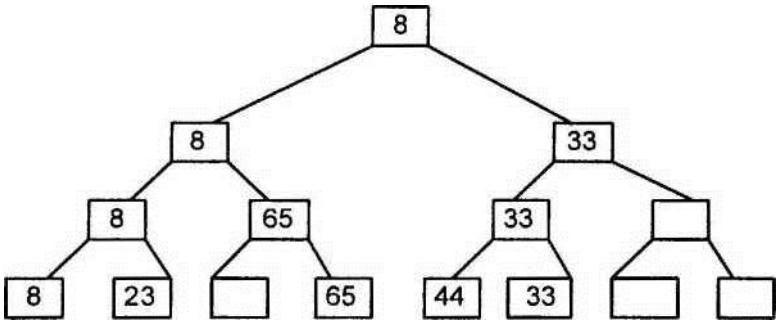


Рис. Четвертий крок

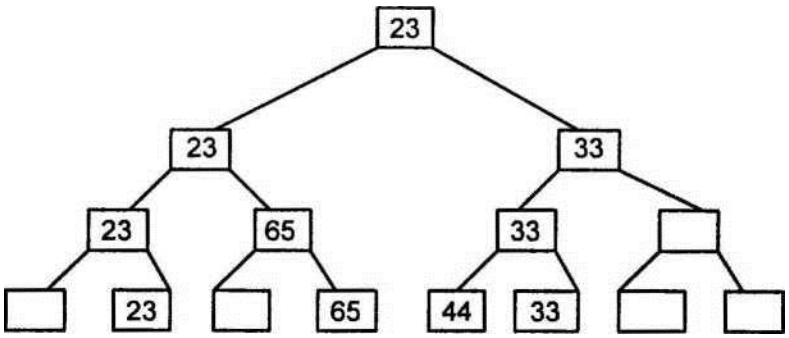


Рис. П'ятий крок

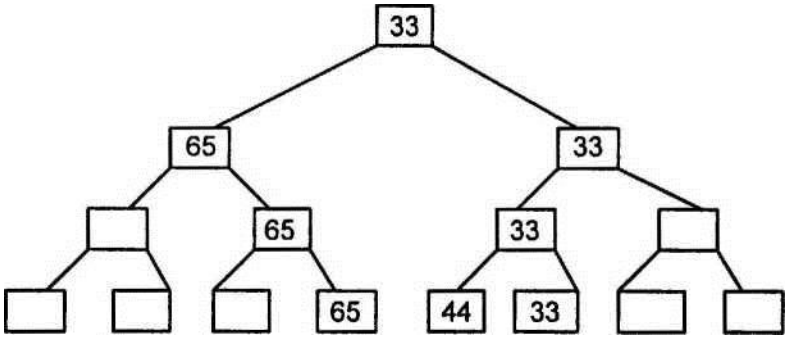


Рис. Шостий крок

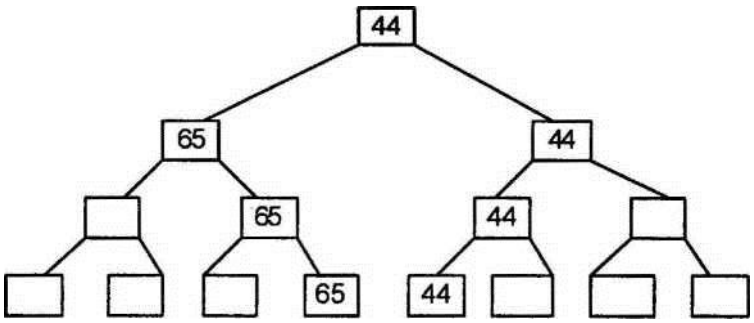


Рис. Сьомий крок

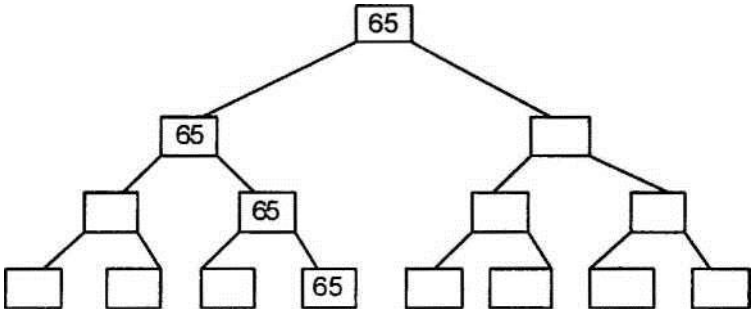


Рис. Восьмий крок

9.1.8. Пірамідальне сортування

Є досконаліший алгоритм, який прийнято називати пірамідальним сортуванням (НеарБоїТ). Його ідея полягає у тому, що замість повного дерева порівняння початковий масив а[1], а[*2i*], ..., а[*n*] перетвориться на піраміду, що має таку властивість, що для кожного *а*[*i*] виконуються умови *а*[*i*]<= *а*[*2i*] і а[i]<= а[2i+1]. Потім піраміда використовується для сортування.

Найнаочніше метод побудови піраміди виглядає при деревовидному зображенні масиву, показаному на рис. 9. ІЗ. Масив подається у вигляді двійкового дерева, корінь якого відповідає елементу масиву л[1]. На другому ярусі знаходяться елементи *а*[*2*] і *а*[3]. На третьому - [4], а[*5*], а[*6*], а[7] і т.ін. Як видно, для масиву з непарною кількістю елементів відповідне дерево буде збалансованим, а для масиву з парною кількістю елементів *n* елемент *a[n]* буде єдиним (найлівішим) листком «майже» збалансованого дерева.

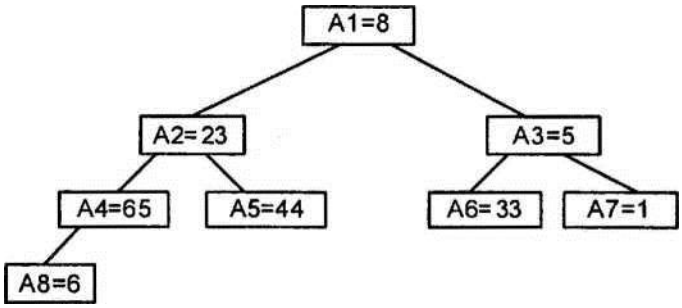


Рис. Приклад пірамідального сортування

Очевидно, що при побудові піраміди нас цікавитимуть елементи a[*n*/2], *a*[*n*/2-1], ..., *a*[1] для масивів з парною кількістю елементів і елементи *a*[(*n*-1)/2], *a*[(*n*-1)/2-1], ..., *a*[1] для масивів з непарною кількістю елементів (оскільки тільки для таких елементів істотні обмеження піраміди). Нехай *i* - найбільший індекс з індексів елементів, для яких існують обмеження піраміди. Тоді береться елемент a[i] у побудованому дереві і для нього виконується процедура просівання, що полягає у тому, що вибирається гілка дерева, яка відповідає min(a[2*i*], a[2*i*+1]), і значення a[i] міняється місцями із значенням відповідного елементу'. Якщо цей елемент не є листком дерева, для нього виконується аналогічна процедура і т.ін. Такі дії виконуються послідовно для a[j], a[i-1], ..., a[1]. Як бачимо, в результаті ми одержимо деревовидне подання піраміди для початкового масиву (послідовність кроків для використовуваного в наших прикладах масиву показана на рис.

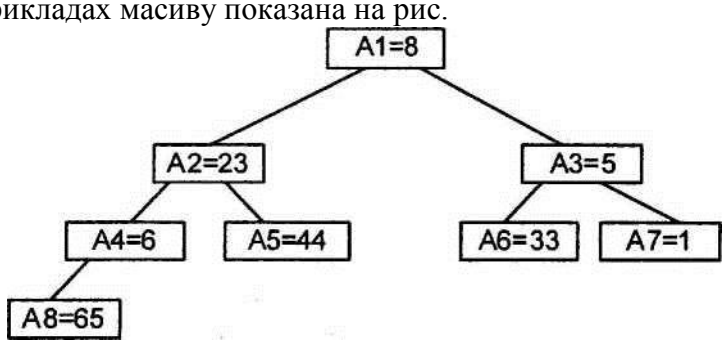


Рис. Пірамідальне сортування. Крок 1.

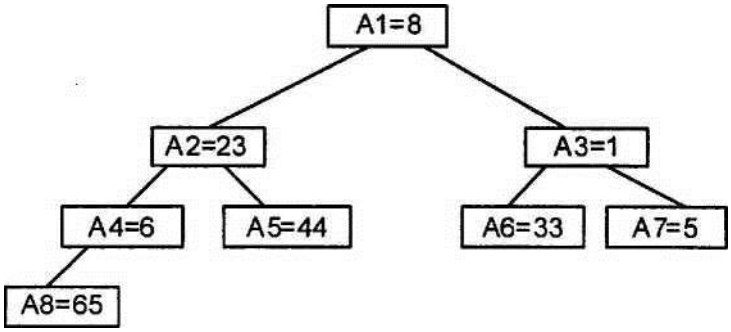


Рис. Пірамідальне сортування. Крок 2.

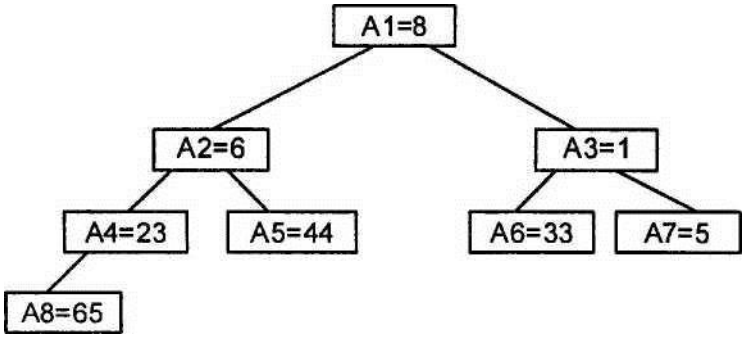


Рис. Пірамідальне сортування. Крок 3.

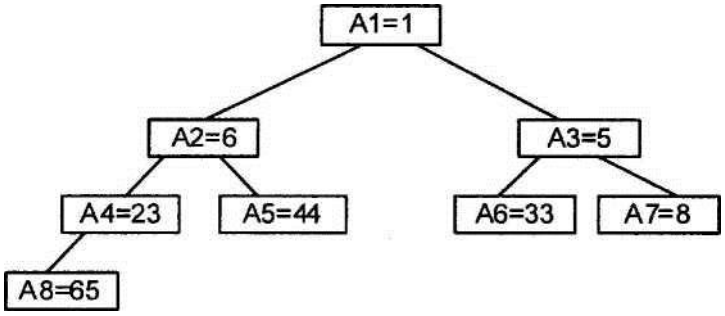


Рис. Пірамідальне сортування. Крок 4.

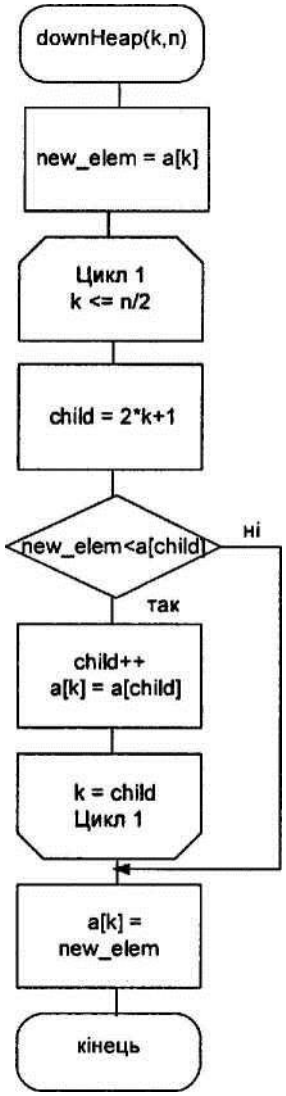


Рис. Блок-схема методу впорядкування піраміди

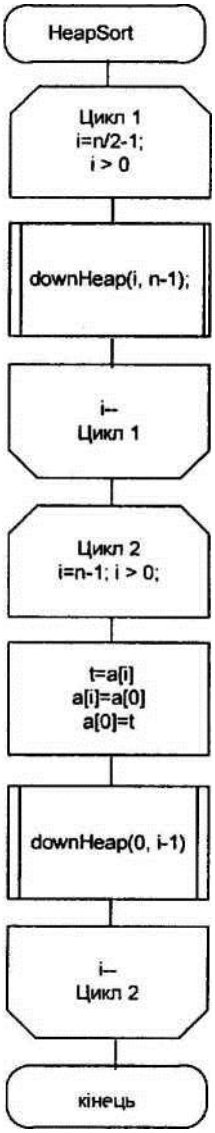


Рис. Блок-схема побудови піраміди та її перевірки

9.1.9. Побудова піраміди методом Флойда

1964 року Флойд запропонував метод побудови піраміди без явної побудови дерева (хоча метод заснований на тих самих ідеях). Побудова піраміди методом Флойда для нашого стандартного масиву показана в таблиці 9.7.

Таблиця 9.7 Приклад побудови піраміди.

Початковий стан масиву	8 23 5 |65| 44 33 1 6
Крок 1	8 23 |5| 6 44 33 1 65
Крок 2	8 |23| 1 6 44 33 5 65
Крок 3	|8| 6 1 23 44 33 5 65
Крок 4	1 6 8 23 44 33 5 65
	1 6 5 23 44 33 8 65

У таблиці 9.8 показано, як здійснюється сортування з використанням побудованої піраміди. Суть алгоритму полягає в подальшому. Нехай /'-найбільший індекс масиву, для якого вказані умови піраміди. Тоді, починаючи з я[1] до *a[i]* виконуються такі дії.

Таблиця 9.8 Сортування за допомогою піраміди.

Початкова піраміда	1 6 5 23 44 33 8 65
Крок 1	65 6 5 23 44 33 8 1
	5 6 65 23 44 33 8 1
	5 6 8 23 44 33 65 1
Крок 2	65 6 8 23 44 33 5 1 !
	6 65 8 23 44 33 5 1
	6 23 8 65 44 33 5 1
КрокЗ	33 23 8 65 44 6 5 1
	8 23 33 65 44 6 5 1
Крок 4	44 23 33 65 8 6 5 1
	23 44 33 65 8 6 5 1
Крок 5	65 44 33 23 8 6 5 1
	33 44 65 23 8 6 5 1
Крок 6	65 44 33 23 8 6 5 1
	44 65 33 23 8 6 5 1
Крок 7	65 44 33 23 8 6 5 1

На кожному кроці вибирається останній елемент піраміди (у нашому випадку першим буде вибраний елемент *a*[8]. Його значення міняється зі значенням *a*[1], після чого для *a[* 1] виконується просіювання. При цьому на кожному кроці кількість елементів в піраміді зменшується на 1 (після першого кроку як елементи піраміди розглядаються *a*[1], a[*2*], ..., a[n-1]; після другого - a[1], a[*2*], ..., a[n-2] і т.ін., поки в піраміді не залишиться один елемент). Легко побачити (це ілюструється в таблиці), що як результат ми одержимо масив, впорядкований у порядку спадання. Можна модифікувати метод побудови піраміди і сортування, щоб одержати впорядкування у порядку зростання, якщо змінити умову піраміди *a*[*i*]>*a*[2*i*] і *a*[1]>=a[2*i*+1] для всіх значень індекса.

Тема 3. Методи зовнішнього сортування. Пряме злиття, природне злиття, збалансоване багатошляхове злиття, багатофазне сортування.

9.2. МЕТОДИ ЗОВНІШНЬОГО СОРТУВАННЯ

Прийнято називати «зовнішнім» сортування послідовних файлів, розташованих у зовнішній пам’яті і дуже великих, щоб можна було повністю перемістити їх в основну пам’ять і застосувати один з розглянутих у попередньому розділі методів внутрішнього сортування. Найчастіше зовнішнє сортування застосовується в системах керування базами даних при виконанні запитів, і від ефективності вживаних методів істотно залежить продуктивність СУБД.

Мусимо пояснити, чому йдеться саме про послідовні файли, тобто про файли, які можна

читати запис за записом в послідовному режимі, а писати можна тільки після останнього запису. Методи зовнішнього сортування з'явилися, коли найбільш поширеними пристроями зовнішньої пам'яті були магнітні стрічки. Для стрічок послідовний доступ був абсолютно природним. Коли відбувся перехід до пристроїв, що запам'ятовують, з магнітними дисками, що забезпечують «прямий» доступ до буць- якого блоку інформації, здавалося, що послідовні файли втратили свою актуальність. Проте це припущення було помилковим.

Вся річ у тому, що практично всі використовувані на сьогодні дискові пристрої забезпечені рухомими магнітними головками. При виконанні обміну з дисковим накопичувачем виконується підведення головок до потрібного циліндра, вибір потрібної головки (доріжки), прокручування дискового пакету до початку необхідного блоку і, нарешті, читання або запис блоку. Серед всіх цих дій найбільший час займає підведення головок. Саме цей час визначає загальний час виконання операції. Єдиним доступним прийомом оптимізації доступу до магнітних дисків є якомога «ближче» розташу вання файла на накопичувані блоків, що послідовно адресуються. Але і в цьому випадку рух головок буде мінімізованим тільки у тому випадку, коли файл читається або пишеться в послідовному режимі. Саме з такими файлами при потребі сортування працюють сучасні СУБД.

Зазначимо, що насправді швидкість виконання зовнішнього сортування залежить від розміру буфера (або буферів) основної пам'яті, яка може бути використана для цих цілей.

9.2.1. Пряме злиття

Припустимо, що є послідовний файл ***A,*** що складається із записів ***a***\, я2,..., ***an*** (знову для простоти припустимо, що ***n*** є ступенем числа 2). Вважатимемо, що кожен запис складається лише з одного елементу, що є ключем сортування. Для сортування використовуються два допоміжні файли В і С (розмір кожного з них буде ***n/***2).

Сортування складається з послідовності кроків, в кожному з яких виконується розподіл стану файлу ***A*** у файли ***B*** і ***C,*** а потім злиття файлів ***B*** і ***C*** у файл ***A.*** (Помітимо, що процедура злиття для файлів повністю ілюструється рисунком 9.20.) На першому кроці для розподілу послідовно читається файл Л, і записи ***a\, a3,..., a{n***-1) пишуться у файл ***B,*** а записи ***a2, a4, ..., an*** - у файл ***C*** (початковий розподіл). Початкове злиття здійснюється над парами ***(a***I, ***a2), (a3,*** <я4), ..., (я(л-I), ***an),*** і результат записується у файл ***A.*** На другому кроці знову послідовно читається файл ***A,*** і у файл ***B*** записуються послідовні пари з непарними номерами, а у файл ***C***- з парними. При злитті утворюються і пишуться у файл ***A*** впорядковані четвірки записів. І так далі. Перед виконанням останнього кроку файл ***A*** міститиме дві впорядковані підпослідовності розміром «/2 кожна. При розподілі перша з них потрапить у файл ***B***, а друга - у файл ***C.*** Після злиття файл ***A*** міститиме повністю впорядковану послідовність записів. У таблиці 9.10 показаний приклад зовнішнього сортування простим злиттям. Зазначимо, що для виконання зовнішнього сортування методом прямого злиття в основній пам'яті вимагається розташувати всього дві змінні - для розміщення чергових записів з файлів ***B*** і С.

Таблиця Приклад зовнішнього сортування прямим злиттям

Початковий стан	8 23 5 65 44 33 1 6
Перший крок Розподіл	
Файл В	8 5 44 1
Файл С	23 65 33 6
Злиття: файл А	8 23 5 65 33 44 1 6
Другий крок Розподіл	
Файл В	8 23 33 44
Файл С	5 65 1 6
Злиття: файл А	5 8 23 65 1 6 33 44
Третій крок	

Розподіл	
Файл В	5 8 23 65
Файл С	1 6 33 44
Злиття: файл А	1 5 6 8 23 33 44 65

9.2.2. Природне злиття

При використанні методу прямого злиття не береться до уваги те, що початковий файл може бути частково відсортованим, тобто містити впорядковані підпослідовності записів. **Серією** називається підпослідовність записів **ai, a(i**+1), ..., **a]** така, що **ак** <= **а(к**+1) для всіх / <= **к** <у, **ai < a(i-**1) і о/>я(/+]). Метод природного злиття ґрунтується на розпізнаванні серій при розподілі і їх використанні при подальшому злитті.

Як і у разі прямого злиття, сортування виконується за декілька кроків, в кожному з яких спочатку виконується розподіл файла **А** по файлам **В** і С, а потім злиття **В** і **С у** файл **А.** При розподілі розпізнається перша серія записів і переписується у файл **В,** друга — у файл **С** і т.ін. При злитті перша серія записів файлу **В** зливається з першою серією файлу С, друга серія **В** з другою серією **С** і т.ін. Якщо перегляд одного файлу закінчується раніше, ніж перегляд іншого (внаслідок різної кількості серій), то залишок не до кінця переглянутого файла повністю копіюється в кінець файла **А.** Процес завершується, коли у файл і Я залишається тільки одна серія. Приклад сортування файла показаний на рис

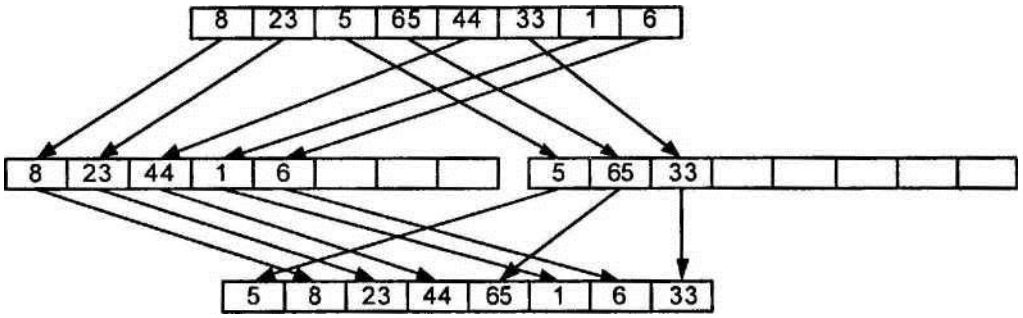


Рис. Зовнішнє пряме сортування злиттям. Перший крок.

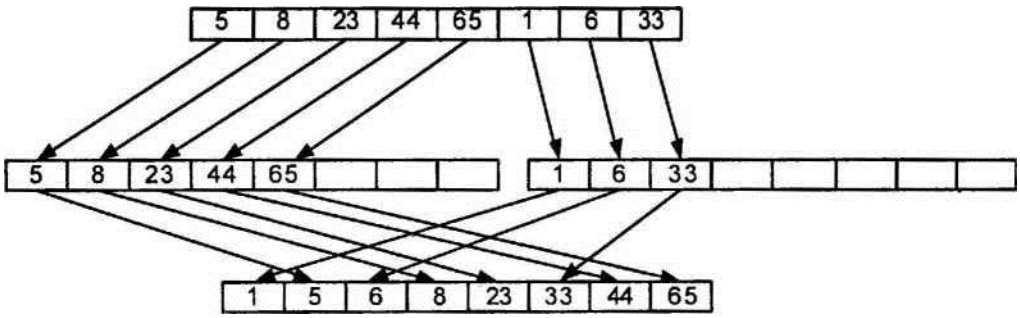


Рис. Зовнішнє пряме сортування злиттям. Другий крок.

Очевидно, що кількість зчтувань перезаписів файлів при використанні цього методу буде не більша, ніж при застосуванні методу прямого злиття, а в середньому - менша. З другого боку, збільшується кількість порівнянь за рахунок тих, які потрібні для розпізнавання кінців серій. Крім того, оскільки довжина серій може бути довільною, то максимальний розмір файлів *В і С* може бути близький до розміру файла *А***.**

9.2.3. Збалансоване багатошляхове злиття

В основі методу зовнішнього сортування збалансованим багатошляховим злиттям є розподіл серій початкового файлу по *т* допоміжних файлів 51, 52, ..., *Вт* і їх злиття в *т* допоміжних файлів CI, С2, ..., *Ст.* На наступному кроці здійснюється злиття файлів CI, С2, ..., *Ст* у файли

51,52, ..*Bm* і т.ін., поки в 51 або CI не утворюється одна серія.

Багатошляхове злиття є природним розвитком ідеї звичного (двошляхового) злиття, ілюстрованого рис. . Приклад тришляхового злиття показаний на рис. .

На рис. показаний простий приклад застосування сортування багато шляховим (багатофазним) злиттям.

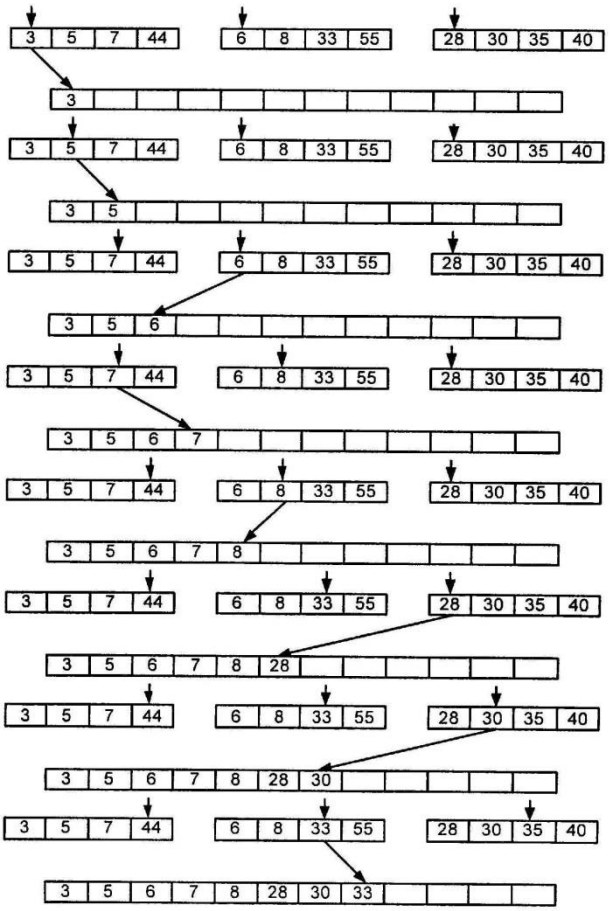


Рис. а) Тришляхове злиття. Початок

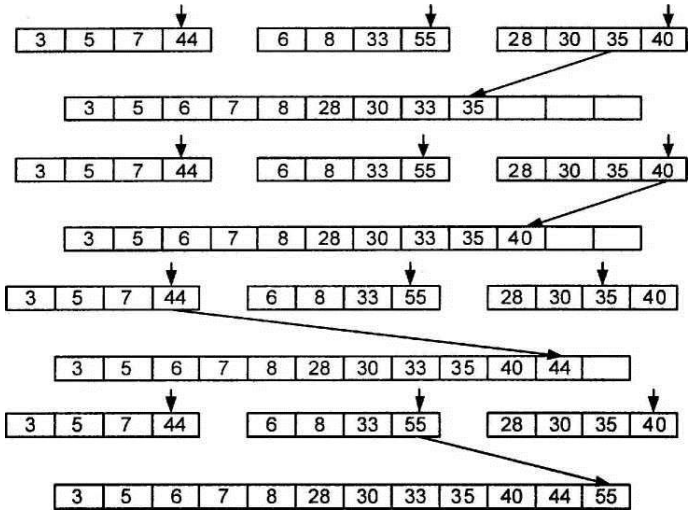


Рис. *Рис. б) Тришляхове злиття. .*

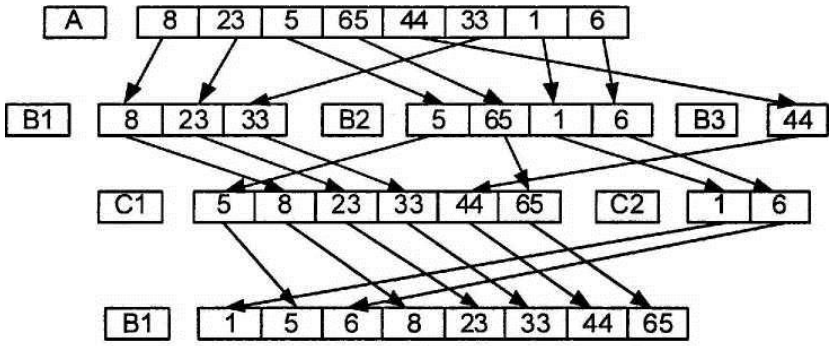


Рис. 9.25. Багатофазне злиття.

Він, зазвичай, дуже тривіальний, щоб продемонструвати декілька кроків виконання алгоритму, проте достатній як ілюстрація загальної ідеї методу. Як показує цей приклад, у міру збільшення довжини серій допоміжні файли з великими номерами (починаючи з номера я) перестають використовуватися, оскільки їм «не дістається» жодної серії. Перевагою сортування збалансованим багатофазним злиттям є те, що кількість проходжень алгоритму оцінюється як 0 (log я) (я - кількість записів у початковому файлі), де логарифм береться по я. Порядок кількості копіювань записів - (log я). Зазвичай, кількість порівнянь не буде меншою, ніж при застосуванні методу простого злиття.

9.2.4. Багатофазне сортування

При використанні розглянутого вище методу збалансованого багатошляхового зовнішнього сортування на кожному кроці приблизно половина допоміжних файлів використовується для введення даних і приблизно стільки ж для виведення злитих серій. Ідея багатофазного сортування полягає у тому, що з наявних т допоміжних файлів (т-1) файл служить для введення злитих послідовностей, а один - для виведення утворюваних серій. Як тільки один з файлів введення стає порожнім, його починають використовувати для виведення серій, одержуваних при злитті серій нового набору (ти-1) файлів. Отже, є перший крок, при якому серії початкового файлу розподіляються по ти-1 допоміжному файлу, а потім виконується багатошляхове злиття серій з (ти-1) файлу, поки в одному з них не утворюється одна серія.

Очевидно, що при довільному початковому розподілі серій по допоміжним файлам алгоритм може не зійтися, оскільки в єдиному непорожньому файлі існуватиме більше, ніж одна серія. Припустимо, наприклад, що використовується три файли 51,52 і 53, і при початковому розподілі у файл 51 вміщені 10 серій, а у файл 52 - 6. При злитті 51 і 52 до моменту, коли ми

дійдемо до кінця 52, в 51 залишаться 4 серії, а у 53 потраплять 6 серій. Продовжиться злиття 51 і 53, і при завершенні перегляду 51 у 52 міститимуться 4 серії, а у 53 залишаться 2 серії. Після злиття 52 і 53 в кожному із файлів 51 і 52 міститиметься по 2 серії, яка злитиметься і утворюють 2 серії в 53 при тому, що 51 і 52 - порожні. Отже, алгоритм не зійшовся (таблиця).

Таблиця Приклад початкового розподілу серій, при якому трифазне зовнішнє сортування не приводить до требаго результату.

К-сть серій у файлі	К-сть серій у файлі В2	К-сть серій у файлі
10	6	0
4	0	6
0	4	2
2	2	0
0	0	2

Оскільки кількість серій у початковому файлі може не забезпечувати можливість такого розподілу серій, застосовується метод додавання порожніх серій, які надалі якомога рівномірніше розподіляються між проміжними файлами і пізнаються при подальшому злитті. Зрозуміло, що чим менше таких порожніх серій, тобто чим ближча кількість початкових серій за вимогами Фібоначчі, тим ефективніше працює алгоритм.

Тема 4. Поняття жадібного алгоритму. Відмінність між динамічним програмуванням і жадібним алгоритмом. Алгоритми: Краскала, Шеннона-Фано, Хафмана, Пріма.

ПОНЯТТЯ ЖАДІБНОГО АЛГОРИТМУ

❖ *Жадібний алгоритм - метод оптимізації задач, заснований на тому, що процес ухвалення рішення можна розбити на елементарні кроки, на кожному з яких приймається окреме рішення.*

Рішення, яке приймається на кожному кроці, має бути оптимальним тільки на поточному кроці і прийматися без урахування попередніх або подальших рішень.

Ознаки того, що задачу можна розв’язати за допомогою жадібного алгоритму:

4. задачу можна розбити на підзадачі;
5. величини, що розглядаються в задачі, можна дробити так само, як і задачу, на елементарніші;
6. сума оптимальних рішень для двох підзадач дасть оптимальне рішення для всієї задачі.

Приклад жадібного алгоритму:

```
к:=1;
v[k]:=1;
kol_ver[1 ]:=kol_ver[1 ]+1; s:=0;
while (k<n) do begin
     min:=maxint; for i:=1 to к do for
 j:=1 to n do
                  if a[v[i]j]<min then begin
                      min:=a[v[i],]];
                      vi:=v[i];
                      vj:=j;
end;
        f:=true;
        for i:=1 to к do if vj=v[i] then
     f:=false; iff then begin k:=k+1;
```

```
        v[k]:=vj;
                kol_ver[vj]:=kol_ver[vj]+1;
            kol_ver[vi]:=kol_ver[vi]+1;       s:=s+a[vi,vj];
            writeln(v[k],'<vi>',vj); readln; end;
        a[vi,vj]:=maxint;
        a[vj,vi]:=maxint;
end;
```

Загалом неможливо сказати, чи можна отримати оптимальне рішення за допомогою жадібного алгоритму стосовно конкретного завдання. Але є дві особливості, характерні для завдань, які вирішуються за допомогою жадібних алгоритмів: принцип жадібного вибору і властивість оптимальності для підзадач.

Говорять, що до завдання оптимізації застосуємо принцип ***жадібного вибору,*** якщо послідовність локально оптимальних виборів дає глобально оптимальне рішення. У цьому полягає головна відмінність жадібних алгоритмів від динамічного програмування: у другому прораховуються відразу наслідки всіх варіантів.

Щоб довести, що жадібний алгоритм дає оптимум, треба спробувати виконати доведення, яке аналогічне до доведення алгоритму задачі про вибір заявок. Спочатку ми показуємо, що жадібний вибір на першому кроці не закриває шлях до оптимального розв'язання: для будь-якого розв'язку є інший варіант, узгоджений з жадібним вибором і не гірший від першого. Потім ми показуємо, що розв'язок, який виник після жадібного вибору на першому кроці, аналогічний до початкового. За індукцією витікатиме, що така послідовність жадібних виборів дає оптимальний розв'язок.

Оптимальність для підзадач

Ця властивість говорить про те, що оптимальний розв'язок всієї задачі містить у собі оптимальні розв'язки підзадач. Наприклад, у задачі про вибір заявок можна помітити, що, якщоЛ — оптимальний набір заявок, що містить заявку номер 1, то *A1=A\{1}* — оптимальний набір заявок для меншої множини заявок 51, що складається з тих заявок, для яких si є *f1.*

Оптимальне розв'язування будується покроково. На кожному кроці часткове розв'язування доповнюється новим елементом, який обирається з допустимої підмножини. Для цього необхідні три процедури: процедура вибору кандидата для розширення часткового розв'язку, процедура визначення допустимості цього кандидата, процедура визначення, чи є цей розширений частковий розв'язок повним розв'язком задачі.

ВІДМІННІСТЬ МІЖ ДИНАМІЧНИМ ПРОГРАМУВАННЯМ І ЖАДІБНИМ АЛГОРИТМОМ

Відмінність між динамічним програмуванням і жадібним алгоритмом можна проілюструвати на прикладі завдання про рюкзак, а точніше, на його дискретному і неперервному формулюванні. Далі ми покажемо, що неперервне завдання вирішується жадібним методом, дискретне ж вимагає тоншого, динамічнішого рішення.

Дискретне завдання про рюкзак. Злодій заліз на склад, на якому зберігається **п** речей. Кожна річ коштує V/ доларів і важить **т** кілограмів. Злодій хоче понести товару на максимальну суму, проте він не може підняти більш **IV** кілограмів (всі числа цілі). Що він може покласти в рюкзак?

Неперервне завдання про рюкзак. Тепер злодій уміє дробити товари і укладати в рюкзак тільки їх частки, а не обов'язково ціле. Зазвичай в дискретному завданні йде мова про золоті злитки різної проби, а в неперервному — про золотий пісок.

2. ПРИКЛАДИ ЖАДІБНИХ АЛГОРИТМІВ 10.3.1. Алгоритм Краскала

Алгоритм Краскала знаходить контурний ліс мінімальної ваги у заданому графі.

Спочатку опрацьована множина ребер встановлюється порожньою. Потім, доки це можливо, виконується така операція: зі всіх ребер, додавання яких до вже наявної множини не викличе появу в ньому циклу, вибирається ребро мінімальної ваги і додається до множини опрацьованих ребер. Коли таких ребер більше немає, алгоритм завершений. Підграф графу, що

містить усі його вершини і знайдену множину ребер, є його контурним лісом мінімальної ваги.

До початку роботи алгоритму необхідно відсортувати ребра за вагою, це вимагає 0 **(E \ogiE))** часу. Після цього компоненти зв'язності зручно зберігати у вигляді системи непересічної множини. Всі операції у такому разі триватимуть **(E (E, V)).**

10.3.2. Алгоритм Шеннона-Фано

Алгоритм Шеннона-Фано — один з перших алгоритмів стискання, який вперше сформулювали американські вчені Шеннон і Фано. Алгоритм використовує коди змінної довжини: символ, що часто зустрічається, позначається кодом меншої довжини, а символ, що рідко зустрічається — кодом більшої довжини. Коди Шеннона-Фано префіксні, тобто, ніяке кодове слово не є префіксом будь-якого іншого. Ця властивість дозволяє однозначно декодувати будь-яку послідовність кодових слів.

Алгоритм обчислення кодів Шеннона-Фано

Код Шеннона-Фано будується за допомогою дерева. Побудова цього дерева починається з кореня. Вся множина кодованих елементів відповідає кореню дерева (вершині першого рівня). Вона розбивається на дві підмножини з приблизно однаковими сумарними ймовірностями. Ці підмножини відповідають двом вершинам другого рівня, які з'єднуються з коренем. Далі кожна з цих підмножин розбивається на дві підмножини з приблизно однаковою сумарною ймовірністю. їм відповідають вершини третього рівня. Якщо підмножина містить єдиний елемент, то йому відповідає кінцева вершина кодового дерева; така підмножина розбиттю не підлягає. Так само чинимо доти, поки не отримаємо всі кінцеві вершини. Гілки кодового дерева позначаємо символами 1 і 0. При побудові коду Шеннона-Фано розбиття множини елементів може бути виконане декількома способами. Вибір розбиття на рівні п може погіршити варіанти розбиття на наступному рівні (н+1) і привести до погіршення коду загалом. Іншими словами, оптимальна поведінка на кожному кроці шляху ще не гарантує оптимальності всієї сукупності дій. Тому код Шеннона-Фано не є оптимальним у загальному сенсі, хоч і дає оптимальні результати при деяких розподілах ймовірності. Для одного і того ж розподілу ймовірності можна побудувати, взагалі кажучи, декілька кодів Шеннона-Фано, і всі вони можуть дати різні результати.

Якщо побудувати всі можливі коди Шеннона-Фано для заданого розподілу ймовірності, то серед них будуть і всі оптимальні коди.

10.3.4. Алгоритм Хафмана

Алгоритм Хафмана (англ. Нийтпап) — адаптивний жадібний алгоритм оптимального префіксного кодування алфавіту з мінімальною надмірністю. Був розроблений 1952 року доктором Массачусетського технологічного інституту Девідом Хафманом. На сьогодні використовується в багатьох програмах стискання даних.

На відміну від алгоритму Шеннона-Фано, алгоритм Хафмана залишається завжди оптимальним і для вторинних алфавітів $\mathbf{т_2}$ з більше ніж двома символами.

Цей метод кодування складається з двох основних етапів:

3) побудова оптимального кодового дерева.

4) побудова відображення код->символ на основі побудованого дерева.

Алгоритм

3. Визначається ймовірність появи символів первинного алфавіту в початковому тексті (якщо вони не задані заздалегідь).

4. Символи первинного алфавіту /лі виписують у порядку зменшення ймовірності.

3.Останні $\mathbf{п_0}$ символів об'єднують у новий символ, ймовірність якого дорівнює

сумарній ймовірності цих символів, видаляють ці символи і вставляють новий символ у список останніх на відповідне місце (за ймовірністю). визначається із системи:

де **а**—ціле число, **т**\ і /и, — потужність первинного і вторинного алфавіту відповідно.

6. Останні $\mathbf{т_2}$ символів знову об'єднують в один і вставляють його на відповідну позицію, заздалегідь видаливши символи, що увійшли до об'єднання.

7. Попередній крок повторюють доти, доки сума всіх $\mathbf{т_2}$ символів не стане рівною 1.

Цей процес можна подати як побудову дерева, корінь якого — символ із ймовірністю

1, який отримано при об'єднанні символів з останнього кроку, його $\mathbf{т_2}$ нащадків — символи з попереднього кроку і так далі.

Кожні $\mathbf{т_2}$ елементів, що розташовані на одному рівні, нумеруються від 0 до $\mathbf{т_2}$ -1. Коди отримують зі шляхів (від першого нащадка кореня і до листка). При декодуванні можна використовувати те ж саме дерево, прочитується по одній цифрі і робиться крок по дереві, доки не досягається листка — годі виводиться символ, що стоїть у листку і відбувається повернення в корінь.

Кодування Шеннона-Фано є досить старим методом стискання, і на сьогодні воно не має особливого практичного застосування. У більшості випадків довжина стисненої послідовності за заданим методом дорівнює довжині стисненої послідовності з використанням кодування Хафмана. Але на деяких послідовностях все ж формуються неоптимальні коди Шеннона-Фано, тому стискання методом Хафмана прийнято вважати ефективнішим.

Приклад реалізації

Приклад реалізації алгоритму Хафмана на мові C++ (замість впорядковування піддерев кожного разу шукаємо в масиві дерево з мінімальною вагою) (текст програми взято з Вікіпедії).

```
// вага цього символу

this.ieaf = leaf; this.character = character; this.weight = weight;

Class Tree {
public Tree child0;       // нащадки «0» і «1»
public Tree childl;
public boolean leaf;      // ознака листка дерева
public int character;     // вхідний символ
public int weight;        // вага цього символу

public Tree() {}
public Tree(int character, int weight, boolean leaf)
{
  Обхід дерева з ґенерацією кодів
•         «Роздрукувати» листове дерево і записати код Хафмана в масив
•         Рекурсивно обійти ліве піддерево (з ґенеруванням коду).
•         Рекурсивно обійти праве піддерево.
*/
public void traverse(String code, Huffman h)
{
if (leaf)
{
System.out.println((char)character +» «+ weight +» «+ code); h.code[character] = code;
}
if (childO != null) child0.traverse(code + «0», h); if (childl != null) child 1.traverse(code + «1», h);
}
}
class Huffman
{
public static final intALPHABETSIZE = 256;
Tree[] tree = new Tree[ALPHABETSIZE]; // робочий масив дерев int weightsf] = new int[ALPHABETSIZE]; II ваги символів
public String[] code = new String[ALPHABETSIZE]; // коди Хафмана private intgetLowestTree(int used)
    { // шукаємо «найлегше» дерево int min=0;
    for (int i=1; i<used; i++)
```

```
        if (tree[i].weight < tree[min].weight) min = i;
        return min;
        }
        public void growTree( int[] data)
        { // нарощуємо дерево
            for (int i=0; i<data.length; I++) // шукаємо ваги символів weights[data[i]]++;
        // заповнюємо масив з «листкових» дерев
        int used = 0;                          //з використаними символами
        for (int c=0; з < ALPHABETSIZE; C++)
        {
            int w = weights[c];
            if (w != 0) tree[used++]= new Tree(з, w, true);
        }
        while (used > 1)
        {                                      // парами зливаємо легкі гілки
            int min = getLowestTree( used );   // шукаємо першу гілку
            int weightO = tree[min].weight;
            Tree temp = new Tree();            // створюємо нове дерево
            temp.childO = tree[min];           // і прищеплюємо першу гілку
            tree[min]= tree[—used];            // на місце першої гілки поміщаємо
            // останнє дерево в списку
            min = getLowestTree( used );       // шукаємо 2 гілку і
            temp.childl = treefmin];           // прищеплюємо їїдо нового дерева
            temp.weight = weightO + tree[min].weight; // рахуємо вагу нового
                                                      //дерева
            tree[min]= temp;                   // нове дерево поміщаємо на місце 2 гілки
        }                                      // отримали 1 дерево Хафмана
        }
    public void makeCodeQ { // запускаємо обчислення кодів Хафмана tree[0].traverse(«», this);
        }
        public String coder( int[] d a t a)
        {// кодує дані рядка з 1 і 0 String str = «»;
        for (int i=0; i<data.length; i++) str += code[data[i]]; return str;
             public String decoder(String data)
         {
                String str=»»;        // перевіряємо в циклі дані на входження
                int 1 = 0;            // коду, якщо так, то відкидаємо його ...
                while(data.length() > 0)
                {
                    for (int c=0; c < ALPHABETSIZE; c++)
                    {
                        if (weights[c]>0 && data.startsWith(code[c]))
                        {
                            data = data.substring(code[c].length(), data.IengthQ); str += (char)c;
                        }
                    }
                }
                return str;
         }
    }
        public class HuffmanTest { // тест і
      демонстрація
```

```
        public static void main(String[] args)
    {
        Huffman h = new HuffmanQ;
        String str = «to be or not to be?»; int data[] =
      new int[str.length()]; for(int i=0; i<str.length();
      i++) data[i]= str.charAt(i); h.growTree( data);
      h.makeCodeQ; str = h.coder(data);
        System.out.println(str);
        System.out.println(h.decoder( str));
    }
}
```

Змістовий модуль 5. Методи обчислень. Основні проблеми чисельного розв'язання задач. Системи лінійних алгебраїчних рівнянь.

Тема 1. Основні поняття чисельних методів. Класифікація похибок. Абсолютна і відносна похибки, середня квадратична похибка, поширення похибок. Підвищення точності обчислень.

Серед інформаційних технологій, які лежать в основі всіх напрямів підготовки спеціалістів з комп'ютерних технологій, особливе місце займає математичне моделювання. При цьому під *математичною моделлю* фізичної системи, об'єкта або процесу звичайно розуміють сукупність математичних співвідношень (формул, рівнянь, логічних виразів), які визначають характеристики стану і властивості системи, об'єкта і процесу та їх функціонування залежно від параметрів їх компонентів, початкових умов, вхідних збуджень і часу. Загалом математична модель описує функціональну залежність між вихідними залежними змінними, через які відображається функціонування системи, незалежними (такими, як час) і змінюваними змінними (такими, як параметри компонентів, геометричні розміри та ін.), а також вхідними збудженнями, прикладеними до системи.

Згадана функціональна залежність, що відображається математичною моделлю, може бути явною чи неявною, тобто може бути зображена або як просте алгебраїчне співвідношення, або ж як велика за розміром сумісна система диференціально-алгебраїчних рівнянь. До того, як обчислювальна техніка набула широкого розповсюдження, переважали явні функціональні моделі низьких порядків, пристосовані до можливостей розрахунків ручним способом або розрахунків з малим ступенем механізації (логарифмічна лінійка, арифмометр та ін.).

Саме вони і є сьогодні теоретичною основою багатьох інженерних та природничих дисциплін, яка дозволяє під час проектування проводити наближені розрахунки з точністю до кількох десятих відсотка з подальшим обов'язковим макетуванням проектованого об'єкта та його експериментальним доведенням до потрібних параметрів, внаслідок чого розробка нового виробу розтягується на багато років.

Сучасні комп'ютери дозволяють у багатьох випадках відмовитися від натурного макетування проектованих виробів, замінивши його математичним моделюванням (обчислювальним експериментом), що дуже важливо, коли натурне макетування складне або практично неможливе (наприклад, моделювання прориву дамби, переміщення всюдиходу поверхнею Марса та ін.). Але при цьому повинна бути істотно підвищена точність математичних моделей об'єктів та систем, що враховують багато фізичних ефектів та дестабілізуючих чинників, якими раніше нехтували. В результаті розмірність і складність математичних моделей істотно зростають, а їх розв'язання в аналітичному вигляді стає неможливим. Це звичайний для сучасної науки і техніки компроміс, що полягає в отриманні нової якості одного параметра (висока точність обчислювального експерименту і відмова від натурного макетування) за рахунок зменшення чи ускладнення іншого параметра (відмова від звичних для вищої математики аналітичних рішень).

Для кожної математичної моделі звичайно формулюється математична задача. У загальному випадку, коли функціональна залежності для множини вхідних даних (значення незалежних та змінюваних змінних і вхідних збуджень), що виступають як множина аргументів, задана неявно, за допомогою математичної моделі необхідно визначити множину вихідних залежних змінних, що виступають як множина значень функцій. При цьому відповідно до виду математичної моделі розрізняють такі базові типи математичних задач:

- ♦ розв'язання системи лінійних (в загальному випадку лінеаризованих) рівнянь;
- ♦ розв'язання нелінійних алгебраїчних рівнянь;
- ♦ апроксимація масиву даних або складної функції набором стандартних, більш простих функцій;
- ♦ чисельне інтегрування і диференціювання;
- ♦ розв'язання систем звичайних диференціальних рівнянь;
- ♦ розв'язання диференціальних рівнянь в частинних похідних;
- ♦ розв'язання інтегральних рівнянь.

Прості математичні задачі малої розмірності, що вивчаються в курсі вищої математики, допускають можливість отримання аналітичних рішень. Складні математичні моделі великої розмірності вимагають застосування чисельних методів, що вивчаються в даному курсі.

Чисельні методи — це математичний інструментарій, за допомогою якого математична задача формулюється у вигляді, зручному для розв'язання на комп'ютері. У такому разі говорять про перетворення математичної задачі в *обчислювальну* задачу. При цьому послідовність виконання необхідних арифметичних і логічних операцій визначається *алгоритмом* її розв'язання. Алгоритм повинен бути рекурсивним і складатися з відносно невеликих блоків, які багаторазово виконуються для різних вхідних даних.

Слід зазначити, що з появою швидких та потужних цифрових комп'ютерів роль чисельних методів для розв'язання наукових та інженерних задач значно зросла. І хоча аналітичні методи розв'язання математичних задач, як і раніше, дуже важливі, чисельні методи істотно розширюють можливості розв'язання наукових та інженерних задач, не дивлячись на те, що самі рівняння математичних моделей з ускладненням структури сучасних виробів стають погано обумовленими та жорсткими, що істотно ускладнює їх розв'язування. Узявши виконання рутинних обчислень на себе, комп'ютери звільняють час вченого або інженера для творчості: формулювання задач і генерування гіпотез, аналізу та інтерпретації результатів розрахунку тощо.

Чисельні методи забезпечують системний формалізований підхід до розв'язання математичних задач. Проте за умов їх ефективного використання окрім уміння присутня і деяка частка мистецтва, що залежить від здібностей користувача, оскільки для розв'язання кожної математичної задачі існує декілька можливих чисельних методів і їх програмних реалізацій для різних типів комп'ютерів. На жаль, для обрання ефективного способу розв'язання поставленої задачі лише інтуїції замало, потрібні глибокі знання і певні навички. Існує декілька переконливих причин, що мотивують необхідність глибокого вивчення чисельних методів майбутніми фахівцями у галузі комп'ютерно-системної інженерії та прикладної математики.

Чисельні методи є надзвичайно потужним інструментарієм для розв'язання проблемних задач, що описуються довільними нелінійними диференціально-алгебраїчними рівняннями великої розмірності, для яких в даний час не існує аналітичних рішень. Освоївши такі методи, майбутній фахівець набуває здібностей до системного аналізу через математичне моделювання найскладніших задач сучасної науки і техніки.

У своїй майбутній професійній діяльності такий фахівець у першу чергу орієнтуватиметься на використання пакетів сучасних обчислювальних програм, причому те, наскільки правильно він буде їх застосовувати, безпосередньо залежатиме від знання і розуміння ним особливостей і обмежень, властивих чисельним методам, що реалізовані в пакеті. Може трапитися, що одна й та сама математична задача за допомогою певного програмно-технічного комплексу буде одним фахівцем успішно розв'язана, а іншим — ні, оскільки в сучасних пакетах передбачено їх налагоджування для конкретної задачі.

Може з'ясуватися, що низку задач неможливо розв'язати з використанням наявних пакетів програм. Якщо майбутній фахівець знає чисельні методи і володіє навичками програмування, він буде в змозі самостійно провести розробку необхідного алгоритму і програмно його реалізувати, вбудувавши в обчислювальний комплекс.

Вивчення чисельних методів стимулює освоєння самих комп'ютерів, оскільки найкращим способом навчитися програмувати є написання комп'ютерних програм власноруч. Правильно застосувавши чисельні методи, майбутній фахівець зможе пересвідчитися у тому, що комп'ютери успішно розв'язують його професійні задачі. При цьому він сам відчує вплив похибок обчислень на результат і навчиться контролювати ці похибки.

Вивчення чисельних методів сприяє також переосмисленню і більш глибокому розумінню математики в цілому, оскільки однією із задач чисельних методів є зведення методів вищої математики до виконання простих арифметичних операцій.

Хоча існує безліч чисельних методів, усі вони (як і алгоритми, що їм відповідають) мають багато спільних властивостей і характеристик. Чисельні методи:

♦ передбачають проведення великої кількості рутинних арифметичних обчислень за допомогою рекурсивних співвідношень, що використовуються для організації *ітерацій,* тобто повторюваних циклів обчислень зі зміненими початковими умовами для поліпшення результату;

♦ направлені на локальне спрощення задачі, коли, наприклад, використовувані нелінійні залежності лінеаризуються за допомогою своїх обчислених похідних або похідні замінюються різницевими апроксимаціями;

♦ значно залежать від близкості початкового наближення (або декількох наближень), необхідного для початку обчислень до розв'язку, від властивостей нелінійних функцій, які використовуються в математичних моделях, що накладає обмеження (для забезпечення єдиного розв'язку) на їх диференційованість, на швидкість зміни функцій та ін.;

Чисельні методи характеризуються:

♦ різною *швидкістю збіжності,* тобто числом ітерацій, виконання яких необхідне для отримання заданої точності розв'язку;

♦ різною *стійкістю,* тобто збереженням достовірності розв'язку під час подальших ітерацій;

♦ різною *точністю* отримуваного розв'язку в разі виконання однакового числа ітерацій або циклів обчислень.

Чисельні методи розрізняються:

♦ за широтою і легкістю застосування, тобто за ступенем своєї *універсальності* та *інваріантності* для розв'язання різних математичних задач;

♦ за *складністю* їх програмування;

♦ за можливостями використання у разі їх реалізації наявних бібліотек функцій і процедур, створених для підтримки різних алгоритмічних мов;

♦ за *ступенем чутливості* до погано обумовлених (або некоректних) математичних задач, коли малим змінам вхідних даних можуть відповідати великі зміни розв'язку.

Основні проблеми чисельного розв'язання задач

При застосуванні чисельних методів розв'язки задач виявляються, як правило, наближеними. Пояснюється це в багатьох випадках тим, що точні методи їх розв'язання дотепер невідомі. Крім того, навіть при застосуванні точного методу задовольняються наближеним розв'язком, зокрема, з таких причин:

— точний розв'язок виявляється трудомістким; тоді як наближений при істотно меншому об'ємі обчислень виявляється цілком прийнятним за своїм характером;

— точність отриманого результату не відіграє істотної ролі, тому що в будь-якому разі заокруглюється до цілого числа (наприклад, при визначенні кількості механізмів, необхідних для виконання даного обсягу робіт).

Наближений розв'язок задачі повинен «не набагато відрізнятися» від точного розв'язку, інакше ним не можна скористатися з конкретною метою. Що означає термін «не набагато відрізняється» або, інакше кажучи, що варто розуміти під неточністю (наближеністю)

розв'язку? Кожен чисельний метод дозволяє оцінювати ступінь неточності розв'язку, одержуваного цим методом. У курсі чисельних методів ступінь неточності розв'язку характеризується поняттям похибки розв'язку. Потрібно зазначити, що теорія похибок є одним із основних розділів обчислювальної математики. Очевидно, що відхилення наближеного результату від точного напряму залежить від коректності поставленої задачі та від наявних вхідних даних. Тому актуальним є дослідження збіжності наближеного розв'язку, що пропонує чисельний алгоритм, до точного розв'язку поставленої задачі .

Таким чином, основними проблемами чисельного розв'язання задач можна вважати:

-проблему оцінки похибки наближеного розв'язку; -проблему коректності та обумовленості поставленої задачі; -проблему збіжності наближеного методу до точного.

1.1 Класифікація похибок

При розв'язанні прикладних задач дуже важливо мати уявлення про точність отриманих результатів. Похибки, що можуть бути закладені в таких результатах, утворюються з багатьох причин.

Можна визначити чотири основні джерела похибок результату чисельного методу:

5) вхідні дані;
6) математична модель;
7) наближений метод;
8) округлення при розрахунках. Проаналізуємо їх.

Похибки вхідних даних

Точні значення багатьох величин практично ніколи не можуть бути введені в процес обчислень, наприклад, ірраціональних величин $\boldsymbol{\pi}$, $\boldsymbol{e}$, $\sqrt{\mathbf{2}}$ та ін. У цих випадках неминучі похибки округлення. При розв'язанні багатьох задач за вхідні беруться значення величин, отриманих з експерименту. З багатьох причин, у тому числі обмеженої точності вимірювальної апаратури і впливу різних випадкових чинників, експериментальні дані завжди мають похибки того або іншого порядку. Так, точність вимірювання температури, відстані, об'єму, ваги залежить від досконалості застосовуваних вимірювальних приладів. Похибки можуть бути у вхідних даних, отриманих теоретично. Природно, що вони впливають на результати розв'язку задачі, однак жодним чином їх усунути не можна. Тому похибки такого типу часто називають *неусувними.*

Похибки математичної моделі

Необхідно зазначити, що в більшості випадків фахівцю вдається підібрати для розв'язання задачі наближений метод, що дозволяє одержати цілком задовільні за ступенем точності результати. Однак розв'язувана задача є не тим реальним завданням, з яким фахівцю доводиться мати справу, а його спрощеною математичною моделлю. Так, при розрахунку авіаційного двигуна або несучої конструкції промислової споруди неможливо ввести до розгляду їх реальну надзвичайно складну форму, врахувати наявність усіх отворів, деталей сполучення і т.п. При визначенні оптимального складу персоналу універмагу, кас попереднього продажу залізничних квитків доводиться припускати, що покупці приходять через рівні проміжки часу, час обслуговування кожного з них однаковий і таке інше.

Розв'язок реальної задачі не збігається із результатом, отриманим при розгляді її математичної моделі навіть із застосуванням точних методів розв'язку, а похибки, що виникають при цьому, можна назвати *похибками математичного моделювання.*

Похибки наближеного методу

У випадку, коли розв'язати задачу точно неможливо, доводиться застосовувати різні наближені методи. Результати такого підходу завчасно містять похибки, характер яких залежить від використовуваного наближеного методу *(похибки методу).*

При застосуванні наближених методів розв'язання задач, наприклад ітераційних, точні значення шуканих величин можуть бути отримані тільки після виконання нескінченного числа етапів обчислень, що практично здійснити неможливо. Доводиться задовольнятися певним

числом етапів і відповідними наближеними результатами із так званими *залишковими похибками.*

Похибки заокруглень при розрахунках При реалізації на ЕОМ алгоритмів, що містять велику кількість операцій множення і ділення, типовими є *похибки округлення.* При виконанні операцій множення кількість розрядів може зрости настільки, що всі вони вже не можуть бути розміщені в елементах запам'ятовуючих пристроїв ЕОМ.

Частину розрядів праворуч доводиться відкидати, округляти числа. Сам по собі процес округлення числа не обов'язково призводить до внесення в нього якої-небудь істотної похибки. Так, при обчисленні зі звичайною точністю в сучасних ЕОМ можна утримувати, наприклад, дев'ять десяткових розрядів. Природно, що простим відкиданням в ЕОМ десятого і наступних розрядів ми вносимо в число лише дуже незначні зміни. Порівняємо дванадцятирозрядне число 1000000,00297 і округлене дев'ятирозрядне число 1000000,00. Внесена в результаті округлення похибка становить величину 0,00297. Однак у процесі виконання великої кількості арифметичних операцій похибки, послідовно накопичуючись, породжують нові. Таке нагромадження похибок округлення може призвести до дуже істотних помилок в остаточних результатах.

Похибки округлення особливо доводиться враховувати при реалізації нестійких обчислювальних процесів, у яких незначні похибки у вихідних даних або результатах проміжних обчислень можуть призвести до істотних помилок у остаточному результаті.

Приклад. Нехай необхідно обчислити величину *c* за формулою

$$c = a - b, \quad (1.1)$$

де $a = 139{,}27$; $b = 138{,}97$. Одержимо $c = 0{,}3$.

Припустимо, що величини *a* і *b* обчислені з похибками, що не перевищують 1% їх точних значень, a=140,62, b=37,62. Обчислюючи величину *c* за формулою (1.1) із наближеними значеннями, одержимо c=140,62-137,62=3,0. Отже, похибки в обчисленні вихідних величин *a* і *b* призвели до десятикратного збільшення числа *c* .

1.2 Абсолютна і відносна похибки

Абсолютна похибка - це модуль різниці між відповідним точним значенням розглянутої величини *A* і наближеним її значенням a. Вона має вигляд

$$\Delta = |A - a|. \quad (1.2)$$

Безпосередньо за значенням абсолютної похибки досить важко робити висновок про ступінь розбіжності між точним значенням *A* величини і його наближеним значенням. Так, похибка 2м цілком припустима при визначенні відстані між Києвом і Сумами та абсолютно неприпустима при вимірюванні розмірів кімнати. Тому застосовується ще одна характеристика наближених величин — їх відносна похибка.

Відносною похибкою δ наближеного значення величини, точне значення якої дорівнює *A,* називається відношення його абсолютної похибки A до модуля точного значення, тобто

$$\delta = \frac{\Delta}{|A|} \quad (1.3)$$

Наприклад, нехай в результаті вимірювання довжини бігової доріжки отримано значення a=99,1м. Точне значення цієї величини $A = 100$м. Абсолютна похибка $\Delta = |100 - 99{,}1| = 0{,}9$. Відносна похибка за формулою (1.3)

становить $\delta = \frac{0{,}9}{|100|} = 0{,}009$.

Із формул (1.2)—(1.3) бачимо, що абсолютна похибка має розмірність оцінюваних цією похибкою величин, відносна похибка завжди безрозмірна.

Величини Δ і δ можуть бути обчислені точно лише в тих випадках, коли відоме не тільки наближене числове значення розглянутої величини, але і її точне значення. Останнє, однак, можливе далеко не у всіх випадках. Крім того, часто доводиться аналізувати похибки деякої множини наближених величин, наприклад, похибки вимірювання розмірів серії виготовлених деталей, викликані недосконалістю застосовуваних вимірювальних інструментів. Якість серії вимірювань для всіх деталей може оцінюватися найбільшою за модулем величиною абсолютної

або відносної похибки їх розмірів. Тому часто вводяться поняття граничних абсолютної та відносної похибок.

За *граничну абсолютну похибку* Δ * наближеного числа може бути взяте будь-яке число, не менше абсолютної похибки цього числа,

$$\Delta^* \geq \Delta . \qquad (1.4)$$

Аналогічно за *граничну відносну похибку 8* * наближеного числа може бути взяте будь-яке число, що задовольняє умову

$$\delta^* \geq \delta . \qquad (1.5)$$

При аналізі серії вимірювань за Δ * і δ * беремо найбільші з отриманих відповідних значень Δ і δ і тим самим визначаємо межі, всередині яких знаходяться відповідні похибки.

Значущими цифрами числа *а* називають усі цифри в його записі, починаючи з першої ненульової зліва. Значущу цифру числа *а* називають *правильною,* якщо абсолютна похибка числа не перевищує одиниці відповідного цій цифрі розряду.

Приклад 1. Для ряду $\sum_{n=0}^{n} \frac{72}{n^2+5n+4}$ знайти суму S аналітично. Обчислити значення часткових сум ряду $S_N = \sum_{n=0}^{n} a_n$

і знайти величину похибки при значеннях N=10,10^2, 10^3,10^4,10^5. Побудувати гістограму залежності правильних цифр результату від N.

Знайдемо точну суму цього ряду:

$$S_N = \sum_{n=0}^{N} \frac{72}{n^2+5n+4} = \sum_{n=0}^{N} \frac{72}{(n+1)(n+4)} =$$

$$= 72 \cdot \sum_{n=0}^{N} \frac{1}{3} \cdot \left(\frac{1}{n+1} - \frac{1}{n+4} \right) = = 24 \cdot \left(1 + \frac{1}{2} + \frac{1}{3} - \frac{1}{N+2} - \frac{1}{N+3} - \frac{1}{N+4} \right),$$

Отже, $S = \lim_{N \to \infty} S_N = 44$. Уведемо функцію часткових сум

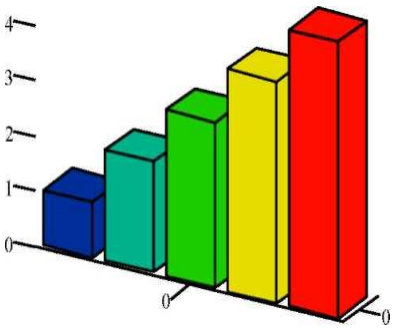
$S(N) = \sum_{n=0}^{N} \frac{72}{n^2+5n+4}$. Тоді абсолютну похибку можна визначити

за допомогою функції $d(N) = |S(N) - S|$.

Результати обчислювального експерименту

N	Значення частк. суми ряду S(N)	Абсолютна похибка *d(N)*	Кільк. правил. цифр M_i
10	S(10)=38.439560439	d(10) =5.56	M_1 = 1
$_2$ 10	S(100)=43.3009269	d (100)=0.699	$M2$ = ^{2}
$_3$ 10	S(1000)=43.9282153	d (1000)=0.072	Mз = ^{3}
$_4$ 10	S(10000)=43.992802	d (10000)=0.0072	M_4 = 4
$_5$ 10	S(100000)=43.999280 2 1599	d (100000)=0.000 72	M_5 = 5

Висновок. Як бачимо з наведеного обчислювального експерименту, збільшення числа членів ряду в 10 разів порівняно з попереднім випадком збільшує число правильних цифр у відповіді на гістограмі



Приклад 2. Для матриці $A=\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$

Розв’язати питання про існування оберненої матриці в таких випадках:

3) елементи матриці задані точно;

4) елементи матриці задані наближено з відносною похибкою

a) $\delta = \alpha\%$ та b) $\delta = \beta\%$.

Знайти відносну похибку результату.

Це питання вирішується шляхом знаходження визначника й порівняння його з нулем. У випадку, коли елементи визначника задані точно, варто обчислити визначник і правильно відповісти на поставлене в задачі питання.

У випадку, коли елементи визначника задані наближено з відносною похибкою 5, питання є складнішим. Нехай елементи матриці позначені через a_{ij} . Тоді кожен елемент матриці a_{ij} тепер уже не дорівнює конкретному значенню, а може набувати будь-якого значення з відрізка [a_{ij}(1-δ); a_{ij} (1+δ)], якщо a_{ij}> 0 , і з відрізка [a_{ij}(1+δ); a^ (1-δ)], якщо a_{ij}< 0. Множина всіх можливих значень елементів матриці являє собою замкнену обмежену множину в 9-вимірному просторі. Сам визначник є неперервною й диференційованою функцією 9 змінних - елементів матриці a_{ij}. За відомою теоремою Вейєрштрасса ця функція досягає на зазначеній множині свого найбільшого та найменшого значень M і т. Якщо відрізок [m, M] не містить точку 0 , то це означає, що при будь-яких припустимих значеннях елементів матриці a визначник не набуває значення 0. Якщо ж точка 0 належить відрізку [т, М], таке твердження буде неправомірним. Буде мати місце невизначеність.

З’ясувати m і M допомагають наступні міркування. Як функція своїх аргументів (елементів матриці a_{ij}.) визначник має таку властивість (принцип максимуму): ця функція досягає свого найбільшого і найменшого значень завжди на границі області . Більше того, можна довести, що ці значення досягаються в точках, координати яких мають вигляд (1±5).

Таких точок 2^9 =512. У кожній з них варто обчислити визначник, а потім вибрати з отриманих значень найбільше та найменше. Це й будуть числа M і m.

1.4 Середні квадратичні похибки

Нехай передбачається проведення серії вимірів деякої величини X. У кожному з вимірів буде отримане якесь її значення, причому залежно від точності приладу, зокрема, ці значення будуть знаходитися в деякому інтервалі, загальне їх число скінченне. Позначимо ці значення $x_1, x_2, \ldots, x_n$,

їх ймовірності $p_1,p_2,...p_n$. Оскільки заздалегідь невідомо, яке значення величини X буде отримано в кожному вимірі, ця величина є випадковою.

Математичне очікування X виражається формулою

$$M[X] = \sum_{i=1}^{n} x_i\, p_i \qquad (1.6)$$

Про якість вимірів, тобто ступінь розкиду помилок виміру, можна роб висновкиити за розмірами дисперсії, або середнього квадратичного відхилення випадкової величини:

$$D[X]= \delta_x^2=\sum_{i=1}^{n} p_i(x_i - M[X])^2. \qquad (1.7)$$

Величина c_x називається в теорії похибок *середньою квадратичною похибкою* вимірювання.

Якщо результати вимірювання є незалежними, тобто результат довільного виміру не залежить від того, які результати отримані в інших вимірах, для них прийнятні теореми Чебишева і Бернуллі. Зокрема, бувають наступні припущення.

3 Якщо випадкова величина X набуває тільки невід'ємних значень, частина яких менша деякого додатного числа a, то

$$p[(X<a)]\geq 1-\frac{M[X]}{a}. \qquad (1.8)$$

4 Якщо a > 0, то

$$p[|(X-M[X]| <a)] \geq 1-\frac{\delta_x^2}{a} \qquad (1.9)$$

Відзначимо, що формулою (1.7) користуються для обчислення середніх квадратичних похибок і в детермінованих процесах.

де A —точне значення числа X, а A_i — абсолютні похибки.

1.5 Поширення похибок

Важливим у чисельному аналізі є питання про те, як помилка, що виникла у визначеному місці в ході обчислень, поширюється далі, тобто чи стає її вплив більшим або меншим залежно від того, як виконуються наступні операції. Сформулюємо деякі правила оцінки похибок при виконанні операцій над наближеними числами:

- при додаванні або відніманні чисел їхні абсолютні похибки додаються;
- при множенні або діленні чисел їхні відносні похибки додаються.

Ці правила можна вивести безпосередньо. Нехай є два наближення a_1 і a_2 до чисел x_2 і x_2, а також відповідні абсолютні похибки Δa_1, Δa_2.

Оцінимо, наприклад, похибку суми

$\Delta(a1 + a2) = |(x1 + x2) - (a1 + a2)\ =$

$= |(x1 - a1) + (x2 - a2)| < |x1 - al| + |x2 - a2| < \Delta al + \Delta a2.$

Для визначення оцінок похибки арифметичних дій можна використовувати загальне правило оцінки похибки функції.

Розглянемо функцію $y=f(x)$. Нехай a - наближене значення аргумента x, Δa - його абсолютна похибка. Абсолютну похибку функції можна вважати її приростом, який можна замінити диференціалом $\Delta y \approx dy$.

Тоді одержимо

$\Delta y = |f'(a)|\Delta a,\ \delta y = |f'(a) / f(a)|\Delta a.$

Застосуємо загальне правило, наприклад, для оцінки похибки суми $f(x_1,x_2)= x_1 + x_2$

$\Delta(a_1+a_2) = |f'_{x1}| * \Delta a_1 + |f'_{x2}| * \Delta a_2 = \Delta a_1 + \Delta a_2$

та добутку $f(x_1,x_2)= x_1x_2$

$\Delta(a_1a_2) = |f'_{x1}(a_1,a_2)|\Delta a_1 + |f'(a_1a_2)|\Delta a_2 = |a_2|\Delta a_1 + |a_1|\Delta a_2.$

Тут через $a\ i$ і a_2 позначені значення величин xi і x_2, задані з абсолютними похибками Δa_1 і Δa_2.

Розглянемо віднімання двох майже рівних чисел. Запишемо вираз для відносної похибки

різниці у вигляді

$\delta (a_1 — a_2) = \Delta (a_1 — a_2) / | a_1 — a_2 | = (\Delta a_1 + \Delta a_2) / | a_1 - a_2 |$

При $a_1 \approx a_2$ ця похибка може бути як завгодно великою. Нехай a1=2520, a2=2518. Абсолютні похибки вихідних даних Δa1=Δa2=0.5; відносні похибки - δa1 ≈ δ a2≈0.002 (0.2%). Відносна похибку різниці буде дорівнювати δ(a1–a2)=(0.5+0.5)/2=0.5 (50%). Оскільки в подальших обчисленнях ця велика відносна похибка буде поширюватися, може виявитися сумнівною точність остаточного результату обчислень.

1.6 Підвищення точності результатів обчислень (рекомендації)

Щоб зменшити можливу похибку результату при розв'язуванні задачі, рекомендується дотримуватися таких правил для практичної організації обчислень.

III Похибка суми кількох чисел при розрахунку на ЕОМ зменшиться, якщо починати додавання з менших за величиною доданків.

Якщо додається досить багато чисел, то їх краще розбити на групи з чисел близьких за величиною, провести додавання в групах за вищезгаданою рекомендацією, після чого отримані суми додати, починаючи з меншої.

Якщо задано *n* додатних чисел приблизно однакової величини, то загальна помилка округлення зменшиться, якщо числа додати спочатку групами по п-чисел, а потім додати *n* - часткових сум. При великих *n* верхня межа округлення при такому способі становить всього *1/n* від відповідної межі при довільному додаванні чисел одне до одного.

Причина того, що не виконується комутативний закон додавання, полягає в округленні проміжних результатів, коли багатозначні числа не вміщуються в розрядну сітку ЕОМ. Тому і не все одно, в якому порядку необхідно виконувати арифметичні операції, щоб результат був якомога точнішим.

IV Варто уникати віднімання двох майже однакових чисел. Обчислюючи різницю двох чисел, доцільно винести за дужки їхній спільний множник. Для прикладу обчислимо величину

$$P = 6.250001*16 - 25.000003*4 = 1*10^{-5}.$$

Винесемо число "4" за дужки, одержимо точний результат:

$P = 4(6.250001 *4 - 25.000003) = 4*10^{-6}$.

Зменшити похибку різниці дозволяють перетворення:

$(a + \varepsilon)^2 - a^2 = \varepsilon (2a + \varepsilon)$;

$\sqrt{a + \varepsilon} - \sqrt{a} = \varepsilon(\sqrt{a + \varepsilon} + \sqrt{a})$;

$a - \sqrt{a^2 + \varepsilon} = -\varepsilon /(a + \sqrt{a^2 + \varepsilon})$;

$1 - a/(a+\varepsilon) = \varepsilon/(a+\varepsilon)$.

Тут ε- мале в порівнянні з a число.

Тема 2. Метод Гауса, метод Краута, метод прогонки.

Розглянемо систему вигляду

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1, \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2, \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots, \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n = b_n. \end{cases}$$

Її матричний вигляд

$$AX = B. \qquad (3.2)$$

Тут A – $\{[a_{ij}],(i,j = \overrightarrow{1,n})\}$ – матриця системи,

$$B = \begin{bmatrix} b_1 \\ b_2 \\ \\ b_n \end{bmatrix}, \quad X = \begin{bmatrix} x_1 \\ x_2 \\ \\ x_n \end{bmatrix}$$ – вектори-стовпці.

Відомо, що система (3.1) має єдиний розв'язок, якщо її матриця невироджена (тобто визначник матриці А відмінний від нуля). У випадку виродженості матриці система може мати безліч розв'язків (якщо ранг матриці А і ранг розширеної матриці, отриманої додаванням до А стовпця вільних членів, однакові) або ж не мати розв'язків узагалі (якщо ранги матриці А і розширеної матриці не збігаються).

Методи чисельного розв'язання СЛАР поділяються на точні і наближені. Метод вважають точним, якщо, нехтуючи похибками округлення, він дає точний результат після виконання певної кількості обчислювальних операцій. Математичні пакети прикладних програм для ПЕОМ містять стандартні процедури розв'язання СЛАР такими поширеними точними методами, як метод Гауса, метод Жордана-Гауса, квадратного кореня та інші.

До наближених методів розв'язання СЛАР належать метод простої ітерації, метод Зейделя, метод релаксації та інші. Вони дозволяють отримати послідовність $\{X^к\}$ наближень до розв'язку X^* таку, що $\lim_{k\to\infty} X^k = X^*$.

Ітераційні методи прості, легко програмуються і мають малу похибку округлення, яка не накопичується, але вони дають збіжну послідовність наближень тільки за виконання певної умови, що гарантує виконання принципу стискаючих відображень

(дивись пункти 2.2 -2.5).

Розглянемо більш детально ці дві різні групи підходів до розв'язання СЛАР.

3.2 Метод Гауса

Цей метод базується на приведенні шляхом еквівалентних перетворень вихідної системи (3.1) до вигляду з верхньою трикутною матрицею.

$$\left.\begin{array}{l} c_{11}x_1 + c_{12}x_2 + \ldots + c_{1n}x_n = d_1 \\ 0 + c_{22}x_2 + \ldots + c_{2n}x_n = d_2 \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ 0 + \ldots + 0 + c_{n-1,n-1}x_{n-1} + c_{n-1,n}x_n = d_{n-1} \\ 0 + 0 + \ldots + 0 + c_{nn}x_n = d_n \end{array}\right\}$$

Тоді з останнього рівняння відразу визначаємо $x_n = \frac{d_n}{c_{nn}}$.

Підставляючи його в попереднє рівняння, знаходимо x_{n-1} і т.д. Загальні формули для отримання розв'язку мають вигляд

$$x_k = \frac{1}{c_{kk}}(d_k - \sum_{j=k+1}^{n} c_{ki} x_j), k = n, n-1, ..., 1.$$

При обчисленнях за формулами (3.4) треба буде виконати приблизно $1/2n^2$ арифметичних дій. Зведення системи (3.1) до вигляду (3.3) можна виконати, послідовно заміняючи рядки матриці системи їх лінійними комбінаціями. Перше рівняння не змінюється. Віднімемо з другого рівняння системи (3.1) перше, помножене на таке число, щоб звернувся в нуль коефіцієнт при x_1. Потім у такий самий спосіб віднімемо перше рівняння з третього, четвертого і т.д. Таким чином обнуляються всі коефіцієнти першого стовпця, що лежать нижче головної діагоналі. Потім за допомогою другого рівняння виключимо з третього, четвертого і т.д. рівнянь коефіцієнти другого стовпця. Послідовно продовжуючи цей процес, виключимо з матриці всі коефіцієнти, що лежать нижче головної діагоналі.

Запишемо загальні формули процесу. Нехай проведене виключення коефіцієнтів з *к-1* стовпця. Тоді залишилися такі рівняння з ненульовими елементами нижче головної діагоналі:

$$\sum_{j=k}^{n} a_{ij}^{(k)} x_j = b_i^{(k)}, k \le i \le n.$$

Помножимо *к-й* рядок на число

$$c_{mk} = \frac{a_{mk}^{(k)}}{a_{kk}^{(k)}}, m > k$$

і віднімемо від *т-го* рядка. Перший ненульовий елемент цього рядка звернеться в нуль, а інші зміняться за формулами

$$a_{ml}^{(k+1)} = a_{ml}^{(k)} - c_{mk} a_{kl}^{(k)},$$

$$b_m^{(k+1)} = b_m^{(k)} - c_{mk} b_k^{(k)}, k < m, l \le n.$$

Виконуючи обчислення при всіх зазначених індексах, виключимо елементи к-го стовпця. Будемо називати таке виключення циклом процесу. Виконання всіх циклів називається прямим ходом виключення.

Після виконання всіх циклів утвориться система, матриця якої має трикутний вигляд . Її легко розв'язати зворотним ходом за формулами (3.4).

Виключення за формулами (3.7) не можна проводити, якщо в ході розрахунків на головній діагоналі виявиться нульовий елемент $a_{kk}^{(k)} = 0$ Але в першому стовпці проміжної

системи (3.5) всі елементи не можуть бути нулями: це означало б, що *detA=0.* Перестановкою рядків можна перемістити ненульовий елемент на головну діагональ і

продовжити розрахунки.

Для зменшення обчислювальної похибки можна кожне повторення зовнішнього циклу починати з вибору максимального за модулем елемента в *к-му* стовпці (головного елемента) і перестановки рівняння з головним елементом так, щоб він виявився на головній діагоналі. Цей варіант називається методом Гауса з вибором головного елемента.

Однією з характеристик ефективності того чи іншого алгоритму вважають обчислювальні витрати, що визначаються кількістю елементарних операцій, які необхідно виконати для одержання розв'язку. Для прямого ходу методу Гауса число арифметичних операцій, відповідно до (3.6), (3.7), становить

$$Q_1(n) = \sum_{k=1}^{n-1} \sum_{m=k+1}^{n} \left[\text{ділення} + \sum_{p=k}^{n+1} (\text{множення} + \text{віднімання}) \right] =$$

$$= \frac{1}{3} n(n-1)(2n + \frac{13}{2}) = \frac{2}{3} n^3 + \frac{3}{2} n^2 - \frac{13}{6} n.$$

Для зворотного ходу за формулами число арифметичних операцій дорівнює

$$Q_2(n) = \sum_{k=1}^{n} (\text{ділення} + \text{віднімання} + \sum_{j=k+1}^{n} \text{множення}) =$$

$$= \frac{1}{2} n(n+3) = \frac{1}{2} n^2 + \frac{3}{2} n.$$

Загальні обчислювальні витрати методу Гауса становлять

$$Q(n) = \frac{2}{3} n^3 + 2n^2 - \frac{2}{3} n \text{, тобто } Q(n) \approx \frac{2}{3} n^3 = O(n^3).$$

Метод Краута

Суть методу Краута, або LU-розкладання, полягає в тому, що це своєрідний перезапис методу Гауса. Він дозволяє зробити зручною комп'ютерну реалізацію методу Гауса. Можна явно виділити два етапи, у яких один робить перетворення з матрицею *A* системи, інший - з вектором правих частин *b*. Отже, нехай дана СЛАР *Ax=b,* наприклад, система розміром 4x4. Запишемо розширену матрицю системи

$$[Ab] = \left[\begin{array}{cccc|c} a_{11} & a_{12} & a_{13} & a_{14} & b_1 \\ a_{22} & a_{23} & a_{24} & a_{25} & b_2 \\ a_{31} & a_{32} & a_{33} & a_{34} & b_3 \\ a_{41} & a_{42} & a_{43} & a_{44} & b_4 \end{array}\right]$$

Тоді, за Гаусом можна явно виділити два етапи (тобто два кроки) - прямий хід (ПХ) і зворотний (ЗХ):

1) ПХ: $a_{k1}^{(i)} = a_{k1}^{(i-1)} - \frac{a_{i1}^{(i-1)}}{a_{i1}^{(i-1)}} - a_{ki}^{(i-1)}$

2) ЗХ: $x_i = b_i^{(i)} - \sum_{j}^{i-1} a_{ij} x_j$

На прямому ході ми робимо так звані "виключення", тобто приводимо матрицю до трикутного вигляду. Тепер легко знайти x_4, а потім і x_3 і т.д. Це був зворотний хід методу Гауса. Всі ці перетворення виконувалися не із самою матрицею, а з розширеною матрицею.

Головна ідея і потреба методу *LU* - декомпозиції полягає в тому, щоб розділити окремо етап перетворення коефіцієнтів матриці і окремо етап перетворення вектора правих частин.

Розглянемо k -ий крок методу Гауса, на якому здійснюється занулення піддіагональних елементів k –го стовпчика матриці $A^{(k-1)}$. Як було зазначено раніше, з цією метою використовується операція

$$a_{ml}^{(k)} = a_{ml}^{(k-1)} - c_{mk} a_{kl}^{(k-1)},$$

$$c_{mk} = \frac{a_{mk}^{(k-1)}}{a_{kk}^{(k-1)}}, m > k \quad m = \overline{k+1, n}, \; l = \overline{k, n}.$$

У термінах матричних операцій така операція еквівалентна множенню $A^{(k)} = M_k A^{(k-1)}$, де елементи матриці M_k, визначаються таким чином:

$$m_{ij}^{k} = \begin{cases} 1, \; i = j; \\ 0, \; i \neq j, \; j \neq k; \\ -c_{k+1,k}, \; i \neq j, \; j = k, \end{cases}$$ тобто матриця M_k має вигляд

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & \ldots -c_{k+1,k} & 1 & 0 & 0 \\ .. & .. & .. & .. & .. & .. \\ 0 & 0 & \ldots -c_{n,k} & 0 & 0 & 1 \end{pmatrix}.$$

При цьому вираз для зворотньої операції запишеться у вигляді $A^{(k-1)} = M_k^{-1} A^{(k)}$, *де*

У результаті прямого ходу методу Гауса отримаємо $A^{(n-1)} = U$,

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & ..c_{k+1,k} & 1 & 0 & 0 \\ .. & .. & .. & .. & .. & .. \\ 0 & 0 & ..c_{n,k} & 0 & 0 & 1 \end{pmatrix} = M_k^{-1}.$$

$A^{(0)} = M_1^{-1} A^{(1)} = M_k^{-1} M_2^{-1} A^{(2)} = M_k^{-1} M_2^{-1} \ldots M_{n-1}^{-1} A^{(n-1)}$,

де $A^{(n-1)} = U$ – верхня трикутна матриця, а $L = M_k^{-1} M_2^{-1} \ldots M_{n-1}^{-1}$ нижня трикутна матриця, що має вигляд

$$L = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ c_{21} & 1 & 0 & 0 & 0 & 0 \\ c_{31} & c_{32} & 1 & 0 & 0 & 0 \\ .. & .. & ..c_{k+1,k} & 1 & 0 & 0 \\ .. & .. & .. & .. & .. & .. \\ c_{n1} & c_{n2} & ..c_{nk} & c_{n,k+1} & ..c_{n,n-1} & 1 \end{pmatrix}.$$

У подальшому LU- розкладання може бути ефективно використано для розв'язання систем лінійних алгебраїчних рівнянь. Це дозволяє один раз перетворити матрицю системи, а потім неодноразово розв'язувати декілька систем з різними правими частинами. Обчислювальні витрати при цьому будуть зводитися тільки до зворотного ходу.

Запишемо $A\,x = b$, як

$L \cdot U \cdot x = b$.

Позначимо

$U \cdot x =$ y.

I , отже ,

$$L \cdot y = b.$$

Таким чином, прямий хід методу *LU*-декомпозиції складається з розкладу матриці *A* на нижню *L* та верхню *U* трикутні матриці - це прямий хід.

Потім визначається вектор y на основі співвідношень:

$y_1 = \frac{b_1}{l_{11}}$, $y_1 = \frac{b}{l_{11}}\left(b_i - \sum_{j=1}^{i-1} l_{ij}\ y_j\right)$.

На зворотному ході методу *LU* – декомпозиції розв'язується рівняння $U \cdot x = y$.

З урахуванням того, що U – трикутна матриця,

$$x_N = \frac{y_N}{u_{NN}} \quad , x_i = \frac{1}{u_{ii}}\left(y_i - \sum_{j=i+1}^{N} u_{ij} x_j\right).$$

Отже *LU*-розкладання є просто свого роду іншою формою запису еквівалентних перетворень матриці за методом Гауса, але проведених з урахуванням умови $A = L \cdot U$.

Приклад. Розв'яжемо СЛАР за схемою *LU*-розкладання:

$$\begin{cases} x_1 - 2x_2 + 3x_3 = 1 \\ 2x_1 + 3x_2 - x_3 = 2 \\ -x_1 - x_2 + x_3 = 3 \end{cases}$$

Виконаємо дії за алгоритмом і отримаємо матриці *L та U у вигляді:*

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -1 & -3/7 & 1 \end{bmatrix}, \ U = \begin{bmatrix} 1 & -2 & 3 \\ 0 & 7 & -7 \\ 0 & 0 & 1 \end{bmatrix}.$$

Спочатку знаходимо розв'язок системи $Lg = b$

$$\begin{cases} g_1 = 1 \\ 2g_1 + g_2 = 2 \\ -g_1 - 3/7 g_2 + g_3 = 3 \end{cases}$$

Отримаємо: $g = \{1,0,4\}$

Тепер реалізуємо зворотний хід методу Гауса, розв'язуючи систему $Ux = g$:

$$\begin{cases} x_1 - 2x_2 + 3x_3 = 1 \\ 7x_2 - 7x_3 = 0 \\ x_3 = 4 \end{cases}$$

Отже, остаточна відповідь: $x_1 = 4$, $x_2 = 4$, $x_3 = -3$.

Ітераційні методи розв'язування СЛАР. Методи простих ітерацій.

При великій кількості рівнянь прямі методи розв'язання СЛАР (за винятком методу прогонки) стають важко реалізованими на ЕОМ насамперед через складність зберігання й обробки матриць великої розмірності. У той же час характерною рисою багатьох СЛАР, що виникають у прикладних задачах є розрідженість матриць. Число ненульових елементів таких матриць є малим у порівнянні з їхньою розмірністю. Для розв'язання СЛАР з розрідженими матрицями краще використати ітераційні методи.

Методи послідовних наближень, у яких при обчисленні наступного наближення розв'язку використовуються попередні, уже відомі наближення розв'язку, називаються ітераційними (дивись 2.4).

Розглянемо СЛАР (3.1) з невиродженою матрицею (det A≠ 0). Розв'яжемо систему (3.1) щодо невідомих при ненульових діагональних елементах *aii≠ 0, i= 1…n* (якщо який-небудь

коефіцієнт на головній діагоналі дорівнює нулю, досить відповідне рівняння поміняти місцями з будь-яким іншим рівнянням). Одержимо систему у вигляді

$$\begin{cases} x_1 = \beta_1 + \alpha_{11}x_1 + \ldots + \alpha_{1n}x_n \\ x_2 = \beta_2 + \alpha_{21}x_1 + \ldots + \alpha_{2n}x_n \\ \ldots \\ x_n = \beta_n + \alpha_{n1}x_1 + \ldots + \alpha_{nn}x_n \end{cases}$$

або у векторно-матричній формі $X=\beta+\alpha X$.

$$X = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}, \beta = \begin{pmatrix} \beta_1 \\ \vdots \\ \beta_n \end{pmatrix}, \alpha = \begin{pmatrix} \alpha_{11} & \cdots & \alpha_{n1} \\ \vdots & \cdots & \vdots \\ \alpha_{n1} & \cdots & \alpha_{nn} \end{pmatrix}.$$

Вирази для компонентів вектора β та матриці α еквівалентної системи:

$$\beta_i = \frac{b_i}{a_{ii}}; \alpha_{ij} = -\frac{a_{ij}}{a_{ii}}, i, j = 1\ldots n, i \neq j; \alpha_{ij} = 0, i = j, i = 1\ldots n.$$

При такому способі приведення вихідної СЛАР до еквівалентного вигляду метод простих ітерацій ще називають методом Якобі. За нульове наближення $X(0)$ вектора невідомих візьмемо вектор правих частин $X(0)=\beta$ або $(x1\ (0), x2\ (0), \ldots, xn\ (0))^*=(\beta1,\beta2,\ldots,\beta n)^*$.

Тоді метод простих ітерацій набере вигляду.

$$\begin{cases} X^{(0)} = \beta \\ X^{(1)} = \beta + \alpha X^{(0)} \\ X^{(2)} = \beta + \alpha X^{(1)} \\ \ldots\ldots\ldots\ldots \\ X^{(k)} = \beta + \alpha X^{(k-1)} \end{cases}$$

Бачимо перевагу ітераційних методів у порівнянні, наприклад, з розглянутим вище методом Гауса. В обчислювальному процесі беруть участь тільки добутки матриці на вектор, що дозволяє працювати тільки з ненульовими елементами матриці, значно спрощуючи процес зберігання й обробки матриць. При цьому не відбувається накопичення похибки заокруглення.

Визначення збіжності ітераційного процесу можна знайти в 2.3-2.5. З огляду на сформульовані там теореми, має місце достатня умова збіжності методу простих ітерацій для СЛАР.

Метод Зейделя розв'язання СЛАР

Метод простої ітерації досить повільно збігається. Для його прискорення існує метод Зейделя. Суть його в тому, що при обчисленні компонентів $xi(к+1)$ вектора невідомих на $(k+1)$-ій ітерації використовуються $x1(к+1), x2\ (к+1),\ldots,xi\text{-}1(к+1)$, уже обчислені

на $(k+1)$-ій ітерації. Значення інших компонентів беруться з попередньої ітерації. Так само, як і у методі простих ітерацій, будується еквівалентна СЛАР (3.26) і за початкове наближення береться вектор правих частин $X0=(\beta1,\beta2,\ldots,\beta n)^*$. Тоді метод Зейделя для пошуку наближення $X(к+1)$ має вигляд

$$\begin{cases} x_1^{k+1} = \beta_1 + \alpha_{11}x_1^k + \alpha_{12}x_2^k + \ldots + \alpha_{1n}x_n^k \\ x_2^{k+1} = \beta_2 + \alpha_{21}x_1^{k+1} + \alpha_{22}x_2^k + \ldots + \alpha_{2n}x_n^k \\ x_3^{k+1} = \beta_3 + \alpha_{31}x_1^{k+1} + \alpha_{32}x_2^{k+1} + \alpha_{33}x_3^k + \ldots + \alpha_{3n}x_n^k \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ x_n^{k+1} = \beta_n + \alpha_{n1}x_1^{k+1} + \alpha_{n2}x_2^{k+1} + \ldots + \alpha_{nn-1}x_{n-1}^{k+1} + \alpha_{nn}x_n^k \end{cases}$$

Із цієї системи бачимо, що $Xk+1=\beta+BXk+1+CXk$, де B – нижня трикутна матриця з діагональними елементами, що дорівнюють нулю, а C - верхня трикутна матриця з діагональними елементами, відмінними від нуля, $\alpha=B+C$. Отже, □ □ □□ k□ $k\ E\ B\ X\ CX$ 1 ( ) ,

звідки □ 1 1 1 ( ) ( )□ □ □ X □ E □ $B\ CX$ □ E □ $B\ k\ k$.

Таким чином, метод Зейделя є методом простих ітерацій з матрицею правих частин □=*(E-B)-1C* і вектором правих частин *(E-B)-1β,* й, отже, збіжність і похибку методу Зейделя можна досліджувати за допомогою формул, виведених для методу простих ітерацій, у яких замість матриці □ підставлена матриця *(E-B)-1C*, а замість вектора правих частин - вектор *(E-B)-1β*. Для практичних обчислень важливо, що як достатні умови збіжності методу Зейделя можуть бути використані умови, наведені вище для методу простих ітерацій (□ □1 або, якщо

використовується еквівалентна СЛАР у формі (3.1), -діагональна перевага матриці *A*). У випадку виконання цих умов для оцінки похибки на k -ій ітерації можна використати вираз

Відзначимо, що, як і метод простих ітерацій, метод Зейделя може збігатися й при порушенні умови □ □1.

Приклад. Методом Зейделя розв’язати СЛАР із попереднього прикладу.

Розв’язання. Діагональна перевага елементів вихідної матриці СЛАР гарантує збіжність методу Зейделя. Ітераційний процес будуємо в такий спосіб:

$$x^{(0)} = (1{,}2 \quad 1{,}3 \quad 1{,}4)^T$$

$$\begin{cases} x_1^{(1)} = 1{,}2 - 0{,}1*1{,}3 - 0{,}1*1{,}4 = 0{,}93 \\ x_2^{(1)} = 1{,}3 - 0{,}2*0{,}93 - 0{,}1*1{,}4 = 0{,}974 \\ x_3^{(1)} = 1{,}4 - 0{,}2*0{,}93 - 0{,}2*0{,}974 = 1{,}0192 \end{cases}$$

$$\begin{cases} x_1^{(2)} = 1{,}2 - 0{,}1*0{,}974 - 0{,}1*1{,}0192 = 1{,}0007 \\ x_2^{(2)} = 1{,}3 - 0{,}2*1{,}0007 - 0{,}1*1{,}0192 = 0{,}998 \\ x_3^{(2)} = 1{,}4 - 0{,}2*1{,}0007 - 0{,}2*0{,}998 = 1{,}0003 . \end{cases}$$

Таким чином, уже на другій ітерації похибка $\left\|x^{(2)} - x^{(*)}\right\| < 10^{-2} = \varepsilon$, тобто метод Зейделя в цьому випадку збігається швидше ніж метод простих ітерацій.

Приклад . Розв'язання СЛАР **Ax=b**, отримане за допомогою вбудованої функції *lsolve* (пакет Mathcad).

Перевірка достатньої умови збіжності методу Зейделя

$$\text{normB}(B,n) := \left| \begin{array}{l} \text{for } i \in 1..n \\ \quad s_i \leftarrow \sum_{j=1}^{n} |B_{i,j}| \\ \max(s) \end{array} \right.$$

$\text{normB}(B,4) = 0.8$ *Достатня умова виконана.*

$$A := \begin{pmatrix} 15 & 3 & 4 & 5 \\ 2 & 16 & 4 & 5 \\ 2 & 3 & 17 & 5 \\ 2 & 3 & 4 & 18 \end{pmatrix} \qquad b := \begin{pmatrix} 13 \\ -1 \\ 17 \\ -50 \end{pmatrix}$$

Перетворення системи $Ax=b$ до вигляду $x=Bx+c$, зручного для ітерацій.

$x := \text{lsolve}(A,b)$

$$PB(A,n) := \left| \begin{array}{l} \text{for } i \in 1..n \\ \quad \text{for } j \in 1..n \\ \quad\quad B_{i,j} \leftarrow 0 \text{ if } i = j \\ \quad\quad B_{i,j} \leftarrow \frac{-A_{i,j}}{A_{i,i}} \text{ if } i \neq j \\ B \end{array} \right.$$

$$Pc(A,b,n) := \left| \begin{array}{l} \text{for } i \in 1..n \\ \quad c_i \leftarrow \frac{b_i}{A_{i,i}} \\ c \end{array} \right. \qquad c := Pc(A,b,4)$$

ORIGIN:= 1 - нумерація масивів починається з одиниці.

$B := PB(A,4)$

$$B = \begin{bmatrix} 0 & -0.2 & -0.2666666667 & -0.3333333333 \\ -0.125 & 0 & -0.25 & -0.3125 \\ -0.1176470588 & -0.1764705882 & 0 & -0.2941176471 \\ -0.1111111111 & -0.1666666667 & -0.2222222222 & 0 \end{bmatrix}$$

Тема 1. Метод простих ітерацій. Ітераційний метод Ньютона, модифікаційний метод Ньютона.

Метод простих ітерацій

Припустимо, що рівняння *f(x)=0* за допомогою деяких тотожних перетворень зведене до вигляду $x = \varphi(x)$. Відмітимо, що таке перетворення можна робити різними способами, і при цьому матимемо різні функції $\varphi(x)$ в правій частині рівняння. Рівняння *f(x)=0* еквівалентне рівнянню $x = x + \lambda(x) f(x)$ для будь-якої функції $\lambda(x) \neq 0$. Таким чином, можна взяти $\varphi(x) = x + \lambda(x) f(x)$ і при цьому вибрати функцію (або постійну) $\lambda \neq 0$ так, щоб функція $\varphi(x)$ задовольняла тим властивостям, які знадобляться нам для забезпечення знаходження кореня рівняння.

Для знаходження кореня рівняння $x = \varphi(x)$ виберемо деяке початкове наближення x_0 (розташоване, по можливості, близько до кореня). Далі будемо обчислювати подальші наближення $x_1, x_2, .., x_i, x_{i+1}, ...$ за формулами $x_1 = \varphi(x_0), x_2 = \varphi(x_1),$ і так далі, тобто використовуючи кожне обчислене наближення до кореня як аргумент функції $\varphi(x)$ в черговому обчисленні. Такі обчислення за однією і тією ж формулою $x_{i+1} = \varphi(x_i)$, коли отримане на попередньому кроці значення використовується на подальшому кроці, називаються ітераціями. Ітераціями називають часто і самі значення x_i, отримані в цьому процесі (тобто, в нашому випадку, послідовні наближення до кореня). Відмітимо той факт, що x^* – корінь рівняння $x = \varphi(x)$, означає, що x^* є абсциса точки перетину графіка $y = \varphi(x)$ з прямою $y=x$. Якщо ж при якому-небудь x_0 обчислено значення $x_1 = \varphi(x)$ і взято за новий аргумент функції, то це означає, що через точку графіка $(x_0, \varphi(x_0))$ проводиться горизонталь до прямої $y=x$, а звідти опускається перпендикуляр на вісь. Там і знаходитиметься новий аргумент x_1.

Прослідкуємо, як змінюються послідовні наближення x_i при різних варіантах взаємного розташування графіка $y = \varphi(x)$ і прямої $y=x$.

1) Графік $y = \varphi(x)$ розташований, принаймні в деякому околі кореня , що включає початкове наближення x_0, в деякому куті зі сторонами, що мають нахил менше $\frac{\pi}{4}$ до горизонталі (тобто сторони кута – прямі $y = f(x^*) \pm k(x - x^*)$, де $0<k<1$):

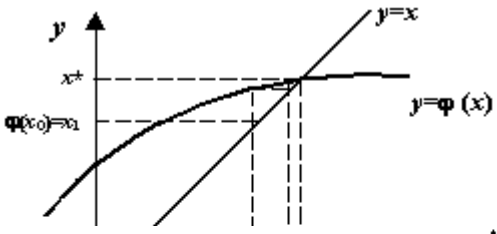
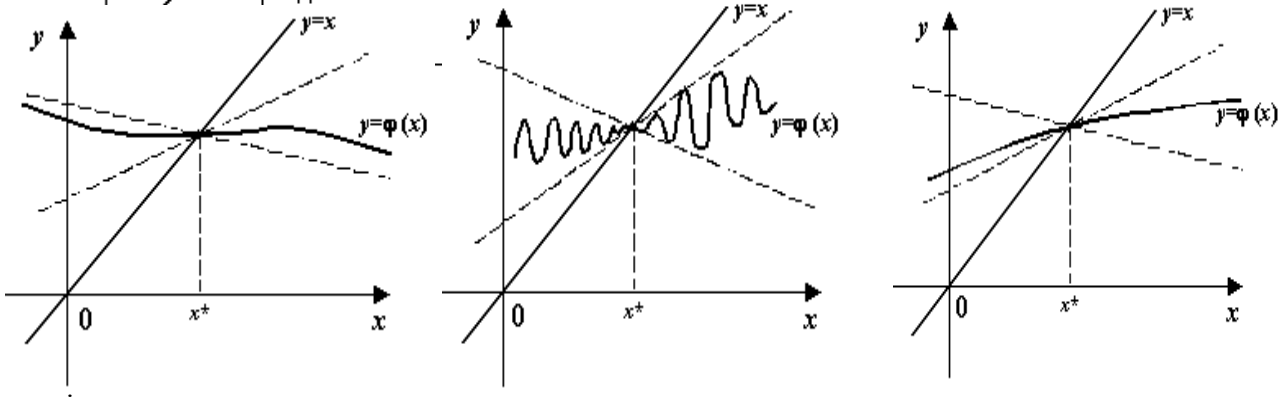


Рис.2.Графік перетинає пряму $y=x$ під малим кутом:



варіанти розташування.

Якщо припустити додатково, що функція $\varphi(x)$ має похідну $\varphi'(x)$, то цей випадок відповідає тому, що виконується нерівність $|\varphi'(x)| < 1$, при x, близьких до кореня x^*. Простежимо в цьому випадку за поведінкою послідовних наближень $x_0, x_1, ...$

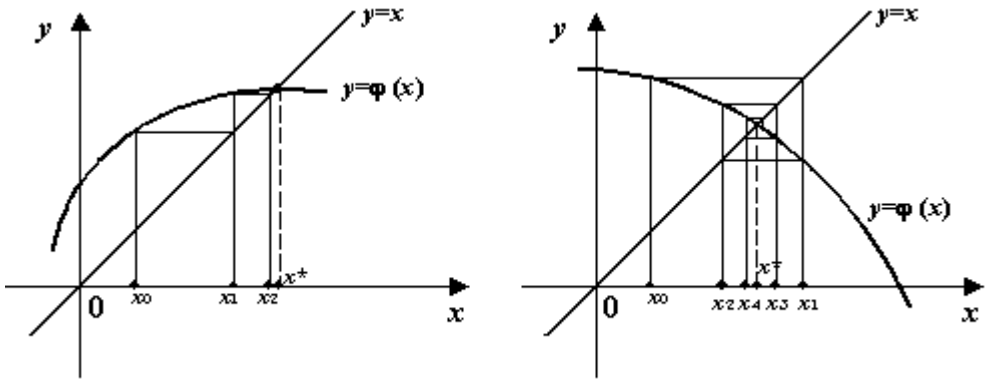


Рис.3.Наближення, що збігаються до кореня у випадку $|\varphi'(x)|<1$.

Ми бачимо, що кожне наступне наближення x_{i+1} буде в цьому випадку розташовано ближче до кореня x^*, ніж попереднє x_i. При цьому, якщо графік при $x<x^*$, лежить нижче за горизонталь $y=\varphi(x^*)$, а при $x>x^*$– вище за неї (що, у разі наявності похідної, вірно, якщо $0<\varphi'(x)<1$), то наближення x_i поводяться монотонно: якщо $x_o<x^*$, то послідовність $\{x_i\}$ монотонно зростає і прямує до x^*, а якщо $x_o>x^*$, то монотонно спадає і також прямує до x^*. Якщо ж графік функції $\varphi(x)$ лежить вище за горизонталь $y=\varphi(x^*)$ при $x<x^*$ і нижче за неї при $x>x^*$ (якщо $-1<\varphi'(x)<0$), то послідовні наближення поводяться інакше: вони "скачуть" навколо кореня, з кожним стрибком наближаючись до нього, але так само прямують до x^* при $i\to\infty$.

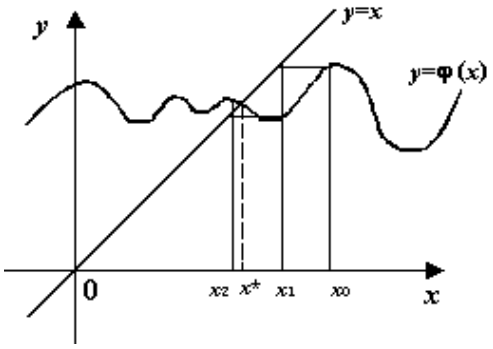


Рис.4.У випадку немонотонної функції ітерації, що сходяться, можуть поводитися нерегулярно

Відмітимо, що якщо функція $\varphi(x)$ не монотонна в околі точки x^*, то послідовні наближення можуть поводитися нерегулярно (тобто не монотонно і не потрапляючи почергово то лівіше, то правіше кореня, а роблячи стрибки відносно кореня при довільних номерах.

2) Графік $y=\varphi(x)$ розташований, принаймні в деякому околі кореня, що включає початкове наближення x_0, в деякому куті зі сторонами, що мають нахил більше $\frac{\pi}{4}$ до горизонталі (тобто сторони кута – прямі $y=f(x^*)\pm k(x-x^*)$, де $k>1$):

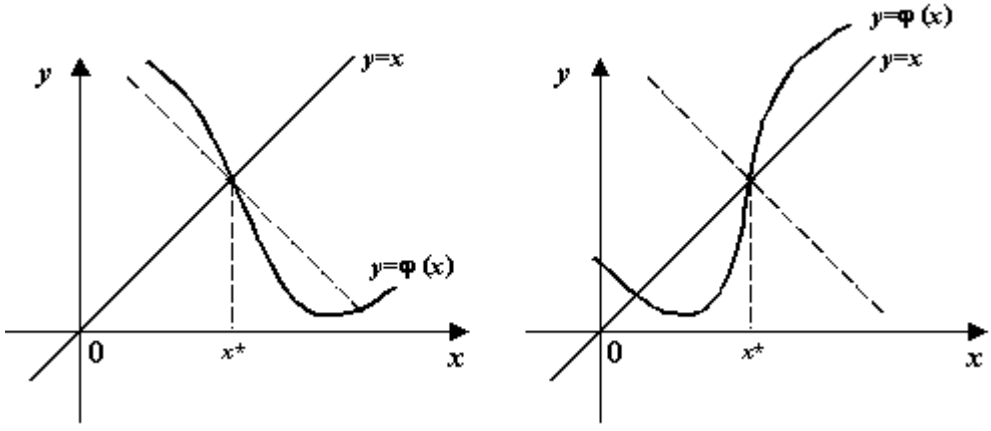


Рис.5.Графік перетинає пряму $y=x$ під великим кутом: варіанти розташування.

Якщо функція $\varphi(x)$ має похідну $\varphi'(x)$, то при x, близьких до кореня x^* виконується нерівність $|\varphi'(x)|>1$.

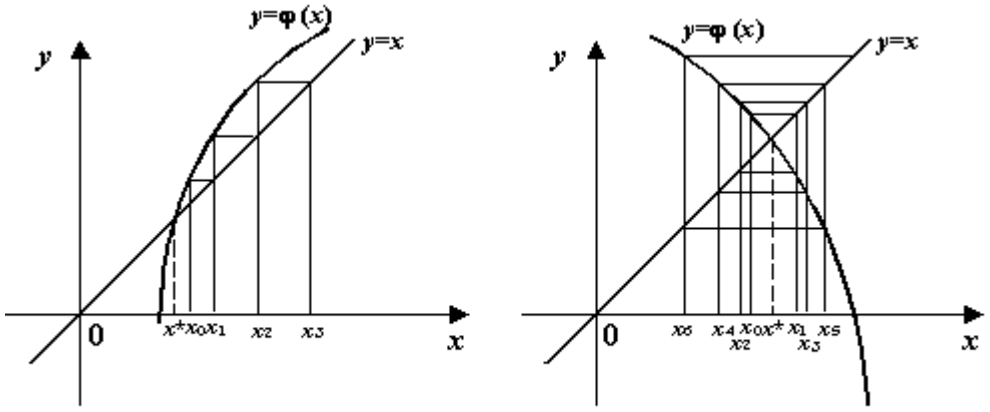
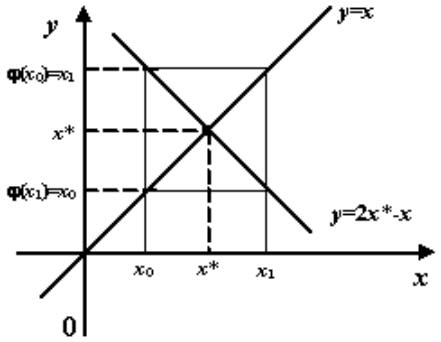


Рис.6.Послідовність $x_0, x_1, x_2, \ldots$ розбіжна у випадку $|\varphi'(x)| > 1$.

Кожна наступна ітерація x_{i+1} буде в цьому випадку розташована далі від кореня x^*, ніж попередня x_i. При цьому, залежно від того, чи перетинає графік пряму $y=x$ "знизу вгору" або "згори донизу", послідовність $\{x_i\}$ монотонно віддаляється від кореня x^* або ж ітерації віддаляються від x^*, потрапляючи почергово то справа, то зліва від кореня.

Ще одне зауваження: якщо не виконується ні умова $|\varphi'(x)| < 1$, ні $|\varphi'(x)| > 1$, то ітерації $x_1, x_2, .., x_i, x_{i+1}, \ldots$ можуть зациклюватися. Напри-лад, якщо рівняння має вигляд: x=2x*-x:



2. §II РОЗВ'ЯЗУВАННЯ НЕЛІНІЙНИХ РІВНЯНЬ

2.1.1. Постановка задачі:

Розглянемо задачу знаходження коренів рівняння

$$f(x) = 0, \qquad (1)$$

де $f(x)$ – задана функція дійсного змінного.

Розв'язування даної задачі можна розкласти на декілька етапів:

а) досліджена розташування коренів (в загальному випадку на комплексній площині) та їх кратність;

б) відділення коренів, тобто виділення областей, що містять тільки один корінь;

в) обчислення кореня з заданою точністю за допомогою одного з ітераційних алгоритмів.

Далі розглядаються ітераційні процеси, що дають можливість побудувати числову послідовність x_n, яка збігається до шуканого кореня x_* рівняння (1).

2.2. 1. Метод ділення проміжку навпіл (метод дихотомії)

Нехай $f \in C[a,b]$, $f(a)f(b) < 0$ і відомо, що рівняння (1) має єдиний корінь $x_* \in [a,b]$. Покладемо $a_0=a$, $b_0=b$, $x_0=(a_0+b_0)/2$. Якщо $f(x_0) = 0$, то $x_* = x_0$. Якщо $f(x_0) \neq 0$, то покладемо

$$a_{n+1} = \begin{cases} x_n, & \text{якщо } \operatorname{sign} f(a_n) = \operatorname{sign} f(x_n), \\ a_n, & \text{якщо } \operatorname{sign} f(a_n) \neq \operatorname{sign} f(x_n), \end{cases} \qquad (2)$$

$$b_{n+1}=\begin{cases}x_n, \text{ якщо } \operatorname{sign} f(b_n)=\operatorname{sign} f(x_n),\\ b_n, \text{ якщо } \operatorname{sign} f(b_n)\neq\operatorname{sign} f(x_n),\end{cases} \quad (3)$$

$$x_{n+1}=\frac{a_{k+1}+b_{k+1}}{2}, \quad n=0,1,2,\ldots, \quad (4)$$

і обчислимо $f(x_{n+1})$. Якщо $f(x_{n+1})=0$, то ітераційний процес зупинимо і будемо вважати, що $x_*\approx x_{n+1}$. Якщо $f(x_{n+1})\neq 0$, то повторюємо розрахунки за формулами (2)-(4).

З формул (2), (3) видно, що $\operatorname{sign} f(a_{n+1})=\operatorname{sign} f(a_n)$ і $\operatorname{sign} f(b_{n+1})=\operatorname{sign} f(b_n)$. Тому $f(a_{n+1})f(b_{n+1})<0$, а отже шуканий корінь x_* знаходиться на проміжку $[a_{n+1},b_{n+1}]$. При цьому має місце оцінка збіжності

$$|x_n-x_*|\le\frac{b-a}{2^{n+1}}. \quad (5)$$

Звідси випливає, що кількість ітерацій. які необхідно провести для знаходження наближеного кореня рівняння (1) з заданою точністю ε задовольняє співвідношенню

$$n\ge\left[\log_2\frac{b-a}{\varepsilon}\right]. \quad (6)$$

де [*c*] – ціла частина числа *c*.

Серед переваг даного методу слід відзначити простоту реалізації та надійність. Послідовність $\{x_n\}$ збігається до кореня x_* для довільних неперервних функцій *f*(*x*). До недоліків можна віднести невисоку швидкість збіжності методу та неможливість безпосереднього узагальнення систем нелінійних рівнянь.

2.3. 2. Метод простої ітерації

Метод простої ітерації застосовується до розв'язування нелінійного рівняння виду

$$x=\varphi(x). \quad (7)$$

Перейти від рівняння (1) до рівняння(7) можна багатьма способами, наприклад, вибравши

$$\varphi(x)=x+\psi(x)f(x), \quad (8)$$

де $\psi(x)$ – довільна знакостала неперервна функція.

Вибравши нульове наближення x_0, наступні наближення знаходяться за формулою

$$x_{n+1}=\varphi(x_n), \quad n=0,1,2,\ldots. \quad (9)$$

Наведемо достатні умови збіжності методу простої ітерації.

Теорема 1. *Нехай для вибраного початкового наближення* x_0 *на проміжку*

$$S=\{x:|x-x_0|\le\delta\} \quad (10)$$

функція φ(*x*) *задовольняє умові Ліпшиця*

$$|\varphi(x')-(x'')|\le q|x'-x''|, \quad x',x''\in S \quad (11)$$

де $0<q<1$, *і виконується нерівність*

$$|\varphi(x_0)-x_0|\le(1-q)\delta. \quad (12)$$

Тоді рівняння (7) має на проміжку S єдиний корінь x_*, *до якого збігається послідовність (9), причому швидкість збіжності визначається нерівністю*

$$|x_n-x_*|\le\frac{q^n}{1-q}|\varphi(x_0)-x_0|. \quad (13)$$

Зауваження: якщо функція φ(*x*) має на проміжку *S* неперервну похідну $\varphi'(x)$, яка задовольняє умові

$$|\varphi'(x)| \le q < 1, \qquad (14)$$

то функція φ(*x*) буде задовольняти умові (11) теореми 1.

З (13) можна отримати оцінку кількості ітерацій. які потрібно провести для знаходження розв'язку задачі (7) з наперед заданою точністю ε:

$$n \ge \left[\frac{\ln \dfrac{|\varphi(x_0) - x_0|}{(1-q)\cdot\varepsilon}}{\ln(1/q)} \right] + 1. \qquad (15)$$

Наведемо ще одну оцінку. що характеризує збіжність методу простої ітерації:

$$|x_n - x_*| \le \frac{q}{1-q}|x_n - x_{n-1}|. \qquad (16)$$

2.4. 3. Метод релаксації

Для збіжності ітераційного процесу (9) суттєве значення має вибір функції φ(*x*). Зокрема, якщо в (8) вибрати $\psi(x) = \tau = \text{const}$, то отримаємо метод релаксації.

$$x_{n+1} = x_n + \tau f(x_n), \quad n = 0,1,2,\ldots, \qquad (17)$$

який збігається при

$$-2 < \tau f'(x) < 0. \qquad (18)$$

Якщо в деякому околі кореня виконуються умови

$$f'(x) < 0, \quad 0 < m_1 < |f'(x)| < M_1, \qquad (19)$$

то метод релаксації збігаються при $\tau \in (0, 2/M_1)$. Збіжність буде найкращою при

$$\tau = \tau_{\text{опт}} = 2/(m_1 + M_1). \qquad (20)$$

При такому виборі τ для похибки $z_n = x_n - x_*$ буде мати місце оцінка

$$|z_n| \le q^n |z_0|, \quad n = 0,1,2,\ldots, \qquad (21)$$

де $q = (M_1 - m_1)/(M_1 + m_1)$.

Кількість ітерацій, які потрібно провести для знаходження розв'язку з точністю ε визначається нерівністю

$$n \ge \left[\frac{\ln(|z_0|/\varepsilon)}{\ln(1/q)} \right] + 1. \qquad (22)$$

Зауваження: якщо виконується умова $f'(x) > 0$, то ітераційний метод (17) потрібно записати у вигляді

$$x_{n+1} = x_n - \tau f(x_n).$$

2.5. 4. Метод Ньютона

Метод Ньютона застосовується до розв'язування задачі (1), де *f*(*x*) є неперервно-диференційованою функцією. На початку обчислень вибирається початкове наближення x_0. Наступні наближення обчислюються за формулою

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}, \quad n = 0,1,2,\ldots, \quad f'(x_n) \ne 0. \qquad (23)$$

З геометричної точки зору x_{n+1} є значенням абсциси точки перетину дотичної до кривої $y=f(x)$ в точці $(x_n, f(x_n))$ з віссю абсцис. Тому метод Ньютона називають також методом дотичних.

Теорема 2. *Якщо* $f(x) \in C^2[a,b]$, $f(a)f(b) < 0$, а $f''(x)$ *не змінює знака на* [a,b], *то виходячи з початкового наближення* $x_0 \in [a,b]$, *що задовольняє умові* $f(x_0)f''(x_0) > 0$,

можна обчислити методом Ньютона єдиний корінь x_* рівняння (1) з будь-якою степінню точності.

Теорема 3. *Нехай x_* – простий дійсний корінь рівняння (1) і $f(x)\in C^2(S)$, де* $S=\{x:|x-x_*|\le\delta\}$,

$$0<m_1=\min_{x\in S}|f'(x)|,\quad M_2=\max_{x\in S}|f''(x)|, \tag{24}$$

причому

$$q=\frac{M_2|x_0-x_*|}{2m_1}<1. \tag{25}$$

Тоді для $x_0\in S$ метод Ньютона збігається, причому для похибки справедлива оцінка

$$|x_n-x_*|\le q^{2^n-1}|x_0-x_*|. \tag{26}$$

З оцінки (26) видно, що метод Ньютона має квадратичну збіжність, тобто похибка на (n+1)-й ітерації пропорційна квадрату похибки на n-й ітерації.

Модифікований метод Ньютона

$$x_{n+1}=x_n-\frac{f(x_n)}{f'(x_0)},\quad n=0,1,2,\ldots \tag{27}$$

дозволяє не обчислювати похідну $f'(x)$ на кожній ітерації, а отже і позбутися можливого ділення на нуль. Однак цей алгоритм має тільки лінійну збіжність.

Кількість ітерацій, які потрібно провести для знаходження розв'язку задачі (1) з точністю ε задовольняє нерівності

$$n\ge\left[\log_2\left(\frac{\ln(|x_0-x_*|/\varepsilon)}{\ln(1/q)}\right)+1\right]+1. \tag{28}$$

Приклад 1. *Розв'язати рівняння*

$$x+\sin x-1=0 \tag{29}$$

методом ділення проміжку навпіл з точністю $\varepsilon=10^{-4}$.

Розв'язання. Спочатку знайдемо проміжок, де рівняння має єдиний корінь. Оскільки похідна функції $f(x)=x+\sin x-1$ не змінює знак, то корінь у рівнянні (29) буде один. Легко бачити, що $f(0)=-1<0$, а $f\left(\frac{\pi}{2}\right)=\frac{\pi}{2}>0$. Отже корінь належить проміжку $\left[0,\frac{\pi}{2}\right]$. Виберемо $a_0=0,\quad b_0=\frac{\pi}{2}$. Згідно з формулою (6), отримаємо, що для знаходження кореня з точністю 10^{-4} необхідно провести 13 інтеграцій. Відповідні значення x_n наведені в табл. 1.

Табл.2

n	x_n	$f(x_n)$
0	0785398E+00	0492505E+00
1	0392699E+00	0224617E+00
2	0589049E+00	0144619E+00
3	0490874E+00	0377294E-01
4	0539961E+00	0540639E-01
5	0515418E+00	0831580E-02
6	0503146E+00	0146705E-01
7	0509282E+00	0316819E-02
8	0512350E+00	0257611E-02

9	0510816E+00	0295467E-03
10	0511583E+00	0114046E-02
11	0511199E+00	0422535E-03
12	0511007E+00	0635430E-04
13	0510911E+00	0116016E-03

Приклад 2. *Знайти додатні корені рівняння*

$x^3-x-1=0$ (30)

методом простої ітерації з точністю $\varepsilon=10^{-4}$.

Розв'язання. Графічне дослідження рівняння (30) показує, що існує єдиний дійсний додатній корінь цього рівняння і він належить проміжку [1,2]. Оскільки на цьому проміжку $x \neq 0$, то рівняння (30) можна подати у вигляді

$$x = \sqrt{\frac{1}{x}+1}. \qquad (31)$$

Позначимо $\varphi(x) = \left(\frac{1}{x}+1\right)^{1/2}$. Перевіримо виконання умов теореми про збіжність методу простої ітерації. Виберемо $x_0=1{,}5$, тоді $\delta=0{,}5$. Розглянемо

$$\varphi'(x) = -\frac{1}{2\sqrt{x^3+x^4}}; \quad \max_{1\le x\le 2}|\varphi'(x)| = \frac{1}{2\sqrt{2}},$$

тобто $q = \frac{1}{2\sqrt{2}}$.

тоді $|\varphi(x_0)-x_0| = \left|\sqrt{\frac{2}{3}+1}-1{,}5\right| = 0{,}205, \quad (1-q)\delta = 0{,}5\cdot\left(1-\frac{1}{2\sqrt{2}}\right) \approx 0{,}3232,$

а отже умова (12) виконується. З формули (15) маємо, що кількість ітерацій, які необхідно провести для знаходження кореня з точністю $\varepsilon=10^{-4}$ повинна задовольняти умові $n \ge 8$. Відповідні значення x_n та $x_n-\varphi(x_n)$ наведені в табл.2.

Табл.2

n	x_n	$x_n-\varphi(x_n)$
0	0150000E+01	0209006E+00
1	0129099E+01	0411454E-01
2	0133214E+01	0901020E-02
3	0132313E+01	0193024E-02
4	0132506E+01	0415444E-03
5	0132464E+01	0892878E-04
6	0132473E+01	0191927E-04
7	0132471E+01	0417233E-05
8	0132472E+01	0953674E-06

Виходячи з нерівності (16) і отриманих результатів видно, що для досягнення заданої точності достатньо було провести 5 ітерацій ($n=5$). Взагалі слід відзначити, що апостеріорна оцінка (16) є більш точною і її використання може заощадити деяку кількість обчислень.

Приклад 3. *Методом релаксації знайти найменший за модулем від'ємний корінь рівняння*

$x^3-3x^2-1=0$ (32)

з точністю $\varepsilon=10^{-4}$.

Розв'язання. Спочатку виділимо корені рівняння (32) користуючись наступною таблицею

Табл.3

x	-4	-3	-2	-1	0	1	2	3
sign$f(x)$	–	–	+	+	–	+	+	+

З даної таблиці видно, що рівняння має три корені розташовані на проміжках [–3;–2], [–1;0], [0;1]. Будемо знаходити корінь на проміжку [–1;0]. Обчисливши значення $f(-0{,}5)=-0{,}375$ можна уточнити проміжок існування кореня [–1;–0,5].

Позначимо $f(x)=x^3-3x^2-1$. Тоді $f'(x)=3x^2+6x<0, \quad x\in[-1;-0{,}5]$ і є монотонно зростаючою функцією на [–1;–0,5] (оскільки $f''(x)=6x+6\geq 0$).

Тому $$m_1=\min_{x\in[-1;-0{,}5]}|f'(x)|=|f'(-0{,}5)|=2{,}25,$$

$$M_1=\max_{x\in[-1;-0{,}5]}|f'(x)|=|f'(-1)|=3.$$

Тоді, відповідно до формул (20) і (21), будемо мати вигляд

$$x_{n+1}=x_n+\tau_{\text{опт}}(x_n^3+3x_n^2-1). \qquad (33)$$

Вибравши за початкове наближення точку $x_0=-0{,}5$ будемо мати оцінку $|z_0|\leq 0{,}5$, а кількість ітерацій, які потрібно провести для знаходження розв'язку з точністю $\varepsilon=10^{-4}$ буде дорівнювати 5 (див. (22)). В табл. 4 наведені відповідні дані ітераційної послідовності:

Табл.4

n	x_n	$f(x_n)$
0	0500000E+00	0142857E+00
1	0642857E+00	0985700E-02
2	0652714E+00	0105500E-04
3	0652704E+00	0596046E-07
4	0652704E+00	0000000E+00
5	0652704E+00	0000000E+00

Із наведених даних видно, що необхідна точність досягається раніше 5-ї ітерації. Це досить характерно для апріорних оцінок типу (22).

Приклад 4. *Методом Ньютона знайти найменший додатній корінь рівняння*

$x^3+3x^2-1=0$ (34)

з точністю $\varepsilon=10^{-4}$.

Розв'язання. З табл. 3 видно, що рівняння (34) має єдиний додатній корінь, що належить проміжку [0;1]. обчислимо $f(0{,}5)=-0{,}125$. Тепер будемо шукати корінь на проміжку [0,5;1]. Нехай $f(x)=x^3+3x^2-1$. Тоді $f'(x)=3x^2+6x>0, \quad f''(x)=6x+6>0, \quad x\in[0{,}5;1]$.

$$m_1=\min_{x\in[0{,}5;1]}|f'(x)|=|f'(0{,}5)|=3{,}75,$$

$$M_2=\max_{x\in[-1;-0{,}5]}|f''(x)|=|f''(1)|=12.$$

Виберемо $x_0=1$, тоді $|x_0-x_*|\leq 0{,}5$. З формули (25) маємо

$$q = \frac{12 \cdot 0{,}5}{2 \cdot 3{,}75} = 0{,}8 < 1.$$

Тобто всі умови теореми про збіжність методу Ньютона виконані. З формули (28) маємо, що для досягнення заданої точності достатньо провести 7 ітерацій. Відповідні обчислення наведені в табл. 5.

Табл.5

n	x_n	$f(x_n)$
0	01000000E+01	03000000E+01
1	06666667E+00	06296297E+00
2	05486111E+00	06804019E-01
3	05323902E+00	01218202E-02
4	05320890E+00	04395228E-06
5	05320889E+00	04230802E-07
6	05320889E+00	04230802E-07
7	05320889E+00	04230802E-07

2.5.1. Задачі

Знайти одним з ітераційних методів дійсні корені рівнянь з точністю ε *(наприклад* $\varepsilon=10^{-4}$*).*

48) $x^3 - 5x^2 + 4x + 0{,}092 = 0$

49) $x^3 - 4x^2 - 7x + 13 = 0$

50) $x^4 + x^3 - 6x^2 + 20x - 16 = 0$

51) $x^3 + \sin x - 12x + 1 = 0$

52) $x^3 - 10x^2 + 44x + 29 = 0$

53) $x + \sin x - 12x = 0{,}25$

54) $3x + \cos x + 1 = 0$

55) $x^3 - 3x^2 - 17x + 22 = 0$

56) $x^4 - 2x^3 - 3{,}74x^3 + 8{,}18x - 3{,}48 = 0$

57) $x^2 + 4\sin x - 1 = 0$

58) $x^3 + 4\sin x = 0$

59) $x^4 - 10x^3 + 48{,}16x^2 + 108{,}08x + 70{,}76 = 0$

60) $x^4 - 3x^3 + 20x^2 + 44x + 54 = 0$

61) $x^3 - 3x^2 - 14x - 8 = 0$

62) $x^3 - x - 1 = 0$

63) $3x - \cos x - 1 = 0$

64) $3x^2 - \cos^2 \pi x = 0$

65) $x^2 + 4\sin x = 0$

66) $(x-1)^3 + 0{,}5e^x = 0$

67) $x^3 + 4x - 6 = 0$

68) $x^3 - 2x^2 + x + 1 = 0$

69) $x^2 \lg x - 1 = 0$

70) $x^3 + 6x^2 + 9x + 2 = 0$

71) $\mathrm{sh}x - 12\mathrm{th}x - 0{,}311 = 0$

72) $e^x - 2(x-1)^2 = 0$

73) $e^{-x} + x^2 - 2 = 0$

74) $x^4 + 4x - 2 = 0$

75) $x^4 + 2x - 1 = 0$

76) $x^3 - x^2 + x - 3 = 0$

77) $x^5 + x - 3 = 0$

78) $x^7 + x + 4 = 0$

79) $2^x + x^2 - 1{,}15 = 0$

80) $3^{-x} - x^2 + 1 = 0$

81) $x^4 - 2x^3 + x^2 - 2x + 1 = 0$

82) $x^5 - 5x + 2 = 0$

83) $x^7 + 6x - 5 = 0$

84) $x^4 + 2x - 2 = 0$

85) $(x-1)^2 - \sin 2x = 0$

86) $x^4 + 2x^2 - 6x + 2 = 0$

87) $x^5 - 3x^2 + 1 = 0$

88) $5x^3 + 2x^2 - 15x - 6 = 0$

89) $x^6 - 3x^2 + x - 1 = 0$

90) $(x-1)^2 - 0{,}5e^x = 0$

91) $3x^4 + 4x^3 - 12x^2 - 5 = 0$

92) $x^2 \cos 2x = 1$

93) $x^2 - 3 + 0{,}5^x = 0$

94) $x^2 - 10\sin x = 0$

Тема 2. Метод січних. Метод градієнтного спуску. Метод релаксацій.

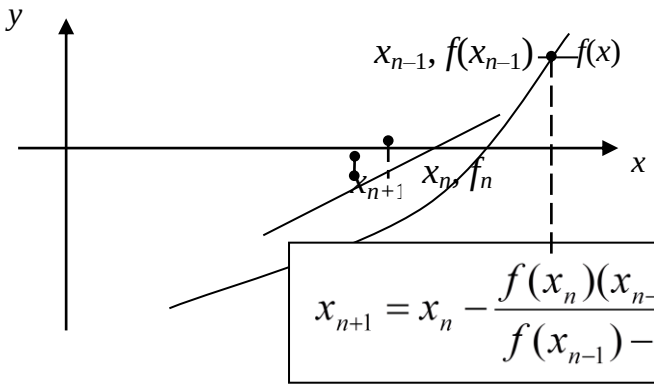
Якщо знаходження $f'(x)$ коштує дорогого, або неможливе, метод січних є кращим вибором, ніж метод Ньютона.

В цьому алгоритмі починають з двома початковими числами x_n та x_{n-1}. Абсциси *n, n-1* вибирають по одну сторону від кореня. На наступне уточнення x_{n+1} одержують з x_n та x_{n-1} як єдиний нуль лінійної функції, що приймає значення $f(x_n)$ в x_n та $f(x_{n-1})$ в x_{n-1}. Ця лінійна функція являє собою січну до кривої $f(x)$, що проходить через її точки з абсцисами x_n та x_{n-1} – звідси назва методу січних.

де $f_n = f(x_n)$. Праву частину краще не зводити до спільного знаменника.

Оскільки крок методу січних вимагає лишень одного обчислення функції, цей метод можна оцінити як більш швидкий в порівнянні з методом Ньютона.

Схема алгоритму для цього методу така ж, як і для методу Ньютона (дещо інший вигляд має ітераційна формула).



$$\frac{y - f(x_n)}{f(x_{n-1}) - f(x_n)} = \frac{x - x_n}{x_{n-1} - x_n}$$

$$x_{n+1} = x_n - \frac{f(x_n)(x_{n-1} - x_n)}{f(x_{n-1}) - f(x_n)}$$

$$x_{n+1} = x_n - \frac{f_n}{(f_{n-1} - f_n)/(x_{n-1} - x_n)};$$

Слід мати на увазі, що поблизу кореня x^* значення $f(x_n)$ та $f(x_{n-1})$ мало відрізняються, при діленні на їх різницю в методі виникає втрата значущих цифр, тому краще проводити обчислення за такою формулою:

$$x_{n+1} = x_n - \frac{f(x_n)}{[f(x_{n-1}) - f(x_n)]/(x_{n-1} - x_n)}.$$

Сутність градієнтного методу оптимізації полягає в тому, що задаються довільно, або виходячи з наявної апріорної інформації про положення точки екстремуму, початковим значенням вектора незалежних змінних $\vec{u}^{(0)}$.

Потім виконується зміна $\vec{u}^{(0)}$ на $\Delta\vec{u}^{(0)}$,тобто роблять крок $\Delta\vec{u}^{(0)}$ з метою наблизитися до точки екстремуму u_{OPT}. Потім роблять новий крок $\Delta\vec{u}^{(1)}$ і т.д.

Таким чином, на кожній ітерації обчислюється значення вектора для наступної ітерації:

$$\vec{u}^{(k+1)} = \vec{u}^{(k)} + \Delta\vec{u}^{(k)}.$$

Оскільки напрямок вектора градієнта вказує напрямок найшвидшого збільшення функції, то кроки Δu виконують у напрямку градієнта при пошуку максимуму й антиградієнта при пошуку мінімуму. Надалі, для визначеності, будемо розглядати задачу на мінімум. Тоді $\Delta\vec{u}^{(k)} = -\lambda\vec{S}^{(k)}$, де λ- множник, що визначає величину кроку $\Delta\vec{u}^{(k)}$; $\vec{S}^{(k)}$-одиничний вектор градієнта; k - номер ітерації. Знак "-" указує на напрямок антиградієнта.

У такий спосіб:

$$\vec{u}^{(k+1)} = \vec{u}^{(k)} - \lambda\vec{S}^{(k)}.$$

Алгоритм градієнтного пошуку часто застосовують у наступному виді:

$$\vec{u}^{(k+1)} = \vec{u}^{(k)} - h\nabla f(\vec{u}^{(k)}). \qquad (1.1)$$

У цьому випадку величина кроку $h\nabla f(\vec{u}^{(k)})$ змінюється автоматично відповідно до зміни величини градієнта.

Величина h зветься параметром кроку й залишається постійною. Алгоритм має ту перевагу, що при наближенні до точки мінімума довжина кроку автоматично зменшується.

Ітераційна формула (1.1) може бути записана в наступній формі:

$$\begin{bmatrix} u_1^{(k+1)} \\ --- \\ --- \\ --- \\ u_n^{(k+1)} \end{bmatrix} = \begin{bmatrix} u_1^{(k)} \\ --- \\ --- \\ --- \\ u_n^{(k)} \end{bmatrix} - h \begin{bmatrix} \dfrac{\partial f(u^{(k)})}{\partial u_1} \\ ----- \\ \dfrac{\partial f(u^{(k)})}{\partial u_n} \end{bmatrix},$$

або в скалярному виді:

$$u_i^{(k+1)} = u_i^{(k)} - h\frac{\partial f(\vec{u}^{(k)})}{\partial u_i} = u_i^{(k)} - h \cdot grad_{u_i} f(\vec{u}^{(k)}) \cdot$$

1.2.3 Вплив величини кроку на градієнтний пошук.

Питання вибору величини кроку є досить важливим і в остаточному підсумку визначає працездатність і швидкість збіжності алгоритму.

Якщо розмір кроку обраний занадто малим, то рух до оптимуму буде довгим через необхідність обчислення частинних похідних у багатьох точках.

При великому кроці в районі оптимуму можуть виникнути незатухаючі коливання незалежних змінних і знижується точність знаходження екстремуму.

При дуже великому кроці можливі розбіжні коливання.

На Рис зображені лінії постійного рівня функції $f(u_1,u_2)$.

Процес пошуку при великому h зображений послідовністю точок A_0, A_1, A_2, A_3.

Змістовий модуль 7. Апроксимація функцій.

Тема 1. Поняття про наближення функцій. Інтерполювання функції. Інтерполювання за Лагранжем.

АПРОКСИМАЦІЯ ФУНКЦІЙ

Поняття про наближення функцій

Нехай величина "y" є функцією аргумента "x", тобто будь−якому значенню "x" з області визначення поставлено у відповідність значення "y".

На практиці досить часто бувають випадки, коли неможливо записати зв'язок між "x" та "y" у вигляді деякої залежності $y = f(x)$. Най-більш поширеним випадком, коли вид зв'язку між параметрами x та y невідомий, є задання цього зв'язку у вигляді таблиці $\{x_i, y_i\}$. Це означає, що дискретній множині значень аргумента $\{x_i\}$ поставлена у відповідність множина значень функції $\{y_i\}$ ($i=\overline{0,n}$). Цими значеннями можуть бути, наприклад, експериментальні дані. На практиці можуть бути потрібні значення величини у і в інших точках, відмінних від вузлів x_i. Однак одержати ці значення можна тільки експериментальним шляхом, що не завжди зручно і вигідно.

З точки зору економії часу та засобів доцільно було б використати наявні табличні дані для наближеного обчислення шуканого параметра "y" при будь−якому значенні (з деякої області, звичайно) визначального параметра "x", оскільки точний зв'язок $y = f(x)$ невідомий.

Цій меті служить задача про наближення (апроксимацію) функцій:

– дану функцію $f(x)$ потрібно наближено замінити (апроксимувати) деякою функцією $\varphi(x)$ так, щоб відхилення (в певному розумінні) $\varphi(x)$ від $f(x)$ в заданій області було найменшим. При цьому функція $\varphi(x)$ називається апроксимуючою.

Наприклад, в тому випадку, коли функція $f(x)$ задається у вигляді таблиці значень, задача апроксимації полягає в наступному: за табличними даними підібрати таку аналітичну залежність $\varphi(x)$, яка мала б просту структуру, згладжувала б особливості заданої експериментальної таблиці і найкращим чином відбивала б загальний хід зміни $f(x)$ в середньому. Тобто основна мета апроксимації – одержати швидкий (економний) алгоритм обчислення значень $f(x)$ для значень x, що не містяться в таблиці даних. Основне питання апроксимації – як вибрати $\varphi(x)$ і як оцінити відхилення $\varphi(x)$ від $f(x)$.

На практиці досить часто $\varphi(x)$ вибирається з класу алгебраїчних поліномів (многочленів)

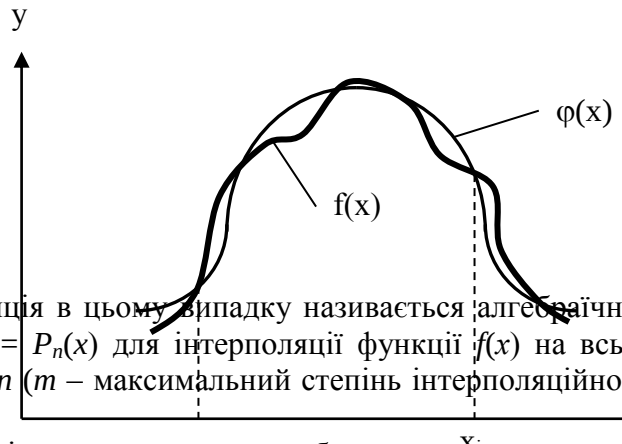
$$\varphi(x)=a_0 + a_1x + a_2x^2 +\ldots+ a_mx^m \qquad (1)$$

Якщо початкова функція задана таблично, тобто на множині окремих точок, то апроксимація називається точковою. Якщо ж початкова функція задана на неперервній множині точок (наприклад, на відрізку $[a; b]$), то апроксимація називається інтегральною (неперервною).

Одним з основних типів точкової апроксимації є інтерполяція. Вона полягає в наступному: для даної функції $y = f(x)$ будуємо функцію $\varphi(x)$, яка в заданих точках x_i ($i=\overline{0,n}$) приймає ті ж значення, що і функція $f(x)$, тобто

$$\varphi(x_i) = f(x_i),$$

а в решті точок відрізку $[a; b]$ з області визначення $f(x)$, наближено представляє $f(x)$ з деякою похибкою. Точки x_i називають вузлами інтерполяції, а $\varphi(x)$ – інтерполюючою функцією. Найчастіше інтерполюючу функцію $\varphi(x)$ виражають через алгебраїчний многочлен степені m.



Інтерполяція в цьому випадку називається алгебраїчною. Якщо використовується один многочлен $\varphi(x) = P_n(x)$ для інтерполяції функції $f(x)$ на всьому інтервалі зміни аргумента x, тобто коли $m = n$ (m – максимальний степінь інтерполяційного многочлена), то це – глобальна інтерполяція.

Інтерполяційні многочлени можуть будуватися окремо для різних частин інтервалу зміни x. В цьому випадку маємо кускову (локальну) інтерполяцію. Як правило, інтерполяційні многочлени використовують для апроксимації функцій у проміжних точках між крайніми вузлами інтерполяції, тобто $x_0 < x < x_n$. Однак іноді вони використовуються і для наближеного обчислення функції зовні інтервалу ($x < x_0$, $x > x_n$). Це наближення називається екстраполяцією.

Таким чином, при інтерполюванні основною умовою є проходження графіка інтерполяційного многочлена через дані значення функції у вузлах інтерполяції. Однак виконання цієї умови в деяких випадках є недоцільним. Наприклад, при великому числі вузлів інтерполяції одержуємо високу степінь полінома у випадку глобальної інтерполяції (це пов'язано з рядом неприємностей – осциляція функції). Крім того, табличні дані можуть містити в собі похибки (якщо ці дані одержані шляхом вимірювань). Отже, інтерполюючий многочлен теж повторював би ці похибки. Вихід із цього становища може бути знайдений вибором такого многочлена, графік якого близько проходить від даних точок.

Поняття "близько" уточнюється при розгляді окремих видів наближення.

Середньо-квадратичне наближення. Степінь полінома m при цьому, як правило, значно менша від n. На практиці не вище 5,6. Мірою відхилення многочлена $\varphi(x)$ від заданої функції $f(x)$ на множині точок (x_i, y_i) ($i = \overline{0, n}$) є величина S, яка дорівнює сумі квадратів різниць між значеннями многочлена та функції в даних точках

$$S = \sum_{s=0}^{n} \left[\varphi(x_i) - y_i \right]^2$$

При побудові апроксимуючого многочлена потрібно підібрати коефіцієнти $a_0, a_1, \ldots, a_m$ так, щоб величина S була мінімальна. В цьому полягає ідея методу найменших квадратів.

Рівномірне наближення. В багатьох випадках, особливо при обробці експериментальних даних, середньоквадратичне наближення зручне, оскільки воно згладжує деякі неточності функції $f(x)$ і дає достатньо правильне уявлення про неї. Однак, іноді ставиться більш жорстка умова і вимагається, щоб у всіх точках деякого відрізку $[a, b]$ модуль відхилення многочлена $\varphi(x)$ від $f(x)$ був менший від деякого ε

$|f(x) - \varphi(x)| < \varepsilon$, $a \le x \le b$

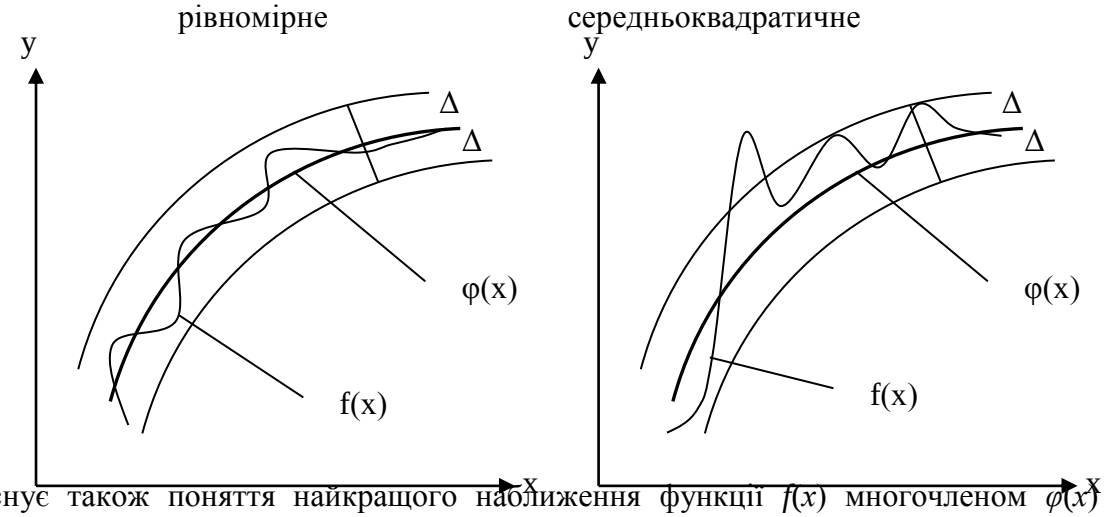
В цьому випадку маємо рівномірну апроксимацію. Тепер введемо такі поняття. Абсолютним відхиленням Δ многочлена $\varphi(x)$ від функції $f(x)$ на відрізку $[a, b]$ називається максимальне значення абсолютної різниці між ними на даному відрізку:

$\Delta = \max |f(x) - \varphi(x)|$, $a \le x \le b$

За аналогією можна ввести середньоквадратичне відхилення

$$\bar{\Delta} = \sqrt{\frac{S}{n}} .$$

На малюнку показано відмінність цих двох видів наближень.



Існує також поняття найкращого наближення функції $f(x)$ многочленом $\varphi(x)$ фіксованої степені m. В цьому випадку коефіцієнти многочлена $a_0, a_1, \dots, a_m$ слід вибирати так, щоб на заданому відрізку $[a, b]$ значення абсолютного відхилення Δ було мінімальне. Многолен $\varphi(x)$ при цьому називається многочленом найкращого рівномірного наближення.

3. ІНТЕРПОЛЯЦІЯ

Під апроксимацією розуміють операцію знаходження невідомих чисельних значень якоїсь величини за відомими її значеннями і чисельними значеннями інших величин, які пов'язані з розглядуваною.

Інтерполяція - частковий випадок апроксимації. Нехай в точках $x_0, x_1, x_2, \dots, x_n$ відомі значення $f(x_0), f(x_1), f(x_2) \dots f(x_n)$ деякої функції $f(x)$. Потрібно відновити функцію $f(x)$ при інших значеннях $x \ne x_i$ ($i = 0, 1, 2, \dots, n$). У цьому випадку будують достатньо просту для обчислення функцію $\varphi(x)$, яка в заданих точках $x_0, x_1, x_2, \dots, x_n$ приймає значення $f(x_0), f(x_1), f(x_2), \dots, f(x_n)$, а в решті точках відрізку $[a, b]$ (область визначення $f(x)$), наближено представляє $f(x)$ з деякою точністю. Задача побудови $\varphi(x)$ називається задачею інтерполювання. Найчастіше інтерполюючу функцію $\varphi(x)$ виражають через алгебраїчний многочлен деякої степені n.

Якщо аргумент x знаходиться зовні відрізку $[a, b]$, то поставлена задача називається екстраполюванням (екстраполяція).

Інтерполяція в цьому випадку називається алгебраїчною. Алгебраїчне інтерполювання функції $y = f(x)$ на відрізку $[a, b]$ полягає в наближеній заміні цієї функції на даному відрізку многочленом $P_n(x)$ степені n, тобто

$f(x) \approx P_n(x)$, (1)

причому в точках $x_0, x_1, x_2, \dots, x_n$, $f(x_i) = P_n(x_i)$, ($i = \overline{0, n}$).

Відмітимо, що двох різних інтерполяційних многочленів одної й тої ж степені n існувати не може. Якщо вважати протилежне, приходимо до висновку, що різниця двох таких многочленів, що є многочленом степені не вище n, має $n + 1$ корінь, а отже тотожно дорівнює нулю.

3.1. Інтерполяційний поліном Лагранжа

Поставимо задачу: знайти многочлен степені $P_n(x)$ степені n, котрий в $n + 1$ даних точках $x_0, x_1, x_2, \dots, x_n$ (ці точки називаються вузлами інтерполяції) приймає дані значення $y_0, y_1, \dots, y_n$.

Для побудови $P_n(x)$ спочатку розглянемо допоміжні (іноді їх називають фундаментальні) многочлени $Q_n^k(x)$, тобто многочлени n-ї степені відносно x, котрі задовільняють таким умовам:

$$\begin{cases} 0, & i \ne k \end{cases}$$

$Q_n^k(x_i) = \begin{cases} 1, & \text{при } i = k \end{cases}$, (к=$\overline{0,n}$).

Ця властивість означає, що, наприклад, многочлен $Q_n^0(x)$ приймає в точці x_0 значення, рівне одиниці, а в решті вузлів – нуль; многочлен $Q_n^1(x)$ в вузлі x_1 приймає значення 1, а в решті – нуль і т. д. В загальному випадку многочлен $Q_n^i(x)$ в вузлі x_i приймає значення 1, а в решті вузлів 0. Тоді шуканий многочлен:

$P_n(x) = y_0Q_n^0(x) + y_1Q_n^1(x) + y_2Q_n^2(x) + \ldots + y_nQ_n^n(x)$ (2).

Оскільки $x_0, x_1, x_2, \ldots, x_{k-1}, x_{k+1}, \ldots, x_n$ – нулі многочлена $Q_n^k(x)$, то

$Q_n^k(x) = c_k(x - x_0)(x - x_1)(x - x_2) \ldots (x - x_{k-1})(x - x_{k+1}) \ldots (x - x_n)$

(це просто інша форма запису полінома степені n).

Визначаючи c_k з умови $Q_n^k(x_k) = 1$, одержимо вираз для c_k (замість x підставляємо x_k)

$$c_k = \frac{1}{(x_k - x_0)(x_k - x_1)\ldots(x_k - x_{k-1})(x_k - x_{k+1})\ldots(x_k - x_n)} \qquad (2)$$

Тоді явний вираз для допоміжних многочленів

$$Q_n^k(x) = \frac{(x - x_0)(x - x_1)\ldots(x - x_{k-1})(x - x_{k+1})\ldots(x - x_n)}{(x_k - x_0)(x_k - x_1)\ldots(x_k - x_{k-1})(x_k - x_{k+1})\ldots(x_k - x_n)} = \prod_{i=0, i\neq k}^{n} \frac{x - x_i}{x_k - x_i} \qquad (3)$$

Формула (2) з врахуванням (3) приймає вигляд :

$$P_n(x) = \sum_{k=0}^{n} y_k \left(\prod_{i=0, i\neq k}^{n} \frac{x - x_i}{x_k - x_i} \right) = \qquad (4)$$

$$= \sum_{k=0}^{n} y_k \frac{(x - x_0)(x - x_1)\ldots(x - x_{k-1})(x - x_{k+1})\ldots(x - x_n)}{(x_k - x_0)(x_k - x_1)\ldots(x_k - x_{k-1})(x_k - x_{k+1})\ldots(x_k - x_n)}$$

Многочлен, що визначається за формулою (4) називається інтерполяційним многочленом Лагранжа, а допоміжні многочлени (3) – коефіцієнтами Лагранжа.

Введемо позначення

$$\omega(x) = (x - x_0)(x - x_1)\ldots(x - x_n)$$

Розглянемо похідну в точці $x_к$

$$\omega'(x_k) = (x_k - x_0)(x_k - x_1)\ldots(x_k - x_{k-1})(x_k - x_{k+1})\ldots(x_k - x_n)$$

Звідси

$$Q_n^k(x) = \frac{\omega(x)}{(x - x_k)\omega'(x)}$$

$$P_n(x) = \omega(x)\sum_{k=0}^{n} \frac{y_k}{(x - x_k)\omega'(x_k)}$$

Розглянемо інтерполяційну формулу Лагранжа для випадку рівновіддалених вузлів інтерполяції, тобто $x_1 - x_0 = x_2 - x_1 = \ldots = x_n - x_{n-1} = h$. Зробимо заміну $x = ht + x_0$, тоді

$t_0 = 0; t_1 = 1; t_2 = 2; \ldots t_n = n$

$x - x_k = h(t - k)$, $\omega(x) = n^{n+1}\omega^*(t)$

$\omega^*(t) = t(t - 1)(t - 2) \ldots (t - n)$

$\omega'(x_k) = (-1)^{n-k}k!(n - k)!h^n$

$$f(x) = f(ht + x_0) \approx t(t - 1)(t - 2) \ldots (t - n)^* \sum_{k=0}^{n} \frac{(-1)^{n-k} y_k}{(t - k)k!(n - k)!}$$

Приклад.

Знайти інтерполяційний многочлен Лагранжа для функції, заданої таблицею

x_i	-3	-1	1	2
y_i	8	6	4	18

При n=3, формула (4) приймає вигляд

$$P(3)=y_0\frac{(x-x_1)(x-x_2)(x-x_3)}{(x_0-x_1)(x_0-x_2)(x_0-x_3)}+y_1\frac{(x-x_0)(x-x_2)(x-x_3)}{(x_1-x_0)(x_1-x_2)(x_1-x_3)}+$$
$$+y_2\frac{(x-x_0)(x-x_1)(x-x_3)}{(x_2-x_0)(x_2-x_1)(x_2-x_3)}+y_3\frac{(x-x_0)(x-x_1)(x-x_2)}{(x_3-x_0)(x_3-x_1)(x_3-x_2)}$$

Підставляючи значення $x_к$ та $y_к$ ($к=\overline{0,3}$).

$$P_3(x)=8\frac{(x+1)(x-1)(x-2)}{(-3+1)(-3-1)(-3-2)}+6\frac{(x+3)(x-1)(x-2)}{(-1+3)(-1-1)(-1-2)}+$$
$$+4\frac{(x+3)(x+1)(x-2)}{(1+3)(1+1)(1-2)}+18\frac{(x+3)(x+1)(x-1)}{(2+3)(2+1)(2-1)}=x^3+3x^2-2x+2$$

$P_3(x)=x^3+3x^2-2x+2$

Тема 2. Інтерполювання за Ньютоном. Інтерполювання за Ермітом. Інтерполяція таблиць.

Інтерполяційний поліном Ньютона

Інтерполяційна формула Лагранжа має два суттєвих недоліки:

3) формула громіздка- кожен доданок є многочленом n-го степеня;

4) якщо з якоїсь причини додаються вузли інтерполювання (наприклад, якщо отримана інтерполяційна формула неточна), то всі обчислення необхідно повторювати знову – ні один із доданків формули Лагранжа не зберігається.

Розглянем форму запису інтерполяційного полінома $P_n(x)$, яка допускає уточнення результатів інтерполяції послідовним додаванням нових вузлів. При цьому будем використовувати таке поняття як розділені різниці функцій.

Нехай маємо функцію $f(x)$ і не обов"язково рівновіддалені вузли інтерполяції x_i (i=0, 1, 2, … , n).

Розділеними різницями 1-го порядку називають величини, які мають зміст, наприклад, середніх швидкостей зміни функції:

$$f(x_i;x_j)=\frac{f(x_j)-f(x_i)}{x_j-x_i}$$
$$f(x_0;x_1)=\frac{f(x_1)-f(x_0)}{x_1-x_0} \qquad (5)$$
$$f(x_1;x_2)=\frac{f(x_2)-f(x_1)}{x_2-x_1}$$

Розділені різниці другого порядку визначаються співвідношеннями

$$f(x_i;x_j;x_k)=\frac{f(x_j;x_k)-f(x_i;x_j)}{x_k-x_i}$$

$$f(x_0;x_1;x_2)=\frac{f(x_1;x_2)-f(x_0;x_1)}{x_2-x_0} \qquad (6)$$

$$f(x_1;x_2;x_3)=\frac{f(x_2;x_3)-f(x_1;x_2)}{x_3-x_1}$$

Аналогічно, розділена різниця *k*−го порядку визначається через розділені різниці (*k*−1) порядку за рекурентною формулою:

$$f(x_i;x_{i+1};...;x_{i+k})=\frac{f(x_{i+1};x_{i+2}...x_{i+k})-f(x_i;x_{i+1};...;x_{i+k-1})}{x_{i+k}-x_i} \qquad (7)$$

Тепер перейдемо безпосередньо до самого інтерполяційного полінома Ньютона. Маєм, наприклад, один вузол інтерполяції x_0.

Виходячи із визначення розділеної різниці 1−го порядку $f(x; x_0)$ маємо:

$$f(x_0;x)=\frac{f(x)-f(x_0)}{x-x_0}=\frac{f(x_0)-f(x)}{x_0-x}$$

$$f(x_0;x)=\frac{f(x)-f(x_0)}{x-x_0}=y_0 \Rightarrow f(x)=y_0+(x-x_0)f(x;x_0)$$

Для розділених різниць другого порядку (два вузли − x_0, x_1)

$$f(x;x_0;x_1)=\frac{f(x;x_0)-f(x_0;x_1)}{x-x_1}$$

$$f(x;x_0)=f(x_0;x_1)+(x-x_1)f(x;x_0;x_1)$$

Підставляючи це значення у формулу для f(x)

$$f(x)=y_0+(x-x_0)\left[f(x_0;x_1)+(x-x_1)f(x;x_0;x_1)\right]=$$

$$y_0+(x-x_0)f(x_0;x_1)+(x-x_0)(x-x_1)f(x;x_0;x_1)$$

Повторюючи цей процес, отримаємо (для n+1 вузлів інтерполяції):

$$f(x)=y_0+(x-x_0)f(x_0;x_1)+(x-x_0)(x-x_1)f(x_0;x_1;x_2)+$$
$$+...+(x-x_0)(x-x_1)...(x-x_{n-1})f(x_0;x_1;x_2...x_{n-1};x_n)+$$
$$+(x-x_0)(x-x_1)...(x-x_n)f(x;x_0;x_1;...x_n)= \qquad (8)$$
$$=P_n(x)+(x-x_0)(x-x_1)...(x-x_n)f(x;x_0;x_1;...x_n)$$

Оскільки $P_n(x)$ − інтерполяційний поліном для функції $f(x)$, то його значення у вузлах інтерполяції співпадають із значеннями функції $f(x)$ (а, значить, і співпадають і розділені різниці)

$P_n(x_i) = f(x_i) = y_i$, $(i=\overline{0,n})$,

оскільки залишковий член в цих вузлах

$$R_n(x)=(x-x_0)(x-x_1)...(x-x_n)f(x;x_0;x_1;...x_n)=0$$

(x приймає значення x_0, x_1, … , x_n, тому один із співмножників завжди рівний 0, через те залишковий член у вузлах інтерполяції дорівнює нулю).

Тому замість (8) можна записати

$$P_n(x) = y_0 + (x - x_0)f(x_0; x_1) + \ldots + (x - x_0)(x - x_1)\ldots$$
$$\ldots(x - x_{n-1})f(x_0; x_1; x_2 \ldots x_{n-1}; x_n) =$$
$$= y_0 + \sum_{k=1}^{n}(x - x_0)(x - x_1)\ldots(x - x_{k-1})f(x_0; x_1; \ldots x_k) = \quad (9)$$
$$= y_0 + \sum_{k=1}^{n}\left(\prod_{i=0}^{k-1}(x - x_i)\right)f(x_0; x_1; \ldots x_k)$$

Це і є інтерполяційний поліном Ньютона з розділеними різницями.

Для того, щоб пересвідчитись, що інтерполяційний поліном приймає значення y_i в вузлах інтерполяції x_i, візьмемо два вузли x_0 та x_1

$n = 2$ $\quad f(x) = y_0 + (x - x_0)f(x_0; x_1) + (x - x_0)(x - x_1)f(x; x_0; x_1)$

При $x = x_1$

$f(x_1) = y_0 + (x_1 - x_0)(f(x_1) - f(x_0))/(x_1 - x_0) = y_0 + f(x_1) - f(x_0)$;

$f(x_1) = f(x_1)$

Якщо маєм чотири вузли інтерполяції (n=3), то поліном Ньютона має вигляд:

$P_n(x) = y_0 + (x - x_0)f(x_0; x_1) + (x - x_0)(x - x_1)f(x_0; x_1; x_2) +$
$+ (x - x_0)(x - x_1)(x - x_2)f(x_0; x_1; x_2; x_3)$

Якщо ж маєм вже шість вузлів, тобто n=5, то йде просте нарощу-вання формули:

$P_n(x) = y_0 + (x - x_0)f(x_0; x_1) + (x - x_0)(x - x_1)f(x_0; x_1; x_2) +$
$+ (x - x_0)(x - x_1)(x - x_2)f(x_0; x_1; x_2; x_3) +$
$+ (x - x_0)(x - x_1)(x - x_2)(x - x_3)f(x_0; x_1; x_2; x_3; x_4) +$
$+ (x - x_0)(x - x_1)(x - x_2)(x - x_3)(x - x_4)f(x_0; x_1; x_2; x_3; x_4; x_5)$.

Приклад.

Знайти інтерполяційний поліном Ньютона:

x	−3	−1	1	2
y	8	6	4	18

При n = 3 інтерполяційний поліном Ньютона буде мати вигляд:

$P_n(x) = y_0 + (x - x_0)f(x_0; x_1) + (x - x_0)(x - x_1)f(x_0; x_1; x_2) +$
$+ (x - x_0)(x - x_1)(x - x_2)f(x_0; x_1; x_2; x_3)$

j	x_j	y_j	k=1	k=2	k=3
0	$x_0 = -3$	$y_0 = 8$	$\frac{6-8}{-1+3} = -1$	0	
1	$x_1 = -1$	$y_1 = 6$	$\frac{4-6}{1+1} = -1$	$\frac{14+1}{2+1} = 5$	$\frac{5-0}{2+3} = 1$
2	$x_2 = 1$	$y_2 = 4$	$\frac{18-4}{2-1} = 14$		
3	$x_3 = 2$	$y_3 = 18$			

$P_3(x) = 8 + (x + 3)(-1) + (x + 3)(x + 1)*0 + (x + 3)(x + 1)(x - 1)*1 =$
$= 8 - x - 3 + (x + 3)(x^2 - 1) = 5 - x + x^3 + 3x^2 - x - 3 =$
$= x^3 + 3x^2 - 2x + 2$

Перш ніж приступати до заповнення таблиці, розпишемо розділені різниці

Розділені різниці 1−го порядку

$$f(x_0;x_1)=\frac{y_1-y_0}{x_1-x_0}=\frac{6-8}{-1+3}=-1;$$

$$f(x_1;x_2)=\frac{y_2-y_1}{x_2-x_1}=\frac{4-6}{1+1}=-1;$$

$$f(x_2;x_3)=\frac{y_3-y_2}{x_3-x_2}=\frac{18-4}{2-1}=14$$

Розділені різниці 2−го порядку

$$f(x_0;x_1;x_2)=\frac{f(x_1;x_2)-f(x_0;x_1)}{x_2-x_0}=\frac{-1+1}{1+3}=0$$

$$f(x_1;x_2;x_3)=\frac{f(x_2;x_3)-f(x_1;x_2)}{x_3-x_1}=\frac{14+1}{2+1}=5$$

Розділені різниці 3−го порядку

$$f(x_0;x_1;x_2;x_3)=\frac{f(x_1;x_2;x_3)-f(x_0;x_1;x_2)}{x_3-x_0}=\frac{5-0}{2+3}=1$$

При написанні програми будемо користуватися наступним алгоритмом: позначимо через k – порядок розділених різниць ($k=\overline{1,n}$ – межі зміни k, де n – порядок (найвищий) інтерполюючого полінома), а через i – число розділених різниць ($i=\overline{n,k}$ – межі зміни i) для даного порядку k.

k=1 $y_3'=\frac{y_3-y_2}{x_3-x_2}\Rightarrow y_2'$ збереглося

$$y_2'=\frac{y_2-y_1}{x_2-x_1}=y_1$$

$$y_1'=\frac{y_1-y_0}{x_1-x_0}=y_0$$

k=2 $y_3''=\frac{y_3'-y_2'}{x_3-x_1}$

$$y_2''=\frac{y_2'-y_1'}{x_2-x_0}$$

k=3 $$y_3'''=\frac{y_3''-y_2''}{x_3-x_0}$$

y^k – кількість штрихів – це порядок

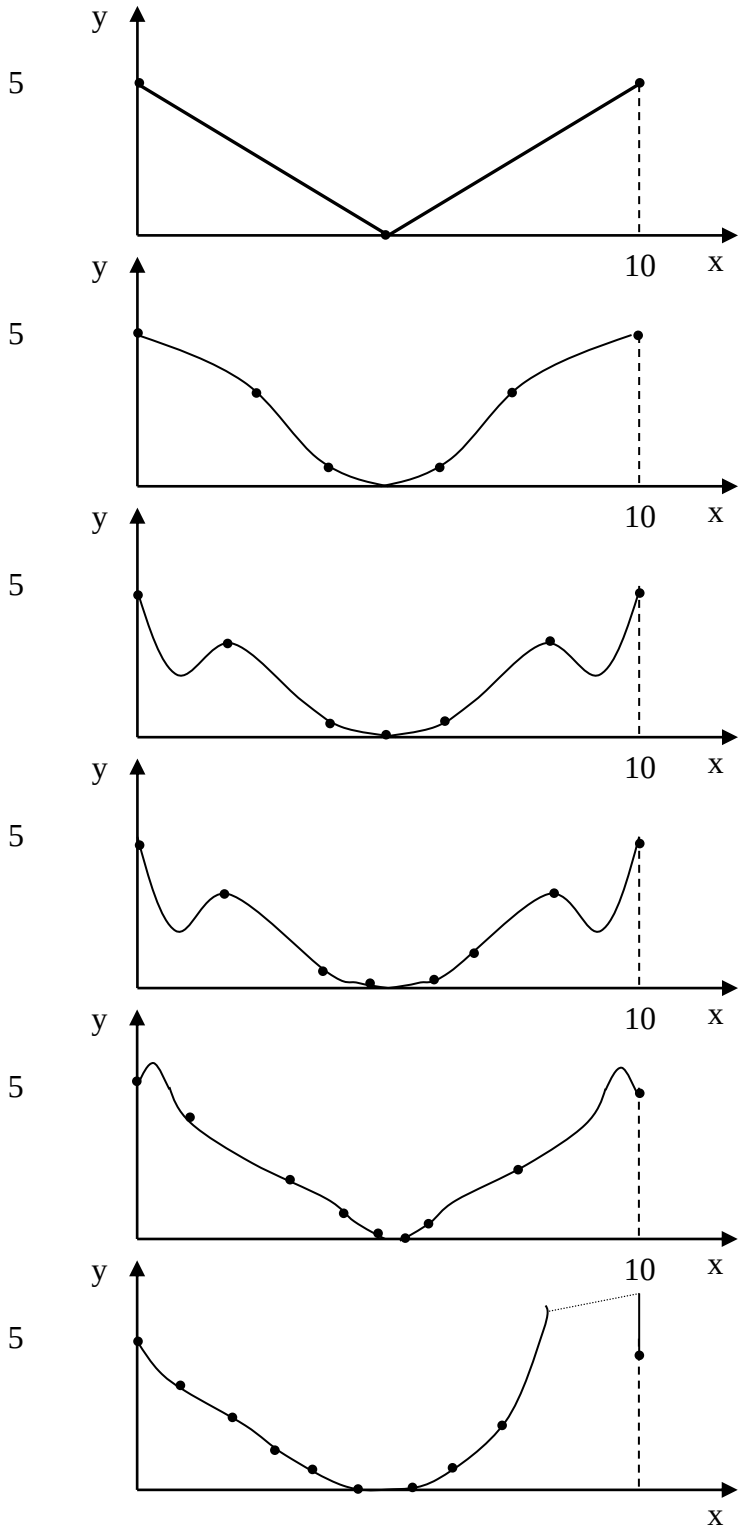
k = 1 до n

Для i = 0 до n ввести значення x_i, y_i

Для k = 1 до n

Для i = n до k з кроком –1

$y_i = (y_i - y_{i-1})/(x_i - x_{i-1})$
Тоді відповідно збережуться y_0, y_1', y_2'', y_3'''



Інтерполяція функції $y = |x - 5|$ з допомогою полінома Ньютона на 6-10 точках.

Підбір емпирічних формул

1. Характер експериментальних дослідних даних

При інтерполюванні функцій використовується відома умова $\varphi(x_i) = = f(x_i)$ – рівність значень інтерполяційного многочлена та даної функції у вузлах інтерполяції. Якщо $f(x_i)$ містить похибку, то апроксимуючий многочлен $\varphi(x_i)$ цю похибку повторить.

При обробці експериментальних даних, одержаних в результаті спостережень або вимірювань, потрібно мати на увазі похибки цих даних. Ці похибки можуть бути зумовлені недосконалістю вимірювального приладу, суб'єктивними причинами, різноманітними випадковими факторами.

Похибки експериментальних даних (ЕД) можна умовно розбити на 3 групи:

4) систематичні;
5) випадкові;
6) грубі.

Систематичні – дають, як правило, відхилення в одну сторону від істинного значення вимірювальної величини. Вони можуть бути сталими або закономірно змінюватись при повторі експерименту і їх причина та характеристики відомі. Систематичні похибки можуть бути викликані умовами експерименту (вологістю, температурою середовища і т. п.), дефектом вимірювального приладу, його поганим регулюванням (наприклад зміщення нуля) і т. п. Такі похибки усуваються наладкою апаратури або внесенням відповідних поправок.

Випадкові похибки – визначаються великим числом факторів, які не можуть бути усунуті або достатньо точно враховані при вимірюваннях або обробці результатів. Вони носять випадковий (несистематичний) характер, дають відхилення від середнього значення величини в різні сторони. Вони не можуть бути усунуті в експерименті. З точки зору теорії ймовірності математичне сподівання випадкової похибки дорівнює нулю.

Статистична обробка експериментальних даних дозволяє знайти значення випадкової похибки і довести її до деякого прийнятного рівня шляхом повторювання вимірювань.

Грубі похибки (помилки) явно спотворюють результат вимірювання. Вони надмірно великі і як правило зникають при повторі досліду. Вимірювання з такими похибками відкидаються і не враховується при остаточній обробці результатів вимірювань.

Таким чином, в ЕД завжди є випадкові похибки. Вони можуть бути зменшені шляхом багатократних повторних вимірювань. Однак для цього потрібні значні матеріальні та часові ресурси. Значно дешевше і швидше уточнені дані можна отримати шляхом спеціальної математичної обробки наявних результатів вимірювань (наприклад, статистична обробка дає значення розподілу похибок вимірювань, найбільш ймовірний діапазон зміни шуканої величини (довірчий інтервал) та інші параметри).

Ми розглянемо тільки визначення зв'язку між вхідними параметрами x та шуканою величиною y на підставі результатів вимірювань

2. Емпіричні формули

Маємо таблицю значень:

x_1	x_2	…	x_n
y_1	y_2	…	y_n

Необхідно знайти наближену залежність $y = f(x)$, значення якої при $x = x_i$ ($i=\overline{1,n}$), мало відрізняються від дослідних даних y_i. Наближена функціональна залежність $y = f(x)$, яка одержана на основі експериментальних даних, називається емпіричною формулою.

Одним із способів одержання емпіричних формул є метод найменших квадратів (МНК). Будем вважати, що тип емпіричної формули відомий (це, наприклад, пряма, парабола, многочлен чи інше) і її можна зобразити у вигляді

$$y = \varphi(x, a_0, a_1, \dots, a_m), \qquad (1)$$

де φ - відома функція;

$a_0, a_1, \dots, a_m$ – невідомі сталі параметри.

Задача полягає в тому, щоб визначити такі значення цих параметрів, при яких емпірична формула дає достатньо добре наближення таблично заданої функції.

Ідея МНК полягає в наступному. Запишемо суму квадратів відхилень для всіх точок x_i ($i=\overline{1,n}$)

$$S = \sum_{i=1}^{n} \left[\varphi(x_i, a_0, a_1, \dots, a_m) - y_i\right]^2 \qquad (2)$$

Параметри $a_0, a_1, \dots, a_m$ емпіричної формули (1) будем шукати з умови min функції $S = S(a_0, a_1, \dots, a_m)$. Оскільки тут параметри $a_0, a_1, \dots, a_m$ виступають в ролі незалежних змінних функції S, то її min знайдемо, прирівнюючи до нуля частинні похідні за цими змінними

$$\frac{\partial S}{\partial a_0} = 0; \quad \frac{\partial S}{\partial a_1} = 0; \quad \dots \quad ; \frac{\partial S}{\partial a_m} = 0; \qquad (3)$$

Розглянем випадок, коли за емпіричну функцію вибирають многочлен

$\varphi(x) = a_0 + a_1 x + a_2 x^2 + \dots + a_m x^m$ (4)

Тоді формула визначення суми квадратів відхилень S зобразиться так

$$S = \sum_{i=1}^{n} \left(a_0 + a_1 x_i + a_2 x_i^{\,2} + \dots a_m x_i^{\,m} - y_i\right)^2 \qquad (5)$$

Тоді система рівнянь для визначення $a_0, a_1, \dots, a_m$ з врахуванням (3) набере вигляду

$$\begin{aligned}
&\frac{\partial \mathrm{S}}{\partial \mathrm{a}_0} = 2\sum_{i=1}^{n} \left(a_0 + a_1 x_i + a_2 x_i^{\,2} + \dots a_m x_i^{\,m} - y_i\right) = 0 \\
&\frac{\partial \mathrm{S}}{\partial \mathrm{a}_1} = 2\sum_{i=1}^{n} \left(a_0 + a_1 x_i + a_2 x_i^{\,2} + \dots a_m x_i^{\,m} - y_i\right) x_i = 0 \\
&\text{– –} \qquad (6) \\
&\frac{\partial \mathrm{S}}{\partial \mathrm{a}_m} = 2\sum_{i=1}^{n} \left(a_0 + a_1 x_i + a_2 x_i^{\,2} + \dots a_m x_i^{\,m} - y_i\right) x_i^{\,m} = 0
\end{aligned}$$

Збираючи коефіцієнти при невідомих $a_0, a_1, \dots, a_m$, одержимо наступну систему рівнянь (2 перед знаком суми опускаєм – сталий множник, який не змінює коренів системи):

$$\begin{aligned}
&n a_0 + a_1 \sum_{i=1}^{n} x_i + a_2 \sum_{i=1}^{n} x_i^{\,2} + \dots + a_m \sum_{i=1}^{n} x_i^{\,m} = \sum_{i=1}^{n} y_i \\
&a_0 \sum_{i=1}^{n} x_i + a_1 \sum_{i=1}^{n} x_i^{\,2} + a_2 \sum_{i=1}^{n} x_i^{\,3} + \dots + a_m \sum_{i=1}^{n} x_i^{\,m+1} = \sum_{i=1}^{n} y_i x_i \\
&\text{– –} \qquad (7) \\
&a_0 \sum_{i=1}^{n} x_i^{\,m} + a_1 \sum_{i=1}^{n} x_i^{\,m+1} + a_2 \sum_{i=1}^{n} x_i^{\,m+2} + \dots + a_m \sum_{i=1}^{n} x_i^{\,2m} = \sum_{i=1}^{n} x_i^{\,m} y_i
\end{aligned}$$

Систему (7) можна записати в більш компактному вигляді:

$$\begin{aligned}
&c_0 a_0 + c_1 a_1 + c_2 a_2 + \dots + c_m a_m = d_0, \\
&c_1 a_0 + c_2 a_1 + c_3 a_2 + \dots + c_{m+1} a_m = d_1, \qquad (8) \\
&\text{– –}
\end{aligned}$$

$$c_m a_0 + c_{m+1} a_1 + c_{m+2} a_2 + \ldots + c_{2m} a_m = d_m,$$

де $c_j = \sum_{i=1}^{n} x_i^j, \qquad j = 0, 1, 2, \ldots, 2m$ (9)

$$d_k = \sum_{i=1}^{n} x_i^k y_i, \qquad k = 0, 1, 2, \ldots, m \tag{10}$$

Поліном (4) степені $m < n$, де n − число пар x_i, y_i забезпечує апроксимацію таблично заданої функції $y_i(x_i)$ з мінімальною середньоквадратичною похибкою:

$$E = \sqrt{\frac{\left(\sum_{i=1}^{n} \varepsilon_i^2\right)}{(n+1)}} \tag{11}$$

Якщо $m = n$, то має місце звичайна інтерполяція, тобто

$$\varphi(x_i) = y_i$$

Зауваження щодо побудови програми .

Система (8) − це система лінійних алгебраїчних рівнянь відносно невідомих a_0, a_1, … , a_m. Коефіцієнти при невідомих одержуються за формулами (9) та (10). Для обчислення і зберігання коефіцієнтів c_j потрібен масив із (2m+1) чисел, а для d_k − масив із (m + 1) чисел, де m − степінь полінома, яка задається на початку роботи програми.

Потрібно ввести в циклі ($i=\overline{1,n}$) пари значень x_i, y_i, потім сформувати коефіцієнти при невідомих c_j та вільні члени d_k .

Одержану таким чином систему лінійних алгебраїчних рівнянь розв'язати методом Гауса (з частковим вибором головного елемента), одержуючи значення параметрів a_0, a_1, … , a_m апроксимуючого полінома $\varphi(x)$.

На практиці використовується поліноміальна апроксимація за МНК з автоматичним вибором степені полінома. Алгоритм наступний: задається початкове значення m, потім шукається коефіцієнти полінома a_0, a_1, … , a_m , за формулою (11) обчислюється середньоквадратична похибка і порівнюється із заданою E1. Якщо E > заданої, степінь m збільшується на 1 і все повторюється. Обчислення припиняється при E < E1.

Тема 3. Похибка інтерполяції. Збіжність процесу інтерполяції. Інтерполяційні сплайни.

Змістовий модуль 8. Чисельне розв’язання диференційних рівнянь.

Тема 1. Основні поняття. Диференційні рівняння з однокроковим методом. Метод Ейлера і Рунге-Кутта, схеми Рунге-Кутта другого і четвертого порядку.

Метод Ейлера

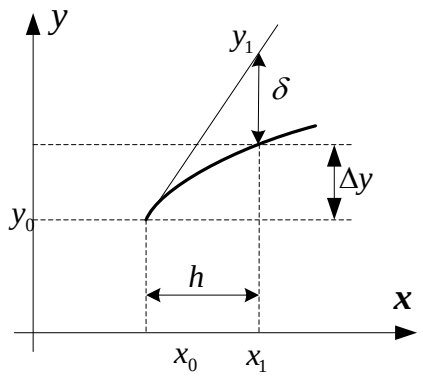
Однокрокові методи призначені для розв’язування диференціальних рівнянь першого порядку виду

$$\frac{dy}{dx} = f(x, y), y(x_0) = y_0 \tag{6}$$

Метод Ейлера є найпростішим методом розв’язування задачі Коші. Він дозволяє інтегрувати ДР першого порядку. Точність його не велика.

h - настільки мале, що значення функції y мало відрізняється від лінійної функції

$tg\alpha$ - тангенс кута нахилу дотичної в точці x_0



$$y_1 = y_o + h \cdot tg\alpha = y_o + hy'(x_0) = y_0 + hf(x_o, y_0)$$

Тобто крива заміняється дотичними. Рух відбувається не по інтегральній кривій, а по відрізках дотичної .

Метод Ейлера базується на розкладі функції y' в ряд Тейлора в околі точки x_0

$$y(x_0 + h) = y(x_0) + hy'(x_0) + \frac{1}{2!}h^2 y''(x_0) + + \frac{h^3}{3!}y'''(x_0) + ... + \frac{h^p}{p!}y^{(P)}(x_0) + ...$$

$$y(x_i + \Delta x) = y(x_i) + y'(x_i)\Delta x_i + 0(\Delta x_i^2)$$

Якщо h мале, то, члени розкладу, що містять в собі h^2, h^3 і т.д. є малими високих порядків і ними можна знехтувати.

Тоді $y(x_0 + h) \approx y(x_0) + hy'(x_0) = y(x_0) + hf(x_o, y_o)$

Похідну $y'(x_0)$ знаходимо з рівняння (6), підставивши в нього початкову умову. Таким чином можна знайти наближене значення залежної змінної при малому зміщенні h від початкової точки. Цей процес можна продовжувати, використовуючи співвідношення.

$$y_{n+1} = y_n + hy'(x_n) = y_n + hf(x_n, y_n),$$

роблячи як завгодно багато кроків.

Похибка методу має порядок h^2, оскільки відкинуті члени, що містять h в другій і вище степенях.

Недолік методу Ейлера - нагромадження похибок, а також збільшення об'ємів обчислень при виборі малого кроку h з метою забезпечення заданої точності.

В методі Ейлера на всьому інтервалі h тангенс кута нахилу дотичної приймається незмінним і рівним $y'(x_n)$. Очевидно, що це призводить до похибки, оскільки кути нахилу дотичної в точках x_n та $x_{n+1} = x_n + h$ різні. Точність методу можна суттєво підвищити, якщо покращити апроксимацію похідної.

Це можна зробити, якщо, наприклад, використати середнє значення похідної на початку та в кінці інтервалу.

Модифікований метод Ейлера

В модифікованому методі Ейлера (метод Ейлера з перерахунком) спочатку обчислюється значення функції в наступній точці за звичайним методом Ейлера.

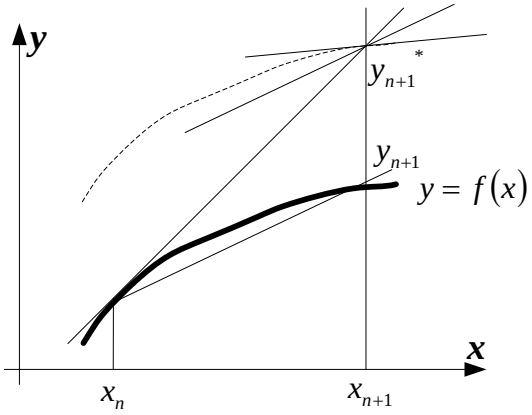
$$y^*_{n+1} = y_n + hf(x_n, y_n) \qquad (9)$$

Воно використовується для обчислення наближеного значення похідної в кінці інтервалу $f\left(x_{n+1},\ \ y^*_{n+1}\right)$.

Обчисливши середнє між цим значенням похідної та її значенням на початку інтервалу, знайдемо більш точне значення y_{n+1}:

$$y_{n+1} = y_n + \frac{1}{2}h\left[f\left(x_n, y_n\right) + f\left(x_{n+1}, y^*_{n+1}\right)\right] \tag{10}$$

Цей прийом ілюструється на рисунку.



В обчислювальній практиці використовується також метод Ейлера-Коші з ітераціями:

2) знаходиться грубе початкове наближення (за звичайним методом Ейлера)

$y^{(0)}_{n+1} = y_n + hf\left(x_n, y_n\right)$

3) будується ітераційний процес

$$y^k_{n+1} = y_n + \frac{h}{2}\left[f\left(x_n, y_n\right) + f\left(x_{n+1}, y^{(K-1)}_{n+1}\right)\right], k = 1,2,3... \tag{14}$$

Ітерації продовжують до тих пір, доки два послідовні наближення не співпадуть з заданою похибкою ε. Якщо після декількох ітерацій співпадіння нема, то потрібно зменшити крок h.

$\left|y^{(K)}_{n+1} - y^{(K-1)}_{n+1}\right| < \varepsilon$

Тобто в модифікованому методі Ейлера, в методі Ейлера-Коші з ітераціями спочатку (на першому етапі) знаходиться наближення для y_{n+1}, а потім воно вже коригується за формулами (10) або (14).

Метод Рунге – Кутта четвертого порядку

Метод Рунге-Кутта об'єднує ціле сімейство методів розв'язування диференціальних рівнянь першого порядку. Найбільш часто використовується метод четвертого порядку.

В методі Рунге-Кутта значення y_{n+1} функції y, як і в методі Ейлера, визначається за формулою

$$y_{n+1} = y_n + \Delta y_n \ (1)$$

Якщо розкласти функцію y в ряд Тейлора і обмежитись членами до h^4 включно, то приріст Δy можна записати у вигляді

$$\Delta y = y\left(x+h\right) - y\left(x\right) = hy'(x) + \frac{h^2}{2!}y''(x) + + \frac{h^3}{3!}y'''(x) + \frac{h^4}{4!}y^{IV}(x) \tag{11}$$

Замість того, щоб обчислювати члени ряду за формулою (11) в методі Рунге-Кутта використовують наступні формули.

$$y_{n+1} = y_n + \frac{K_1 + 2K_2 + 2K_3 + K_4}{6}$$

$$K_1 = hf\left(x_n, y_n\right)$$
$$K_3 = hf\left(x_n + \frac{1}{2}h, y_n + \frac{1}{2}K_2\right)$$
$$K_4 = hf\left(x_n + h, y_n + K_3\right)$$

Це метод четвертого порядку точності.

Похибка на кожному кроці має порядок h^5. Таким чином метод Рунге-Кутта забезпечує значно вищу точність ніж метод Ейлера, однак вимагає більшого об'єму обчислень в порівнянні з методом Ейлера. Це досить часто дозволяє збільшити крок h.

Деколи зустрічається інша форма представлення методу Рунге-Кутта 4-го порядку точності.

$$y_{n+1} = y_n + \frac{h}{6}\cdot\left(K_1 + 2\cdot K_2 + 2\cdot K_3 + K_4\right)$$
$$K_1 = f\left(x_n, y_n\right)$$
$$K_2 = f\left(x_n + \frac{h}{2}, y_n + \frac{h}{2}\cdot K_1\right)$$
$$K_3 = f\left(x_n + \frac{h}{2}, y_n + \frac{h}{2}K_2\right)$$
$$K_4 = f\left(x_n + \frac{h}{2}, y_n + \frac{h}{2}K_3\right)$$

В більшості стандартних програм ЕОМ найчастіше використовується (схема) метод четвертого порядку (Рунге-Кутта).

Тема 2. Багатокрокові методи, метод прогнозу і корекції. Метод Адамса. Задачі Коші.

У цих методах для обчислення значення нової точки використовується інформація про декілька значень, що отримані раніше. Для цього використовуються дві формули: прогнозу і корекції. Алгоритм обчислення для всіх методів прогнозу і корекції однаковий та зображений на рисунку4.3. Вказані методи відрізняються лише формулами і не мають властивості "самостартування", оскільки вимагають знання попередніх значень. Перш ніж використовувати метод прогнозу і корекції, обчислюють початкові дані за допомогою будь-якого однокрокового методу. Часто для цього використовують метод Рунге – Кутта.

Обчислення виконують таким чином. Спочатку за формулою прогнозу та початковим значенням змінних знаходять значення $y_{n+1}^{(0)}$. Індекс (0) означає, що значення, яке прогнозується, є одним із послідовності значень y_{n+1} по мірі їх уточнення. За значенням $y_{n+1}^{(0)}$ за допомогою початкового диференціального рівняння (4.1.) знаходять похідну $y_{n+1}^{(0)'} = f\left(x_{n+1}, y_{n+1}^{(0)}\right)$, яка після цього підставляється у формулу корекції для обчислення уточненого значення $y_{n+1}^{(j+1)}$. В свою чергу, за $y_{n+1}^{(j+1)}$ знаходять похідну $y_{n+1}^{(j+1)'} = f\left(x_{n+1}, y_{n+1}^{(j+1)}\right)$. Якщо це значення не достатньо близьке до попереднього, то воно вводиться у формулу корекції і ітераційний процес продовжується. У випадку близькості значень похідних визначається y_{n+1}, яке і є остаточним. Після цього процес повторюється на наступному кроці, на якому обчислюється y_{n+2}.

Зазвичай при виведенні формул прогнозу і корекції розв'язок рівняння розглядають як процес наближеного інтегрування, а самі формули отримують за допомогою методів чисельного інтегрування.

Якщо диференціальне рівняння $y' = f(x, y)$ проінтегрувати в інтервалі значень від xn до xn+k , то результат матиме вигляд

$$y(x_{n+k}) - y(x_n) = \int_{x_n}^{x_{n+k}} f(x, y)dx$$

.

Цей інтеграл не можна обчислити безпосередньо, тому що y(x) – невідома функція. Вибір методу наближеного інтегрування і буде визначати метод розв'язання диференціальних рівнянь. На етапі прогнозу можна використовувати будь-яку формулу чисельного інтегрування, якщо до неї не входить попереднє значення $y'(x_{n+1})$.

В таблицю 4.1 зведені найбільш розповсюджені формули прогнозу і корекції. Для більшості методів прогнозу і корекції оцінюють похибку, користуючись таким співвідношенням:

$$\Delta \le \frac{1}{5}\left[y_n^{(0)} - y_n^{(j)}\right]$$

Мірою похибки слугує і є $y_n^{(j)}$, що входить до алгоритму рисунку 4.3.

Часто в довідниках приводяться більш точні формули для оцінки похибки багатокрокових методів.

При виборі величини кроку можна скористатися умовою:

$$h < \frac{2}{M_2},$$

де $M_2 = \left|\frac{\partial f}{\partial y}\right|_{max}$.

Виконання цієї умови необхідно для збіжності ітераційного процесу відшукання розв'язку.

Однак у багатьох практичних випадках складність оцінки величини M_2 приводить до того, що найбільш зручним для вибору кроку є спосіб, побудований на оцінці D у процесі обчислень і зменшенні кроку, якщо похибка надто велика. При цьому необхідно враховувати, що оптимальне число ітерацій дорівнює двом.

Задача Коші

Задача Коші формулюється так:

Нехай задане ДР

$$\frac{dy}{dx} = f(x, y) \tag{5}$$

з початковими умовами $y(x_o) = y_0$. Потрібно знайти функцію $y(x)$, що задовольняє дане рівняння, та початкову умову. Для одержання чисельний розв'язку цієї задачі спочатку обчислюють значення похідної, а потім задаючи малий приріст $"x"$, переходять до нової точки

$x_1 = x_0 + h$

Положення нової точки визначають за нахилом кривої, обчисленому з допомогою ДР. Таким чином, графік чисельного розв'язку являє собою послідовність коротких прямолінійних відрізків, якими апроксимується істинна крива $y(x)$. Сам чисельний метод визначає порядок дій при переході від даної точки кривої до наступної.

Існують дві групи методів розв’язування задачі Коші.

2. Однокрокові методи. В них для знаходження наступної точки на кривій $y(x)$ потрібна інформація лише про попередній крок. (Однокроковими є метод Ейлера та методи Руте-Кутта.)
3. Багатокрокові (або методи прогнозування та коригування).

Для знаходження наступної точки кривої $y(x)$ вимагається інформація більш ніж про одну з попередніх точок. До них належать методи Адамса, Мілна, Хеммінга.

Це чисельні методи розв’язування ДР. Вони дають розв’язок у вигляді таблиці значень.