Міністерство освіти і науки України
Тернопільський національний економічний університет
Факультет комп’ютерних інформаційних технологій

АЛГОРИТМИ І СТРУКТУРИ ДАНИХ

навчальний посібник

Тернопіль – 2017

Коваль В.С., Струбицький П.Р. Алгоритми і структури даних. – Навчальний посібник – Тернопіль: ФОП Шпак В. Б. – 2017. – 74 с.

Завдання цього навчального посібника – дати короткий і чіткий виклад основних понять про структури даних та основні алгоритми роботи з ними. Вона призначена для студентів, які мають базові поняття мови програмування і продовжують вивчати програмування. Необхідний рівень попередніх знань може бути обмежений мінімальними знаннями з області структурного програмування.
Посібник рекомендований для студентів освітньо–професійної програми підготовки бакалавра галузі знань 12 «Інформаційні технології» спеціальностей 121 – «Інженерія програмного забезпечення», 122 «Комп'ютерні науки», 123 «Комп'ютерна інженерія», 124 «Системний аналіз», 125 «Кібербезпека», 126 – «Інформаційні системи та технології», а також галузі знань 15 «Автоматизація та приладобудування» спеціальності 151 – «Автоматизація та комп'ютерно-інтегровані технології» „Економічна кібернетика", „Комп'ютерні системи та мережі" та „Програмне забезпечення автоматизованих систем".

Укладачі: Коваль Василь Сергійович, к.т.н., доцент
Струбицький Павло Романович, к.т.н., доцент

Рецензенти: Карпінський М.П.., д.т.н., професор кафедри комп'ютерної інженерії, Тернопільський національний технічний університет імені І.Пулюя

Якименко І.З., к.т.н., доцент кафедри комп'ютерної інженерії, Тернопільський національний економічний університет

Затверджено на засіданні кафедри інформаційно-обчислювальних систем та управління, протокол №1 від 28.08.2017

© Коваль В.С.,
Струбицький П.Р., 2017
© ТНЕУ, 2017

ЗМІСТ

ВСТУП.......... 4
I. МАТЕМАТИЧНІ ОСНОВИ АНАЛІЗУ АЛГОРИТМІВ.......... 5
1.1 Історія розвитку математичної логіки.......... 5
1.2 Основи математичної логіки.......... 6
1.3 Операції над висловленнями.......... 7
1.4 Таблиці істинності.......... 12
1.5 Алгоритм побудови формули за таблицею істинності.......... 14
II. БЛОК-СХЕМИ АЛГОРИТМІВ.......... 15
2.1 Поняття алгоритму.......... 15
2.2 Ефективність алгоритмів.......... 16
2.3 Основні символи та правила побудови схем алгоритмів.......... 19
2.4 Лінійний алгоритм.......... 23
2.5 Алгоритм із розгалуженням.......... 27
2.6 Циклічні алгоритми.......... 31
III. ОСНОВИ ТЕОРІЇ ОБЧИСЛЮВАЛЬНОСТІ.......... 37
3.1 Принцип дедукції.......... 37
3.2 Числення предикатів.......... 40
3.3 Машини Тьюрінга. Обчислюваність за Тьюрінгом.......... 42
3.4 Нормальні алгоритми Маркова. Обчислюваність за Марковим.......... 44
3.5 Система Поста. Обчислюваність за Постом.......... 46
IV. АЛГОРИТМІЧНІ СТРАТЕГІЇ.......... 48
4.1 Поняття та види стратегій.......... 48
4.2 Методи розробки алгоритмів.......... 53
V. КЛАСИ СКЛАДНОСТІ P І NP.......... 58
5.1 Поняття складності алгоритмів.......... 58
5.2 Правила аналізу складності алгоритмів.......... 61
VI. ЛАБОРАТОРНІ РОБОТИ.......... 62
6.1 Лабораторна робота – «Математичні основи аналізу алгоритмів».......... 62
6.2 Лабораторна робота – «Складність алгоритмів».......... 68
ЛІТЕРАТУРА.......... 73

ВСТУП

Теорія алгоритмів – це наука, що вивчає загальні властивості та закономірності алгоритмів, різноманітні формальні моделі їх подання. На основі формалізації поняття алгоритму можливе порівняння алгоритмів за їх ефективністю, перевіркою їх еквівалентності, визначення областей застосовності.

Навчальний посібник націлений на формування знань і умінь, які утворюють теоретичний фундамент, необхідний для постановки і вирішення завдань в галузі інформатики, для коректного розуміння обмежень, що виникають при створенні обчислювальних структур, алгоритмів і програм обробки інформації.

Як відомо, після закінчення кожного десятиріччя елементна база комп'ютерів, операційні системи, засоби доступу та самі програми змінюються корінним чином. Однак структури і алгоритми, що лежать в їх основі, залишаються незмінними протягом набагато більшого часу. Ці основи стали закладатися тисячоліття тому, коли розроблена формальна логіка і розроблені перші алгоритми.

Зараз, коли можливості обчислювальної техніки багаторазово зросли, а самих персональних комп'ютерів значно більше, ніж людей, які вміють їх ефективно використовувати, розуміння того, що можна і що не можна зробити за допомогою сучасної обчислювальної техніки набуває виняткового значення.

Саме загальна теорія алгоритмів показала, що є завдання, які не розв'язні ні при якому збільшенні потужності обчислювальних засобів. Теорія складності обчислень поступово призводить до розуміння того, що бувають завдання розв'язні, але об'єктивно-складні, причому складність їх може виявитися в деякому сенсі абсолютної, тобто практично недоступною для сучасних комп’ютерів.

I. МАТЕМАТИЧНІ ОСНОВИ АНАЛІЗУ АЛГОРИТМІВ

1.1 Історія розвитку математичної логіки

Логіка як наука сформувалася в 4 в. до н.е. Її створив грецький учений Арістотель. Слово «логіка» походить від грецького "логос", що з одного боку означає "слово" або "виклад", а з іншого мислення. У тлумачному словнику Ожегова С.І. сказано: "Логіка наука про закони мислення і його формах". У 17 в. німецький учений Лейбніц задумав створити нову науку, яка була б «мистецтвом обчислення *істини»*. У цій логіці, по думки Лейбніца, кожному висловлюванню відповідав би символ, а міркування мали б вигляд обчислень. Ця ідея Лейбніца, не зустрівши розуміння сучасників, не набула поширення і розвитку і залишилася геніальною здогадкою.

Тільки в середині XIX ст. ірландський математик Джордж Буль втілив ідею Лейбніца. У 1854 році їм була написана робота "Дослідження законів мислення" (Investigation the laws of thought), яка заклала основи алгебри логіки, в якій діють закони, схожі з законами звичайної алгебри, але буквами позначаються не числа, а висловлювання. На мові булевої алгебри можна описати міркування і "вирахувати" їх результати. Проте нею охоплюються далеко не всі міркування, а лише певний тип їх, тому алгебру Буля вважають обчисленням висловлювань.

Алгебра логіки Буля з'явилася зародком нової науки - математичної логіки. На відміну від неї, логіку Аристотеля називають традиційною формальною логікою. У назві "математична логіка" відображені дві особливості цієї науки: по-перше, математична логіка - це логіка, яка використовує мову і методи математики, по-друге, математична логіка викликана до життя потребами математики.

В кінці 19 ст. створена Георгом Кантором теорія множин представлялася надійним фундаментом для всієї математики, в тому числі і математичної логіки, по крайній, мірі, для обчислення висловлювань (алгебри Буля), тому виявилося, що алгебра Кантора (теорія множин) ізоморфна алгебрі Буля.

Математична логіка сама стала областю математики, спочатку здавалася надзвичайно абстрактною і нескінченно далекою від практичних додатків. Однак ця область недовго залишалася долею "чистих" математиків. На початку 20 ст. (1910 р.) російський вчений Еренфест П.С. вказав на можливість застосування апарату булевої алгебри в телефонному зв'язку для опису перемикальних ланцюгів. У 1938-1940 р. майже одночасно з'явилися роботи радянського вченого Шестакова В. І., американського вченого Шеннона і японських учених Накасіми і Хакадзави про застосування математичної логіки в цифровій техніці. Перша монографія, присвячена використанню математичної логіки при проектуванні цифрової апаратури, була опублікована в СРСР радянським ученим Гавриловим М.А. в 1950 р. Надзвичайно важлива роль математичної логіки в розвитку сучасної мікропроцесорної техніки: вона використовується в проектуванні апаратних засобів ЕОМ, у розробці всіх мов програмування і в конструюванні дискретних пристроїв автоматики.

Великий внесок у розвиток математичної логіки зробили вчені різних країн: професор Казанського Університету Порицький П.С., де-Морган, Пірс, Тьюринг, Колмогоров А.Н., Гейдель К. та ін

1.2 Основи математичної логіки

Логічні уявлення - опис досліджуваної системи, процесу, явища у вигляді сукупності *складних висловлювань,* складених з *простих (елементарних) висловлювань* та *логічних зв'язок* між ними. Логічні уявлення та їх складові характеризуються певними властивостями і набором допустимих перетворень над ними (операцій, правил виводу і т.п.), що реалізують розроблені у формальній (математичній) *логіці правильні методи міркувань - закони логіки.*

Способи (правила) формального представлення висловлювань, побудови нових висловлювань з наявних за допомогою логічно правильних перетворень, а також способи (методи) встановлення істинності чи хибності висловлень вивчаються в *математичній логіці.* Сучасна математична логіка включає два основні розділи: *логіку висловлювань* і охоплює її *логіку предикатів* (рис. 1.1), для побудови яких існують два підходи (мови), створюючих два варіанти формальної логіки: *алгебру логіки* і *логічні обчислення.* Між основними поняттями цих мов формальної логіки має місце взаємно однозначна відповідність. Їх ізоморфізм забезпечується в остаточному підсумку єдністю законів логіки, що лежать в основі допустимих перетворень.

Математична логіка

Логіка висловлювань

Логіка предикатів

Способи побудови логіки

Алгебра логіки

Логіка обчислень

Рис. 1.1 – Математична логіка

Основними об'єктами традиційних розділів логіки є висловлювання.

Під **висловленням** розумітимемо речення, про зміст якого можна сказати: істинний він чи хибний, і притому лише одне з двох. Звичайно, це не означення. Поняття висловлення є в логіці висловлень вихідним, неозначуваним. Всі наукові знання (закони та явища фізики, хімії, біології та ін, математичні теореми і т.п.), події повсякденного життя , ситуації, що виникають в економіці і процесах управління, формулюються у вигляді висловлювань. Наказові і питальні речення не є висловлюваннями.

Саме ця властивість — бути істинним чи хибним — є характеристичною для висловлення як предмета вивчення логіки. Поняття істинності і хибності в логіці висловлень не аналізуються, а беруться як дані. Наприклад, “7 — просте число” є висловлення істинне, “ТНЕУ — не вищий навчальний заклад” — висловлення хибне.

Розглянемо вираз “*x* більше від одиниці”. Цей вираз не е висловленням, бо немає змісту твердити про його істинність чи хибність доти, поки символ “*x*” не буде замінено назвою певного дійсного числа.

Визначення 1 Вираз, який не є висловленням, але стає ним після заміни всіх символів змінних, що входять до цього виразу, назвами відповідних предметів, називають ***висловлювальною формою*** або ***невизначеним висловленням***.

Розглядаючи висловлення, виходять з двох основних *припущень*:

а) кожне висловлення є або істинним, або хибним, тобто третього не дано (***закон виключеного третього***);

б) жодне висловлення не є одночасно істинним і хибним (***закон виключення суперечності***).

Позначимо значення “істинне” та “хибне” відповідно через “1” та “0”. Звичайно, “1” і “0” тут не є назвами чисел, а лише символами значень введеної ***функції істинності***. Значення “1” і “0” називають ***значеннями істинності*** чи ***істинісними значеннями.***

Висловлювальні змінні позначають так само, як числові змінні в математиці: $p, q, r, p_1, p_2, p_3, \ldots$ Замість цих символів можна підставляти довільні висловлення. Звичайно, символи $p, q, r, p_1, p_2, p_3, \ldots$ не є висловленнями, вони є ***змінними для висловлень*** (їх також називають ***змінними висловленнями*** або ***пропозиційними буквами*** чи ***пропозиційними змінними***).

Значення функції істинності для даного значення аргументу p позначатимемо $|p|$. Так, позначивши через p висловлення “2 — найменше просте число”, а через q — висловлення: “Число π дорівнює 3,14”, матимемо: $|p|= 1$, $|q|=0$.

1.3 Операції над висловленнями

Однією з основних задач логіки висловлень є дослідження операцій, за допомогою яких з певних вихідних висловлень утворюють нові висловлення. Такі операції і називають ***логічними***.

Означення кожної логічної операції задаватимемо відповідною матрицею (таблицею), в перших стовпчиках якої записуються всі можливі істинісні значення компонентів, а в останньому стовпчику — істинісне значення результату операції.

Визначення 2 Бінарну логічну операцію, яка відповідає зв’язці “і” звичайної мови, позначається символом “$\wedge$” і задається наступною матрицею, називають ***кон'юнкцією***, або ***логічним множенням.***

p	*q*	*p∧q*
0	0	0
0	1	0
1	0	0
1	1	1

Вищенаведене табличне означення операції “кон'юнкція” рівнозначне такому словесному означенню: “Кон'юнкція $p \wedge q$ істинна тоді і тільки тоді, коли обидва компоненти p і q є одночасно істинними”.

У схемах управління логічна операція кон'юнкція реалізується в схемі збіги, показаної на рис. 1.2.

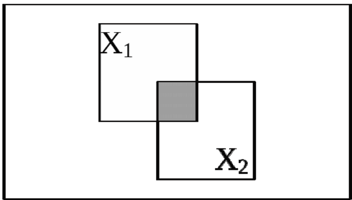


Рис. 1.2 –Діаграма Венна (кон'юнкція)

<u>Визначення 3</u> Бінарну логічну операцію, відповідну зв’язці “або нероздільне”, що позначається символом “∨” і задається наступною таблицею, називають ***диз’юнкцією*** або ***логічним додаванням***.

p	*q*	*p∨q*
0	0	0
0	1	1
1	0	1
1	1	1

Словесне означення диз'юнкції: “Диз’юнкція “$p \vee q$” істинна тоді і тільки тоді, коли принаймні один з її компонентів p або q є істинним, у протилежному випадку диз’юнкція є хибною”.

Операція диз'юнкції часто називається логічним складанням, а також логічною операцією "АБО" (рис. 1.3). Схему, відтворюючу операцію логічного додавання зазвичай називають збиральної схемою.

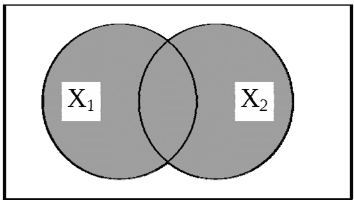


Рис. 1.3 – Діаграма Венна (диз'юнкція)

Зауважимо, що $A \vee 1 = 1$; $A \vee \bar{A} = 1$.

З двох простих висловлювань за допомогою логічних зв'язків можна утворити 16 логічних висловлювань. Але заперечення, кон'юнкція і диз'юнкція є основними логічними зв'язками, так як всі інші можна утворити з основних логічних зв'язків.

Визначення 4 Унарну операція, відповідну виразу "неправильно, що", яку позначають символом "—" і задають наступною таблицею, називають ***логічним запереченням***. Вираз „$\overline{p}$" читають "не p".

p	$\overline{p}$
0	1
1	0

Словесне означення заперечення: "Заперечення $\overline{p}$ істинне тоді і тільки тоді, коли p — хибне, у протилежному випадку $\overline{p}$ — хибне" (рис. 1.4).

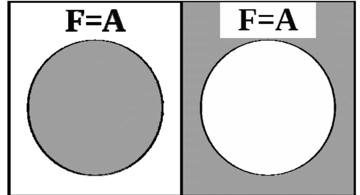


Рис. 1.4 – Діаграма Венна (заперечення)

Заперечення є найпростішою логічною операцією і єдиною логічною операцією, виконуваної над одним аргументом.

Зауважимо, що послідовне виконання двох операцій заперечення Ā призводить до вихідного значення А.

Визначення 5 Операцію, яка відповідає сполучнику "якщо..., то...", позначається символом "→", називають ***імплікацією***. Її задають таблицею:

p	q	$p \to q$
0	0	1
0	1	1
1	0	0
1	1	1

В імплікації p називають ***антецедентом*** або ***посиланням (умовою)***, q — ***консеквентом*** або ***висновком***.

Словесне означення операції "імплікація" таке: "Імплікація „$p \to q$" хибна тоді і тільки тоді, коли її антецедент — істинний, а консеквент — хибний, в усіх інших випадках імплікація — істинна".

Імплікацією висловлень А і В називається висловлювання, позначене символами А → В, яке хибно тоді і тільки тоді, коли А істинно, а В хибно. Читається "А імплікує В". Імплікація - це логічна операція, відповідна союзу "якщо ... то". Запис А → У означає те ж, що і вислів: "якщо А то В", "з А

випливає B" "A є достатня умова для B", для того щоб A необхідно, щоб B "," B є необхідна умова для A "," для того щоб B, достатньо щоб A ". Порівняємо такі пропозиції:

1. Якщо число n ділиться на 4, то воно ділиться на 2.
2. Якщо Іванов захоплений математикою, то Петров нічим не цікавиться.

Очевидно, що сенс союзу "якщо ... то" в цих пропозиціях різний.

З визначення імплікації випливає, що:

1. Імплікація з помилковим антецедентом завжди істинна.
2. Імплікація з істинним консеквент завжди істинна.
3. Імплікація хибна тоді і тільки тоді, коли її антецендент істинний, а консеквент хибний.

Прийняте визначення імплікації відповідає вживанню союзу "якщо ... то" в пропозиції "Якщо буде гарна погода, то я прийду до тебе в гості", яке ви розціните як брехня в тому випадку, коли погода гарна, а приятель до вас не прийде.

Разом з тим визначення імплікації змушує вважати істинним пропозиції як "Якщо 2x2 = 4, то Москва столиця Росії"; "Якщо 2x2 = 5, то я найкрасивіша дівчина Росії". Це пов'язано з тим, що визначеннями логічних операцій сенс складових висловлень не враховується, вони розглядаються як об'єкти, що володіють єдиною властивістю - бути істинними або помилковими (рис. 1.5).

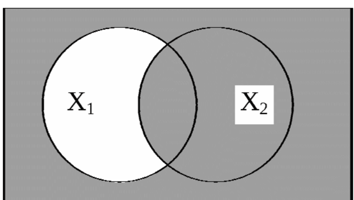


Рис. 1.5 – Діаграма Венна (імплікація)

<u>Визначення 6</u> Бінарну логічну операцію, яка відповідає зв’язці “тоді і тільки тоді”, позначається символом “$\leftrightarrow$”, називають ***еквіваленцією***. Її табличне означення:

p	q	$p\leftrightarrow q$
0	0	1
0	1	0
1	0	0
1	1	1

Словесне означення еквіваленції можна сформулювати так: “Еквіваленція “$p\leftrightarrow q$” істинна тоді і тільки тоді, коли p і q набувають однакових значень істинності, в протилежному випадку еквіваленція — хибна”.

Логічна операція еквівалентності, відповідає союзу "тоді і тільки тоді, коли" і читається "A еквівалентно B", "Для того, щоб A необхідно і достатньо, щоб B".

Коли ми говоримо "A тільки тоді, коли B" то маємо на увазі, що обидві пропозиції A і B одночасно істинні, або одночасно хибні. Наприклад, говорячи:

"Я поїду в Ленінград тоді і тільки тоді, коли ти поїдеш до Києва", ми стверджуємо, що-небудь станеться і те і інше, або ні того, ні іншого. Можна довести використовуючи таблицю істинності, що для будь-яких висловлювань А і В висловлювання $(A \Leftrightarrow B) = 1$ тоді і тільки тоді, коли $(A \Rightarrow У) = 1$ і $(B \Rightarrow A) = 1$.

Це твердження використовується при доказі теорем виду $A \Leftrightarrow B$. Одним із способів доказу істинності висловлювання $A \Leftrightarrow B$ є доказ істинності висловлювання $A \Rightarrow У$ (необхідність) і істинності висловлювання $У \Rightarrow A$ (достатність).

Висловлювання А і В називаються рівносильними. Якщо $(A \Leftrightarrow B) = 1$ кажуть, що формули F_1 і F_2 рівносильні, якщо їх еквіваленції $F1 \Leftrightarrow F2$ - тавтологія (тотожне істинне висловлювання, що на рис. 1.6).

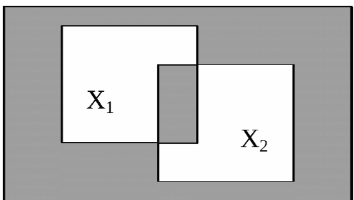


Рис. 1.6 – Діаграма Венна (еквіваленції)

Запис $F_1 \equiv F_2$ читається: "формула F_1 рівносильна формулою F_2"

$$A \Leftrightarrow У \equiv (A \Rightarrow B) \wedge (B \Rightarrow A)$$

Рівносильність є відношення між формулами (також як рівність відношення між числами, паралельність - відношення між прямими). Ставлення равносильности володіє наступними властивостями:

а) рефлективності $F \equiv F$

б) симетричності: якщо $F_1 \equiv F_2$ то $F_2 \equiv F_1$

в) транзитивності: якщо $F_1 \equiv F_2$ і $F_2 \equiv F_3$, то $F_1 \equiv F_3$

Висловлення, які не містять логічних зв'язок, називають ***елементарними*** чи ***атомарними***. Наприклад, висловлення “Рейк’явік — столиця Ісландії” і “3 — просте число” — елементарні висловлювання. Висловлення, яке містить хоча б одну логічну зв’язку, називають ***складним***. Наприклад, висловлення “10 не є простим числом”.

Пропозиційні літери, символи логічних операцій та дужки – це ***вихідні символи*** алгебри висловлень.

Довільну послідовність вихідних символів називають ***виразом мови***.

З множини виразів виділяють підмножину формул.

Формулами алгебри висловлень називають пропозиційні літери і вирази виду

$$\bar{F},\ F \wedge G,\ F \vee G,\ F \rightarrow G,\ F \leftrightarrow G,$$

де F, G – формули.

При цьому пропозиційні літери є елементарними (атомарними) формулами. Наприклад, $(((p \to (\overline{q})) \vee r) \oplus s)$ – формула.

Дужки у формулі означають порядок виконання операції.

Символу кожної логічної операції відповідає пара дужок. Щоб запобігти громіздкості формул, використовують такі правила скорочення:

1) зовнішні дужки у записі кінцевої формули можна опустити;

2) всім логічним операціям приписують відповідний ранг, який знижується зліва на право: $\overline{\ }$, $\wedge$, $\vee$, $\to$, $\leftrightarrow$.

Порядок старшинства логічних операцій наступний: $\neg$, &, $\wedge$, $\to$, $\leftrightarrow$.

Лівіша операція є сильнішою за правішу.

Знаючи ранг операції можна не використовувати дужки.

1.4 Таблиці істинності

У таблиці істинності формули $f(p,q,r)$ кожному символу логічної операції в формулі $f(p,q,r)$ відповідає окремий стовпчик таблиці, останній стовпчик відповідає істинісному значенню, яке визначається даною формулою (її головною операцією). Звернемо увагу на те, що кожен стовпчик таблиці істинності для формули $f(p,q,r)$ відповідає певному кроку процесу її побудови або, як кажуть, певній підформулі $f(p,q,r)$.

<u>Наприклад</u>, нехай задано функцію $f(p,q,r)=(\overline{p} \to q \vee r) \wedge (q \to p \vee \overline{r})$.

p	q	r	$\overline{p}$	$q \vee r$	$\overline{p} \to q \vee r$	$\overline{r}$	$p \vee \overline{r}$	$q \to p \vee \overline{r}$	$f(p,q,r)$
0	0	0	1	0	0	1	1	1	0
0	0	1	1	1	1	0	0	1	1
0	1	0	1	1	1	1	1	1	1
0	1	1	1	1	1	0	0	0	0
1	0	0	0	0	1	1	1	1	1
1	0	1	0	1	1	0	1	1	1
1	1	0	0	1	1	1	1	1	1
1	1	1	0	1	1	0	1	1	1

Часто таблицею істинності формули $f(p_1, ..., p_n)$ називають скорочену таблицю, в якій з вищенаведеної залишають перших n стовпчиків (значень аргументів $p_1, ..., p_n$) і останній стовпчик.

Степінь складності таблиці істинності для формули f швидко зростає із збільшенням кількості різних пропозиційних букв, що входять до f. Так, при n = 3 кількість рядків таблиці дорівнює $2^3 = 8$, при $n = 4$ воно становить $2^4 = 16$, при $n = 5$ дорівнює $2^5 = 32$, а при $n = 10$ — вже 1024. Практично побудувати таблицю істинності в останньому випадку вже неможливо.

<u>**Означення 7**</u> Формули алгебри висловлень $f(p_1, ..., p_n)$, які на всіх наборах $(p_1, …, p_n)$, тобто при всіх можливих розподілах істинісних значень пропозиційних літер $p_1, ..., p_n$, набувають значення 1 (останній стовбець таблиці

істинності – лише 1), називають ***тавтологіями***, ***тотожно істинними формулами*** або ***законами алгебри висловлень***.

Те, що формула алгебри висловлень f є тавтологією, позначають так: $\models f$.

Означення 8 Формулу алгебри висловлень $f\ (p_1\ ,\ \ldots,\ p_n)$, яка набуває значення істинності 0 на всіх 2^n наборах, називають ***суперечністю***. Найпростішим прикладом суперечності є формула $p\wedge \bar{p}$.

Означення 9 Формулу алгебри висловлень, яка не є ні тавтологією, ні суперечністю, називають ***нейтральною***. Прикладом нейтральної формули є $p\rightarrow q$.

Множини тавтологій, суперечностей і нейтральних формул попарно не перетинаються і разом становлять множину всіх формул алгебри висловлень.

Означення 10 Формулу алгебри висловлень, яка не є суперечністю, називають ***виконуваною***.

Так, формула $p\rightarrow p$ — виконувана і формула $p\rightarrow\bar{p}$ теж виконувана при $|p| = 0$; $|p\rightarrow \bar{p}|=1$.

Означення 11 Висловлення називають ***логічно істинним*** (на базі алгебри висловлень) тоді і тільки тоді, коли його логічна структура є тавтологією.

Прикладом логічно істинного твердження є “Трикутник *ABC* — рівнобедрений або трикутник *ABC* — не рівнобедрений” (логічна структура цього твердження — $p\vee\bar{p}$).

Означення 12 Формули алгебри висловлень $f\ (p_1\ ,\ \ldots,\ p_n)$ і $g\ (\ p_1\ ,\ \ldots,\ p_n)$ називають ***рівносильними*** або ***логічно еквівалентними***, якщо їх функції істинності $|f|$ і $|g|$ тотожно рівні, тобто, якщо їх значення на всіх 2^n наборах збігаються.

Рівносильність формул f і g позначатимемо $f\equiv g$ Символ “≡” не є символом операції алгебри висловлень, а означає певне відношення між формулами.

Основні рівносильності (закони) алгебри висловлень:

1. $p\wedge q\equiv q\wedge p$ – комутативність кон’юнкції.
2. $p\vee q\equiv q\vee p$ – комутативність диз’юнкції.
3. $p\wedge(q\wedge r)\equiv(p\wedge q)\wedge r$ – асоціативність кон’юнкції.
4. $p\vee(q\vee r)\equiv(p\vee q)\vee r$ – асоціативність диз’юнкції.
5. $p\wedge(q\vee r)\equiv p\wedge q\vee p\wedge r$ – перший дистрибутивний закон.
6. $p\vee(q\wedge r)\equiv(p\vee q)\wedge(p\vee r)$ – другий дистрибутивний закон.
7. $\overline{(p\wedge q)}\equiv \bar{p}\vee \bar{q}$ } закони
8. $\overline{(p\vee q)}\equiv\bar{p}\wedge\bar{q}$ } де Моргана.
9. $p\rightarrow q\equiv\bar{p}\vee q$.
10. $p\leftrightarrow q\equiv(p\rightarrow q)\wedge(q\rightarrow p)$.
11. $\bar{\bar{p}}\equiv p$ – закон подвійного заперечення.
12. $p\wedge p\equiv p$. } закони
13. $p\vee p\equiv p$. } ідемподентності.
14. $p\rightarrow q\equiv \bar{q}\rightarrow\bar{p}$.
15. $p\vee p\wedge q\equiv p$. } закони
16. $p\wedge(p\vee q)\equiv p$. } поглинання.

17. $u \wedge 1 \equiv u$.
18. $u \wedge 0 \equiv 0$.
19. $u \vee 1 \equiv 1$.
20. $u \vee 0 \equiv u$.

} правила співвідно-шення констант

21. $p \vee \overline{p} \equiv 1$ – закон виключення третього.
22. $p \wedge \overline{p} \equiv 0$ – закон протиріччя.

1.5 Алгоритм побудови формули за таблицею істинності

1. Виділити ті рядки таблиці істинності, в останніх стовпчиках яких міститься величина функції 1.
2. Виписати для кожного такого виділеного рядка кон'юнкцію (логічне «і») таким чином: якщо величина змінної цьому рядку дорівнює 1, то в кон'юнкцію записати позначення цієї змінної, інакше — її заперечення.
3. Всі отримані на попередньому кроці кон'юнкції записати елементами диз'юнкції (логічного «або»).

Приклад скорочення логічних виразів (спрощення логічних формул):

$$X \wedge \overline{Y} \wedge Z \vee X \wedge \overline{Y \wedge Z} \vee X \wedge Y \wedge Z \vee X \wedge \overline{Y} = X \wedge Z \wedge (\overline{Y} \vee Y) \vee X \wedge (\overline{Y \wedge Z} \vee \overline{Y}) =$$

$$X \wedge Z \vee X \wedge (\overline{Y} \vee \overline{Z} \vee \overline{Y}) = X \wedge (Z \vee \overline{Y} \vee \overline{Z}) = X \wedge (1 \vee \overline{Y}) = X \wedge 1 = X .$$

ІІ. БЛОК-СХЕМИ АЛГОРИТМІВ

2.1 Поняття алгоритму

Поняття алгоритму інтуїтивно зрозуміло та часто використовується в математиці та комп'ютерних науках. Говорячи неформально, **алгоритм** - це довільна коректно визначена обчислювальна процедура, на **вхід** якої подається деяка величина або набір величин, а результатом виконання якої є **вихідна** величина або набір значень. Таким чином, алгоритм є послідовністю обчислювальних кроків, які перетворюють вхідні величини у вихідні.

Алгоритм можна також розглядати як інструмент, який призначений для вирішення коректно поставленої обчислювальної задачі. У постановці задачі в загальних рисах визначаються відношення між входом та виходом. В алгоритмі описується конкретна обчислювальна процедура, за допомого якої можна досягнути виконання вказаних відношень.

Можна навести загальні риси алгоритму:

a. *Дискретність інформації.* Кожний алгоритм працює із даними: вхідними, проміжними, вихідними. Ці дані представляються у вигляді скінченних слів деякого алфавіту.
b. *Дискретність роботи алгоритму.* Алгоритм виконується по кроках та при цьому на кожному кроці виконується тільки одна операція.
c. *Детермінованість алгоритму.* Система величин, які отримуються в кожний (не початковий) момент часу, однозначно визначається системою величини, які були отримані в попередні моменти часу.
d. *Елементарність кроків алгоритму.* Закон отримання наступної системи величин з попередньої повинен бути простим та локальним.
e. *Виконуваність операцій.* В алгоритмі не має бути не виконуваних операцій. Наприклад, неможна в програмі призначити значення змінній «нескінченність», така операція була би не виконуваною. Кожна операція опрацьовує певну ділянку у слові, яке обробляється.
f. *Скінченність алгоритму.* Опис алгоритму повинен бути скінченним.
g. *Спрямованість алгоритму.* Якщо спосіб отримання наступної величини з деякої заданої величини не дає результату, то має бути вказано, що треба вважати результатом алгоритму.
h. *Масовість алгоритму.* Початкова система величин може обиратись з деякої потенційно нескінченної множини.

Розглянемо для прикладу задачу сортування послідовності чисел у зростаючому порядку. Ця задача часто виникає на практиці і, фактично, буде центральною проблемою першого розділу даного курсу. **Задача сортування** визначається формально наступним чином.

Вхід: послідовність n чисел $(a_1, a_2,..., a_n\}$

Вихід: перестановка $(a', a2, ..., a'_n)$ вхідної послідовності таким чином, що для всіх її членів виконується співвідношення $a' < a2 < ... < a'_n$.

Наприклад, якщо на вхід подається послідовність <31, 41, 59, 26, 11, 58), то вивід алгоритму сортування повинен бути таким: <26, 31, 41, 41, 58, 59).

Подібна вихідна послідовність називається **екземпляром задачі** сортування. Взагалі, екземпляр задачі складається із входу, який необхідний для розв' язання задачі та який задовольняє усім обмеженням, які присутні в постановці задачі.

В комп'ютерних науках сортування є основною операцією (у багатьох програмах вона використовується в якості проміжного кроку), в результаті чого з'явилось багато якісних алгоритмів сортування. Вибір найбільш адекватного алгоритму залежить від багатьох факторів, в тому числі й від кількості елементів для сортування, від їх порядку у вхідній послідовності, від можливих обмежень, які накладаються на членів послідовності.

Кажуть, що алгоритм є **коректним,** якщо для кожного входу результатом його роботи є коректний вивід. Тоді коректний алгоритм **розв'язує** дану обчислювальну задачу. Якщо алгоритм некоректний, то для деяких входів він може взагалі не завершити свою роботу або видати відповідь, яка відрізняється від очікуваної.

Для чого вивчати алгоритми?

По-перше, алгоритми є життєво необхідними складовими для рішення будь-яких задач з різноманітних напрямків комп'ютерних наук. Алгоритми відіграють ключову роль у сучасному розвитку технологій. Тут можна згадати такі розповсюджені задачі, як:

- розв'язання математичних рівнянь різної складності, знаходження добутку матриць, обернених матриць;
- знаходження оптимальних шляхів транспортування товарів та людей;
- знаходження оптимальних варіантів розподілення ресурсів між різними вузлами (виробниками, верстатами, працівниками, процесорами тощо);
- знаходження в геномі послідовностей, які співпадають;
- пошук інформації в глобальній мережі Інтернет;
- прийняття фінансових рішень в електронній комерції;
- обробка та аналіз аудіо та відео інформації.

Цей список можна продовжувати й продовжувати і, власно кажучи, майже неможливо знайти таку галузь комп'ютерних наук та інформатики, де б не використовувались ті або інші алгоритми.

По-друге, якісні та ефективні алгоритми можуть бути каталізаторами проривів у галузях, які є на перший погляд далекими від комп'ютерних наук (квантова механіка, економіка та фінанси, теорія еволюції).

І, по-третє, вивчення алгоритмів це також неймовірно цікавий процес, який розвиває наші математичні здібності та логічне мислення.

2.2 Ефективність алгоритмів

Припустимо, швидкодія комп'ютера та об'єм його пам'яті можна збільшувати до нескінченності. Чи була би тоді необхідність у вивченні алгоритмів? Так, але тільки для того, щоб продемонструвати, що метод розв'язку має скінченний час роботи і що він дає правильну відповідь. Якщо б комп'ютери були необмежено швидкими, підійшов би довільний коректний

метод рішення задачі. Звісно, тоді найчастіше обирався би метод, який найлегше реалізувати.

Сьогодні є дуже потужні комп'ютери, але їх швидкодія не є нескінченно великою, як і пам'ять. Таким чином, час обчислення - це обмежений ресурс, як і об'єм необхідної пам'яті. Цими ресурсами слід користуватись розумно, чому й сприяє застосування алгоритмів, які ефективні в плані використання ресурсів часу та пам'яті.

Алгоритми, які розроблені для розв'язання однієї та тієї самої задачі, часто можуть дуже сильно відрізнятись за ефективністю. Ці відмінності можуть бути набагато більше помітними, чим ті, які викликані застосуванням різного апаратного та програмного забезпечення.

Як зазначалось вище, в цьому розділі центральну роль буде присвячено задачі сортування. Перший алгоритм, який буде розглядатись - сортування включенням, для своєї роботи вимагає часу, кількість якого оцінюється як c_1n^2, де n - розмір вхідних даних (кількість елементів у послідовності для сортування), сі - деяка стала. Цей вираз вказує на те, як залежить час роботи алгоритму від об'єму вхідних даних. У випадку сортування включенням ця залежність є квадратичною. Другий алгоритм - сортування злиттям - потребує часу, кількість якого оцінюється як $C2n*log2n$. Зазвичай константа сортування включенням менше константи сортування злиттям, тобто $c_i<c_2$, але як ми пересвідчимось у наступних темах, ці константи не відіграють ролі у порівнянні швидкодії різних алгоритмів. Адже зрозуміло, що функція n^2 зростає швидше зі збільшенням n, аніж функція $n\log_2 n$. І для деякого значення $n = n_0$ буде досягнуто такий момент, коли вплив різниці констант перестане мати значення і надалі функція $c_2 n\log_2 n$ буде менша за c_1n^2 для будь-яких $n > n_0$.

Для демонстрації цього розглянемо два комп'ютери - А та Б. Комп'ютер А більш швидкий і на ньому працює алгоритм сортування, а комп'ютер Б більш повільний і на ньому працює алгоритм сортування методом злиття. Обидва комп'ютери повинні виконати сортування множини, яка складається з мільйона чисел. Припустимо, що комп'ютер А виконує мільярд операцій в секунду, а комп'ютер Б - лише десять мільйонів, тобто А працює в 100 разів швидше за Б. Щоб різниця стала більш відчутною, припустимо що код методу включення написаний найкращим програмістом в світі із використанням команд процесору, і для сортування n чисел за цим алгоритмом потрібно виконати $2n^2$ операцій (тобто $c_1=2$). Сортування методом злиття на комп'ютері Б написано програмістом початківцем із використанням мови високого рівня і отриманий код потребує $50n\log_2 n$ операцій (тобто $c_2=50$). Таким чином для сортування мільйона чисел комп'ютеру А буде потрібно

$$\frac{2\cdot\left(10^6\right)^2 \text{ команд}}{10^9 \text{ команд / с}} = 2000 \text{ с},$$

а комп'ютеру Б –

$$\frac{50 \cdot 10^6 \cdot \log_2 10^6 \text{ команд}}{10^7 \text{ команд / с}} \approx 100 \text{ с}.$$

Тож, використання коду, час роботи якого зростає повільніше, навіть при поганому комп'ютері та поганому компіляторі потребує на порядок менше процесорного часу! Для сортування 10 мільйонів чисел перевага сортування злиттям стає ще більш відчутною: якщо сортування включенням потребує для такої задачі приблизно 2,3 дня, то для сортування злиттям - менше 20 хвилин. Загальне правило таке: чим більша кількість елементів для сортування, тим помітніше перевага сортування злиттям.

Наведений вище приклад демонструє, що алгоритми, як і програмне забезпечення комп'ютеру, являють собою **технологію.** Загальна продуктивність системи настільки ж залежить від ефективності алгоритму, як і від потужності апаратних засобів.

Золоте правило розробників алгоритмів

Тепер розглянемо для прикладу просту задачу, яка відома усім ще з початкової школи, а також метод розв'язання цієї задачі - множення двох цілих чисел. Цю задачу можна описати наступним чином:

Вхід: 2 цілих n-розрядних числа x та y

Вихід: добуток чисел xy

Розглянемо приклад для чисел $x = 5678$ та $y = 1234$. Результат відомого з дитинства методу множення в стовпчик буде виглядати наступним чином (рис. 2.1):

```
    5678
   ×1234
--------
   22712
  17034
 11356
 5678
--------
 7006652
```

Рис. 2.1 – Процедура множення

Легко помітити, що елементарні операції, які тут використовуються, це - множення та додавання однорозрядних чисел. Припустивши, що операція множення займає більше часу аніж операція додавання для однієї пари чисел, оцінимо кількість таких елементарних операцій. Всі вони виконується в області, яка вище позначена сірим кольором. В даному прикладі кількість елементарних операцій добутку становитиме $16 = 4^2$, а в загальному випадку становитиме n^2. Тож, кількість операцій для добутку двох цілих n-розрядних чисел методом множення у стовпчик оцінюється як cn^2, де c - деяка стала.

Проте чи можемо ми покращити цей результат, отримавши метод добутку чисел, який буде працювати швидше? Щоб мотивувати себе для пошуку такого методу, наведемо цитату з книги «Розробка та аналіз комп'ютерних алгоритмів» (Аго, Гопкрофт, Ульман, 1974): «Можливо

найбільш важливим принципом для гарного розробника алгоритмів є відмова від того, щоб бути задоволеним результатом». Слідуючи цьому правилу, розглянемо ще раз детальніше природу об'єктів задачі добутку чисел.

За умовою на вхід подається два n-розрядних числа. Припустимо ми розіб'ємо кожне число навпіл, отримавши, так звані, верхнє та нижнє півслова. Тобто, можна записати $x = 10^{n\ 2}\ a + b$ та $y = 10^{n\ 2}\ c + d$, де а, *b, c, d* - цілі n/2-розрядні числа. Тоді добуток *xy* можна представити так:

$$xy = (10^{n/2} a + b) \cdot (10^{n/2} c + d) = 10^{n} ac + 10^{n/2} (ad + bc) + bd. \qquad (2.1)$$

Таким чином ми природно підійшли до рекурсивного методу обчислення добутку двох цілих чисел, який зводить обрахунок добутку двох n-розрядних чисел до обрахунку чотирьох добутків n/2-розрядних чисел. Спробуємо з'ясувати, чи покращиться таким чином швидкість добутку двох чисел. Кожне з чисел *a, b, c, d* мають $n/2$ розрядів, а відтак добуток будь-якої їх пари (якщо використовувати для нього старий алгоритм множення у стовпчик) займатиме $c(n/2)^2$ операцій, тобто $cn^2/4$. Чотири таких добутки в сумі знову дадуть початковий результат: $4\text{-}cn\ /4 = cn$. Отже, виграш за часом не було отримано.

Невже не можливо покращити результат роботи методу множення чисел у стовпчик? Насправді, можливо і відповіддю на це питання є метод множення Карацуби (1960). Якщо подивитись на формулу (2.1), то помітимо, що насправді важливими є не чотири до, а три: *ac, bd* та *(ad* + bc), тобто елементи *ad* та *bc* нас не цікавлять самі по собі, а лише їх сума. Чи можна отримати їх суму перемноживши лише два числа? Так:

$$(a + b)(c + d) = ac + ad + bc + bd = (ad + bc) + sic + bd$$
$$aid + bc = (a + b)(c + d) - ac - bd \qquad (2.2)$$

Таким чином, суму *(s + bc)* можна отримати з добутку двох n/2-розрядних числа (можливо, n/2 + 1) (a+b) та (c+d), а також добутків *ac* та *bd,* які ми вже маємо. І, отже, кількість рекурсивних викликів скоротились з чотирьох до трьох. Аналіз швидкості методу множення приводить до оцінки $3n^{10823}$ » $3n^{11585}$.

2.3 Основні символи та правила побудови схем алгоритмів

Написанню програми на будь-якій мові програмування передує складання алгоритму, який показує послідовність виконання операцій, розгалуженість обчислювального процесу, логіку виконання всього ланцюгу програми від блоку введення вхідних даних до отримання кінцевого результату.

Для використанням блок–схем алгоритмів застосовують основні правила їх оформлення, що чітко регламентуються ГОСТ 19.701-90 (ИСО 5807-85). У таблиці 1 показано форму і наведено зміст найбільш часто використовуваних блоків. Практично всі блоки, приведені в таблиці, будуються на основі «базового» прямокутника розмірами «ab» Перший приведений в таблиці блок під назвою «процес» є «базовим» прямокутником. Розмір а = 10, 15, 20... мм,

тобто кратний п’яти. Розмір b = 1,5a, допускається b = 2a. Розмір «a» вибирається залежно від масштабу блок–схеми і повинен бути однаковим для всіх її блоків.

Таблиця.1 Форма і зміст блоків блок – схем алгоритмів

Найменування	Позначення	Функції
Процес	a b	Виконання операцій присвоєння, наприклад, A = 0, складання з присвоєнням, наприклад C = A+B віднімання, множення і т.д.
Рішення	a b	Вибір напряму виконання алгоритму (програми) залежно від деяких змінних умов
Ввід – вивід	0,25a b 0,2:	Ввід – вивід інформації без незалежно від типу пристрою вводу або виводу
Дисплей	0,5b a b	Введення інформації з дисплея (з клавіатури), виведення інформації на дисплей
Документ	a b	Виведення інформації на папір (на принтер)

Старт – зупинка	0,5a b	Початок – кінець алгоритму (програми)
З’єднувач	5 ϕ0,5a	Перехід на блок номер 5 (номер блоку наведено для прикладу)
Коментар	a	–

Кожну блок-схему починають блоком «Старт-зупинка», усередині якого пишуть слово «Початок», і закінчують блоком «Зупинка», усередині якого пишуть слово «Кінець». Блоки з’єднують лініями. Якщо лінія, що сполучає блоки «надходить» до блоку за напрямом «зверху-вниз» або «зліва – направо», то стрілку на її кінці не ставлять. Якщо ж по напряму «знизу – вгору» або «справа – наліво», то стрілку на її кінці ставлять обов’язково.

Застосування блоку «З’єднувач» дає можливість значно спростити блок–схему. Його застосовують у тому випадку, коли лінію, що сполучає блоки, потрібно вести на значну відстань і часто з перетином інших сполучних ліній. Натомість достатньо після блоку, з якого повинна виходити сполучна лінія, зобразити блок «З’єднувач» і усередині нього написати номер блоку, в який ця лінія повинна прийти.

Блок «Коментар» застосовують у тому випадку, коли усередині блоку не вдається розмістити (написати) всю необхідну інформацію. У цьому випадку до лінії, що сполучає блоки, перед блоком, для якого необхідно написати додаткову інформацію, за допомогою пунктирної лінії приєднують (справа або зліва) блок «Коментар». Інформацію можна розміщувати за висотою – в межах висоти даного блоку, або за шириною – до краю сторінки.

Окремі блоки у алгоритмі можуть нумеруватись всі або деякі, це визначається складністю та розгалуженістю процесу. Нумерація блоків завжди полегшує читання блок-схеми. Блоки між собою з’єднуються стрілками, які показують напрямок обчислювального процесу.

Проектування схем алгоритмів як правило виконують із дотриманням наступної послідовності:

1) постановка задачі;
2) математична формалізація задачі;
3) вибір методу розв’язування задачі;
4) побудова схеми алгоритму;
5) перевірка алгоритму.

Постановка задачі – це чітке формулювання задачі, визначення вхідних даних для її розв’язування і точні вказівки відносно того, які результати і в якому вигляді повинні бути отримані.

Перед виконанням наступних кроків роботи необхідно:

- уважно усвідомити задачу до чіткого розуміння її суті і вимог;

- визначити, які дані є вхідними, тобто такими, які задаються користувачем алгоритму;
- визначити, які дані є вихідними, тобто такими, які треба отримати в результаті розв'язання задачі.

Математична формалізація задачі – це опис задачі у вигляді формул, рівнянь, співвідношень, обмежень. Цей крок є найважливішим при виконанні даної роботи. Більша частина задач потребують математичної формалізації.

Математична формалізація вимагає певного рівня знань, вмінь та навичок в області, до якої належить поставлена задача.

Вибір методу розв'язування задачі полягає у виборі сукупності способів та підходів до розв'язання задачі. Вибір методу залежить як від самої задачі, так і від можливостей комп'ютера. При оцінюванні якості розв'язку задачі враховуються наступні показники:

- оригінальність розв'язку;
- об'єм пам'яті, який займає і використовує алгоритм (програма);
- трудомісткість обчислень, тобто ефективність алгоритму;
- лаконічність і наочність алгоритму.

Побудова схеми алгоритму – це графічний запис алгоритму на основі вибраного методу.

Перед формуванням кінцевого варіанту схеми бажано розглянути і проаналізувати декілька її варіантів. Алгоритм більшою мірою визначається методом, хоча один і той же метод може бути реалізований за допомогою різних алгоритмів.

Перевірка алгоритму полягає в ручній перевірці окремих розв'язків задачі. Отриманий алгоритм необхідно обов'язково перевірити за допомогою тестів. Тест – сукупність вхідних даних для алгоритму з очікуваними результатами. Зазвичай треба підготувати не один, а декілька тестів, що допоможе охопити максимум ситуацій.

Набір тестів називається повним, якщо він дозволяє активізувати всі гілки алгоритму. В наборі тестів виділяють три групи:

- «тепличні» – перевіряють роботу алгоритму при коректних, нормальних вихідних даних найпростішого вигляду;
- «екстремальні» – на межі області визначення, в ситуаціях, які можуть відбутись і на які треба коректно реагувати;

«позамежні» – за межами області визначення – ситуації, безглузді з точки зору постановки задачі, але які можуть відбутись через помилки користувача або інших алгоритмів, які надають вхідні дані для алгоритму, що тестується

В залежності від структури алгоритми бувають:

- лінійні;
- із розгалуженням;
- циклічні.

2.4 Лінійний алгоритм

Алгоритм називається ***лінійним,*** якщо всі його дії виконуються послідовно, одна за одною, від початку до кінця. У лінійному алгоритмі не може бути команди, яка б передбачала різну послідовність виконання алгоритму.

Формально при побудові схем лінійних алгоритмів використовуються три символи (рис. 2.2):

- термінатор;
- ввід/вивід;
- процес.

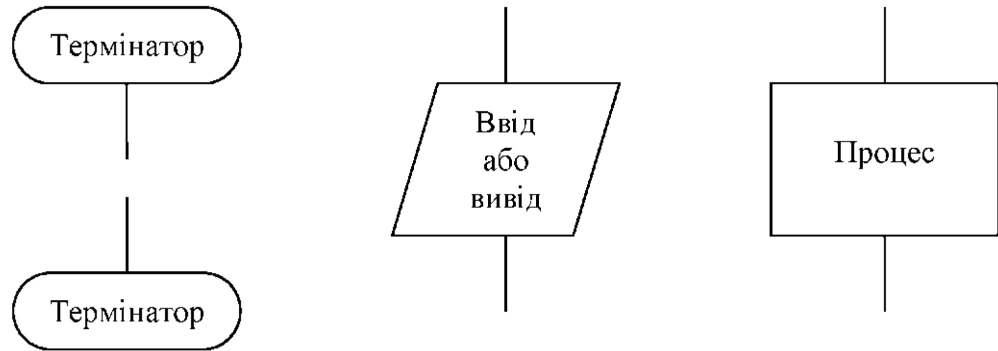


Рис. 2.2 – Символи, які використовуються при побудові схем лінійних алгоритмів

Символ «Термінатор» починає і завершує будь-яку схему алгоритму. В ньому записується відповідне слово: «Початок» або «Кінець». Від блоку «Початок» відходить лише одна лінія.

Всередині символу «Ввід/вивід» записуються значення, які вводяться в алгоритм (програму) або виводяться з нього. При цьому вказується відповідне слово: «Ввід» або «Вивід».

Всередині символу «Процес» записуються текстові вказівки, формули або оператори мови програмування.

З символів «Ввід/вивід» та «Процес» може виходити лише одна лінія.

За допомогою ліній, якими з’єднуються символи, позначається послідовність виконання кроків алгоритму. Після виконання операцій, розташованих в одному символі переходять по лінії до виконання операцій іншого символу.

При з’єднанні символів застосовують наступне правило, яке визначає, чи потрібно ставити стрілку на кінці з’єднувальної лінії. Якщо лінія задає так званий *основний напрямок* виконання операцій (зверху вниз або зліва направо), то стрілки на кінцях ліній можна не ставити. В протилежному випадку стрілки є обов’язковими (рис 2.3).

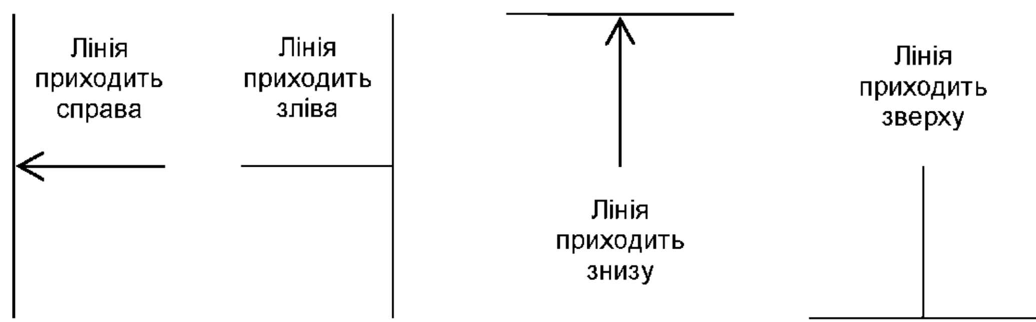


Рис. 2.3 – Ілюстрація правила виставлення стрілок на кінцях з'єднувальних ліній в схемі алгоритму

Лінії, якими з'єднуються символи, не повинні перетинатись. В разі складності або неможливості (наприклад, при розміщенні схеми алгоритму на декількох сторінках) забезпечення нерозривності з'єднувальних ліній слід використовувати з'єднувальний символ, як показано на рис. 2.4.

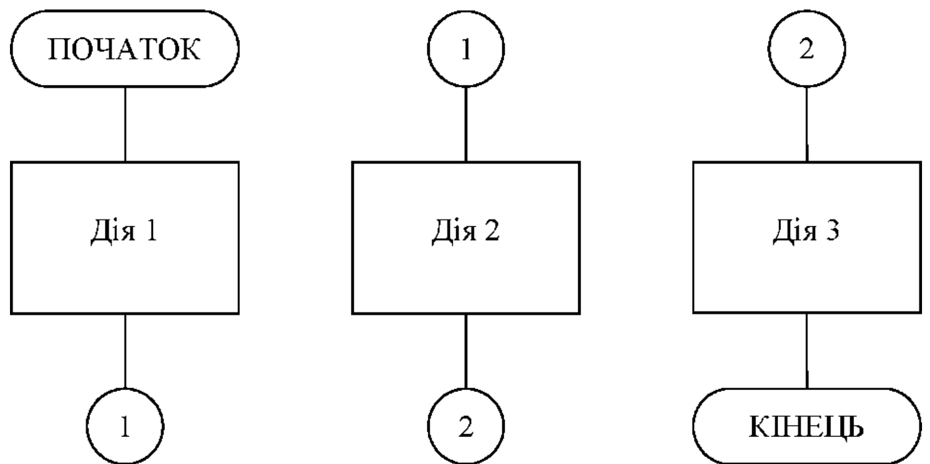


Рис 2.4 – Приклад використання з'єднувального символу

В кожному з'єднувальному символі проставляється числовий ідентифікатор. Такими, що з'єднані між собою, вважаються ті з'єднувальні символи, які містять однакові ідентифікатори.

Приклади лінійних алгоритмів

<u>Постановка задачі 1:</u> уряд гарантує, що інфляція в поточному році складатиме р % на місяць. Якого зростання цін за рік можна очікувати?

Розв'язання:

Вхідними даними в цій задачі є рівень інфляції, що задається у процентах, та інтервал часу, протягом якого треба обчислити зростання цін.

Вихідні дані – коефіцієнт зростання цін – можна обчислити як відношення ціни будь-якого товару в кінці року до ціни цього товару на початку року.

Проведемо математичну формалізацію задачі. Позначимо ціну деякого товару в даний час c_1, а ціну того ж товару в кінці року – c_{12}. Тоді ціна товару в кінці року обчислюється наступним чином:

$$c_{12} = c_1 k, \quad (2.3)$$

де к – коефіцієнт зростання ціни, який в свою чергу визначається як

$$k = \frac{c_{12}}{c_1}. \tag{2.4}$$

Якщо за 1 місяць ціна збільшується на p %, це означає, що до початкової ціни одається p її сотих частин, тобто:

$$c_2 = c_1 + c_1\frac{p}{100} = c_1\left(1 + \frac{p}{100}\right), \tag{2.5}$$

$$c_3 = c_2 + c_2\frac{p}{100} = c_2\left(1 + \frac{p}{100}\right), \tag{2.6}$$

$$\ldots$$

$$c_{12} = c_{11} + c_{11}\frac{p}{100} = c_{11}\left(1 + \frac{p}{100}\right). \tag{2.7}$$

Звідси легко побачити:

$$c_3 = c_2\left(1 + \frac{p}{100}\right) = c_1\left(1 + \frac{p}{100}\right)^2, \tag{2.8}$$

$$c_4 = c_3\left(1 + \frac{p}{100}\right) = c_1\left(1 + \frac{p}{100}\right)^3, \tag{2.9}$$

$$\ldots$$

$$c_{12} = c_{11}\left(1 + \frac{p}{100}\right) = \ldots = c_1\left(1 + \frac{p}{100}\right)^{12}. \tag{2.10}$$

Таким чином, коефіцієнт збільшення ціни за рік становитиме:

$$c_3 = \frac{c_{12}}{c_1}\frac{c_1\left(1 + \frac{p}{100}\right)}{c_1} = \left(1 + \frac{p}{100}\right)^2 \tag{2.11}$$

Алгоритм розв’язання спочатку запишемо у вигляді послідовності інструкцій:

- ввести p – рівень інфляції на місяць у процентах;
- обчислити $k = \left(1 + \frac{p}{100}\right)^{12}$ - коефіцієнт зростання цін за рік;
- вивести k.

Схема алгоритму представлена на рис. 2.5.

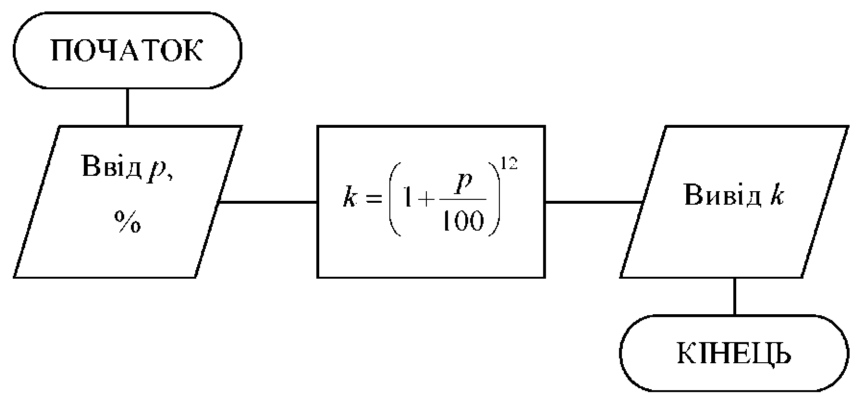


Рис. 2.5 – Схема алгоритму розв'язання задачі про інфляційне зростання цін

Постановка задачі 2: в електричному ланцюзі, зображеному на рис. 2.6, визначити силу струму в опорі R3

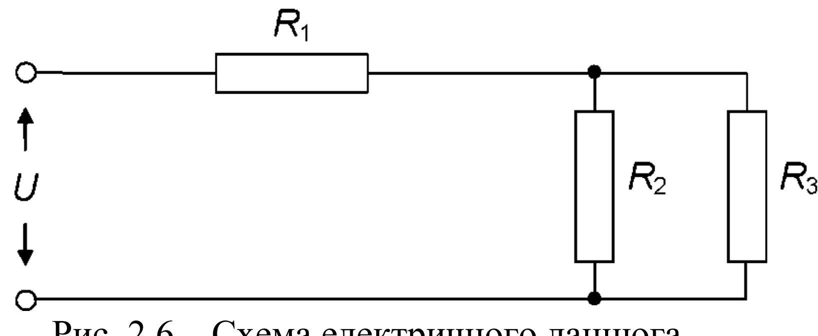


Рис. 2.6 – Схема електричного ланцюга

Розв'язання:

Вхідними даними в задачі є величина напруги U, до якої підключено ланцюг та опори R_1, R_2, R_3.

Вихідними даними є сила струму, який протікає через опір $R_3 - I_3$.

Проведемо математичну формалізацію задачі. Згадаємо, що за законом Ома сила струму, який протікає через ділянку електричного кола, прямо пропорційна величині напруги на цій ділянці та обернено пропорційна опору ділянки:

Для визначення сили струму, який протікає в опорі R_3 необхідно знайти напругу на ньому. Ця напруга обчислюється як різниця між напругою на всьому ланцюзі та падінням напруги на опорі R1:

$$U_3 = U - I_1 R_1 .$$

Струм Ii, який протікає в опорі Ri і викликає падіння напруги на ньому, визначається як відношення напруги U до сумарного опору ланцюга R, що дорівнює

$$R = R_1 + \frac{R_2 R_3}{R_2 + R_3} . \tag{2.12}$$

Таким чином, отримуємо наступну послідовність операцій в алгоритмі:

- ввід U, R_1, R_2, R_3;
- обчислення сумарного опору ланцюга - R;
- обчислення сили струму, який протікає в опорі R_1 - I_1;
- обчислення напруги на опорі R_3 - U_3;
- обчислення сили струму, який протікає в опорі R_3 - I_3;
- вивід I_3.

Схема алгоритму представлена на рис 2.7.

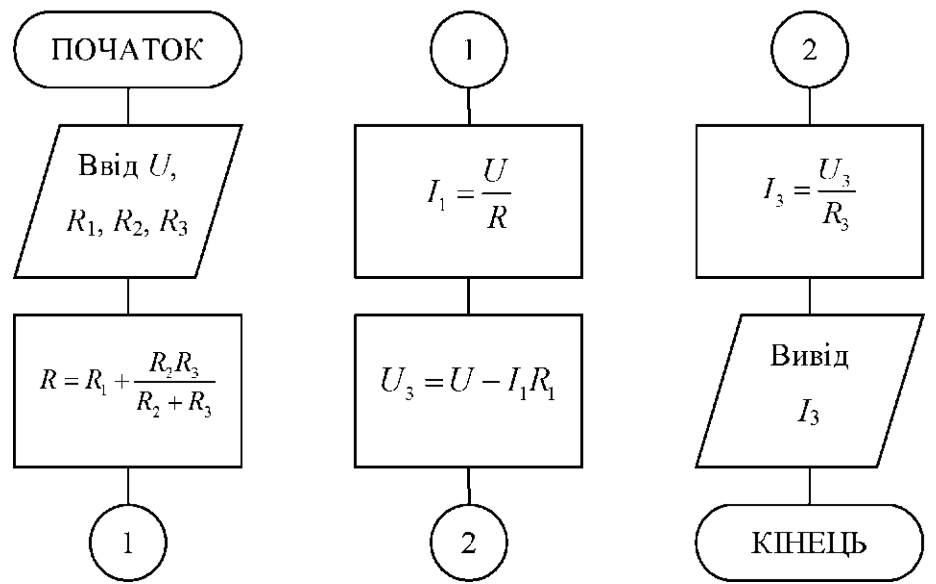


Рис. 2.7 - Схема алгоритму розв'язання задачі про електричний ланцюг

2.5 Алгоритм із розгалуженням

Розгалужений алгоритм - це алгоритм, в якому виконуються ті або інші дії залежно від результату перевірки *умови.*

При виконанні розгалуженого алгоритму від його початку до кінця можна пройти різними шляхами залежно від вихідних даних.

Розгалужений алгоритм визначається наявністю або структури *розгалуження,* або структури *вибору.*

При побудові схем розгалужених алгоритмів використовується символ "Рішення" у різних модифікаціях (рис. 2.8).

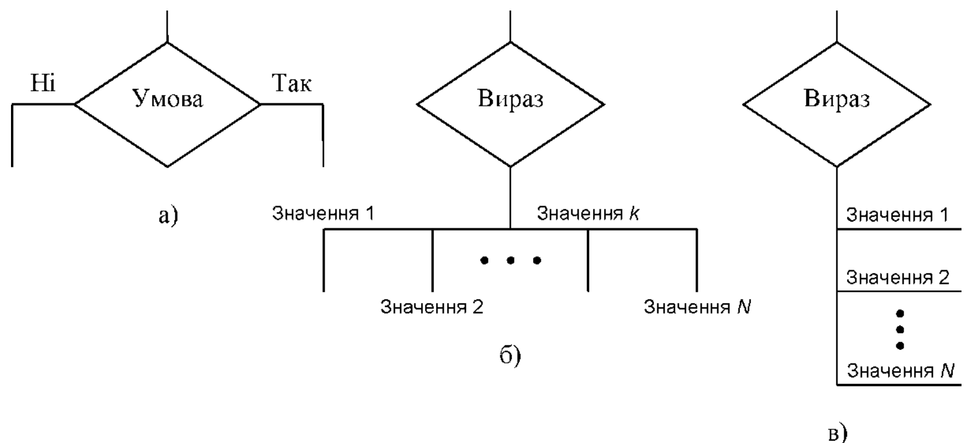
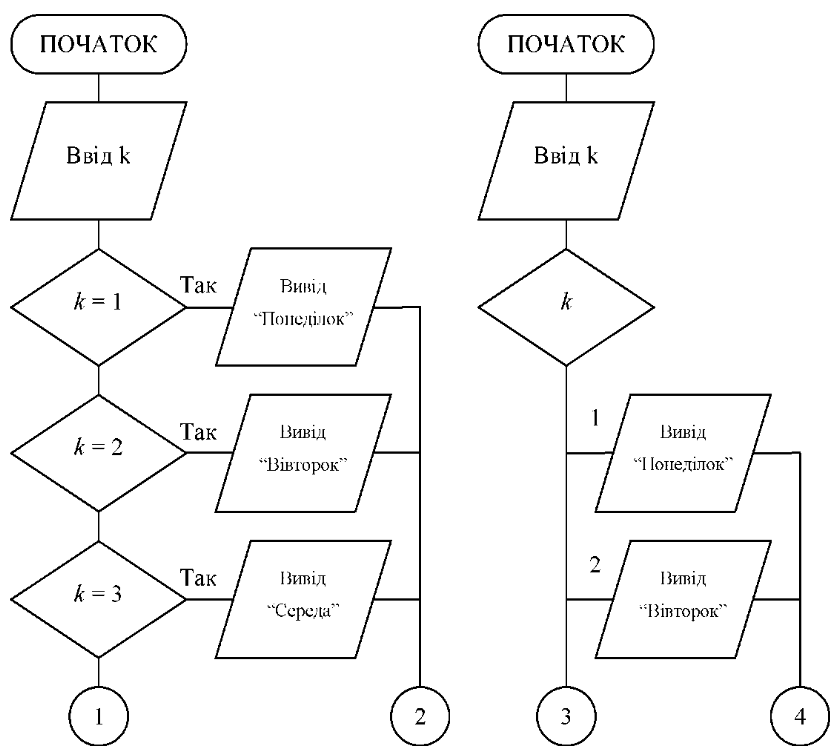


Рис. 2.8 - Символ "Рішення" та його модифікації:
а - розгалуження; б, в – вибір

Якщо символ "Рішення" в схемі алгоритму визначає структуру розгалуження, то в ньому повинен розміщуватись *логічний вираз,* який може набувати лише двох значень: *істина* і *неправда.* Значення логічного виразу при конкретних вхідних даних визначає напрямок подальшого виконання алгоритму (гілка "Так" або гілка "Ні").

Якщо символ "Рішення" в схемі алгоритму визначає структуру вибору, то в ньому повинен розміщуватись вираз, який може набувати значення з деякої кінцевої дискретної множини. Залежно від значення виразу при конкретних вхідних даних алгоритм виконуватиметься по відповідній гілці.

Структура вибору використовується в таких алгоритмах, в яких при різних значеннях одного й того ж виразу необхідно виконувати різні операції. В цьому випадку схема алгоритму побудована на основі структури вибору (рис. 2.9б) буде менш громіздкою та більш наочною, ніж схема алгоритму, побудована на основі структури розгалуження (рис. 2.9а).



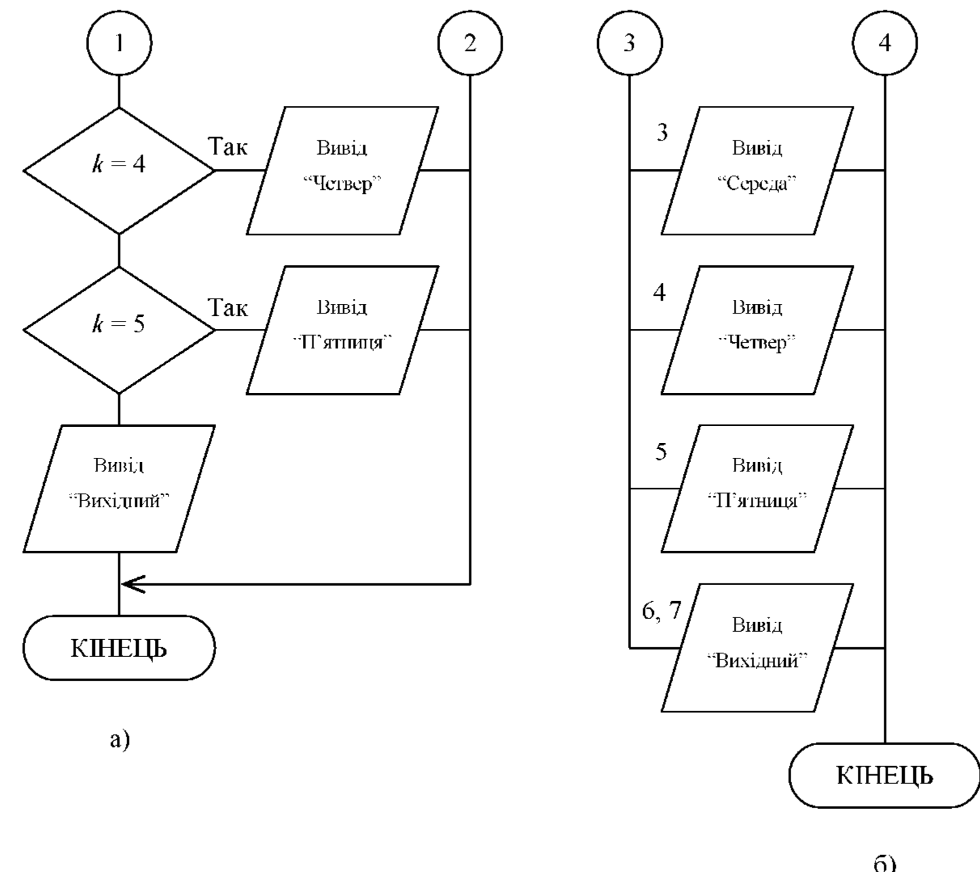


Рис. 2.9 - Реалізація схеми алгоритму:
а) - на основі структури розгалуження; б) - на основі структури вибору

Приклад алгоритму із розгалудженням

Постановка задачі: знайти найменше з трьох заданих чисел a, b і c.

Вхідними даними є числа *a, b* і *c,* вихідними - найменше число з них.

Розв'язання:

На цій нескладній задачі проілюструємо відзнаку між механізмами розв'язання, використовуваними людиною та ЕОМ. Уявимо ситуацію, коли всі три числа записані на аркуші паперу і показані людині. Мозок людини є паралельною обчислювальною системою з величезною кількістю "процесорів", тому для нього не буде складно миттєво зафіксувати у зоровій пам'яті всі три числа, оцінити їх співвідношення і через долю секунди визначити найменше або найбільше число. Із значним спрощенням можна сказати, що алгоритм, за яким працює мозок при розв'язанні даної задачі, полягає в перевірці умови

$$((a < b) \textbf{ і } (a < c)) \textbf{ або } ((b < a) \textbf{ і } (b < c)) \textbf{ або } ((c < a) \textbf{ і } (c < b)), \qquad (2.13)$$

причому всі складові цієї умови обробляються немов би водночас. Зрозуміло, що на відміну від людини, ЕОМ в кожен момент часу може оперувати із суворо обмеженою кількістю даних, тому подібний алгоритм для ЕОМ є непридатним. В той же час, якщо кількість чисел, серед яких треба знайти найменше, збільшити до ста або, навіть, тисячі, то виконавець-людина втратить спроможність «охопити» їх разом і також буде вимушений виконувати обробку послідовно, застосовуючи алгоритм «машинного» виду.

Наведений приклад показує, що алгоритм, розрахований на виконання послідовною ЕОМ, повинен орієнтуватись на послідовну обробку даних, здійснення руху до розв'язку крок за кроком.

Алгоритм розв'язання задачі в текстовому вигляді побудуємо наступним чином:

- ввести *a, b,* c;
- якщо $a <$ b:
  - якщо $a < c$ вивести а; о інакше вивести с;
- інакше:
  - якщо $b < c$ вивести *b;* о інакше вивести *c.*
- завершення алгоритму.

Після закінчення «першого раунду» – перевірки умови $a < b$ – стає відомо, яке з цих двох чисел «претендує» називатись найменшим. Якщо таким є число *a,* то залишається порівняти його з числом *c* для здійснення кінцевого висновку. Аналогічне порівняння необхідно виконати, якщо меншим виявилось число *b.* Випадок, коли будь-яка пара чисел співпадає в даному алгоритмі не розглядається.

Зауважимо, що попарне порівняння чисел можна здійснювати у довільному порядку, наприклад, спочатку порівняти *b* і *c,* а потім, за результатами цього порівняння – *b* і *a* або *c* і *a.*

Схема алгоритму представлена на наступному рис. 2.10.

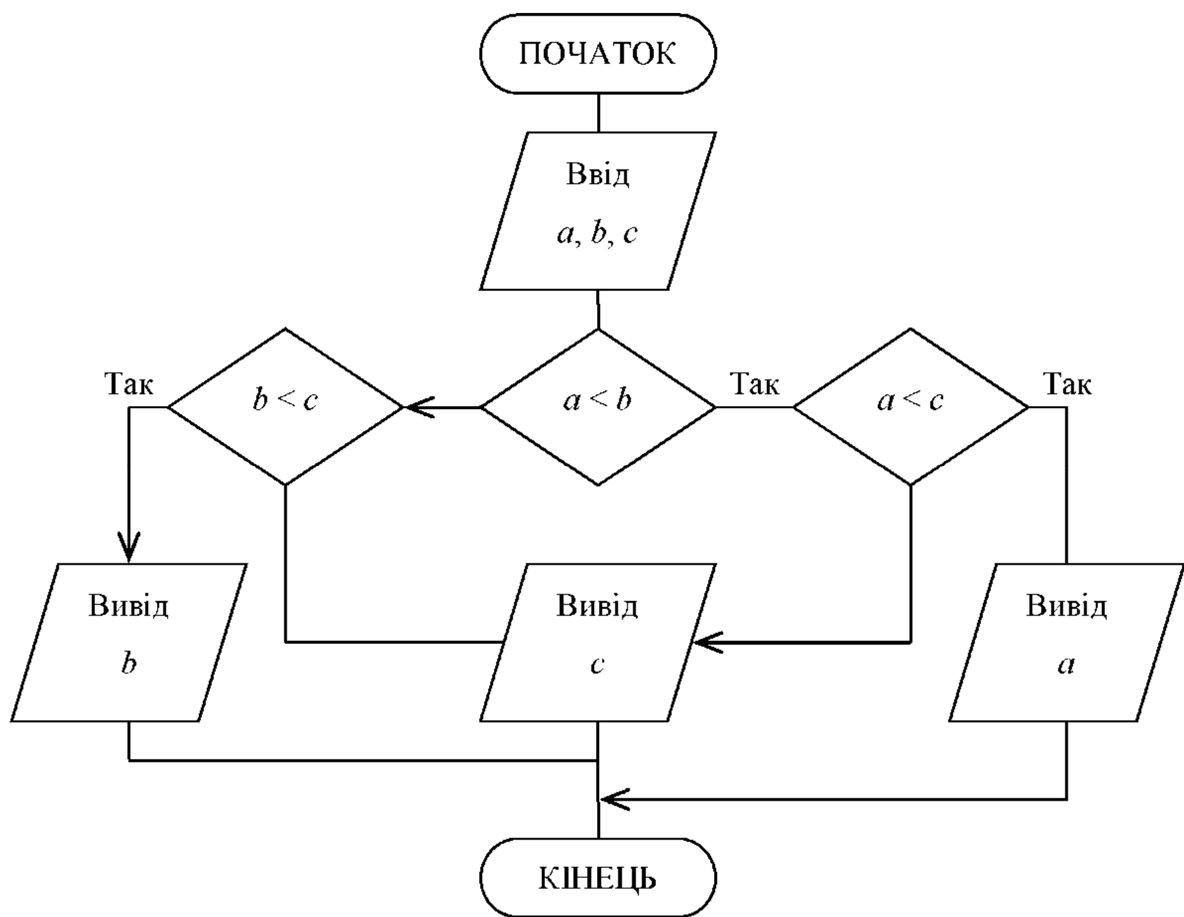


Рис. 2.10 – Схема алгоритму розв'язання задачі про найменше з трьох чисел

2.6 Циклічні алгоритми

Алгоритм називається *циклічним,* якщо одна і та ж послідовність дій в ньому повторюється декілька разів, доки не виконається задана умова.

Команда або група команд, виконання яких повторюється при кожному проходженні циклу, називається *тілом циклу.*

Ознакою циклічного алгоритму є наявність в ньому однієї із наступних структур:

- цикл з передумовою (цикл ПОКИ);
- цикл з післяумовою (цикл ДО);
- цикл з параметром (цикловою змінною).

Структура циклу з передумовою описується наступним чином:

ПОКИ є істиною **умова циклу** повторювати **тіло циклу**.

Схема алгоритму циклу з передумовою представлена на рис. 2.11 а. Особливістю циклу з передумовою є те, що в тому випадку, коли умова циклу заздалегідь не є справедливою, тіло циклу не виконається жодного разу (не відбудеться *вхід в цикл).*

Структура циклу з післяумовою описується наступним чином:

повторювати **тіло циклу ДО** виконання **умови циклу**.

Схема алгоритму циклу з післяумовою представлена на рис. 2.11 б. Особливістю циклу з післяумовою є те, що його тіло гарантовано виконається хоча б один раз незалежно від справедливості умови циклу.

Цикл з параметром використовується тоді, коли для кожного значення деякої змінної, яка називається *цикловою змінною, параметром циклу* або *лічильником,* необхідно виконати однаковий набір команд. Структуру циклу з параметром можна описати наступним чином:

ПОКИ лічильник циклу не набув останнього значення повторювати **тіло циклу.**

Таким чином, цикл з параметром за своєю структурою є циклом ПОКИ, в якому передумовою є набуття лічильником циклу певного кінцевого значення. Схема алгоритму циклу з параметром представлена на рис. 2.11 в.

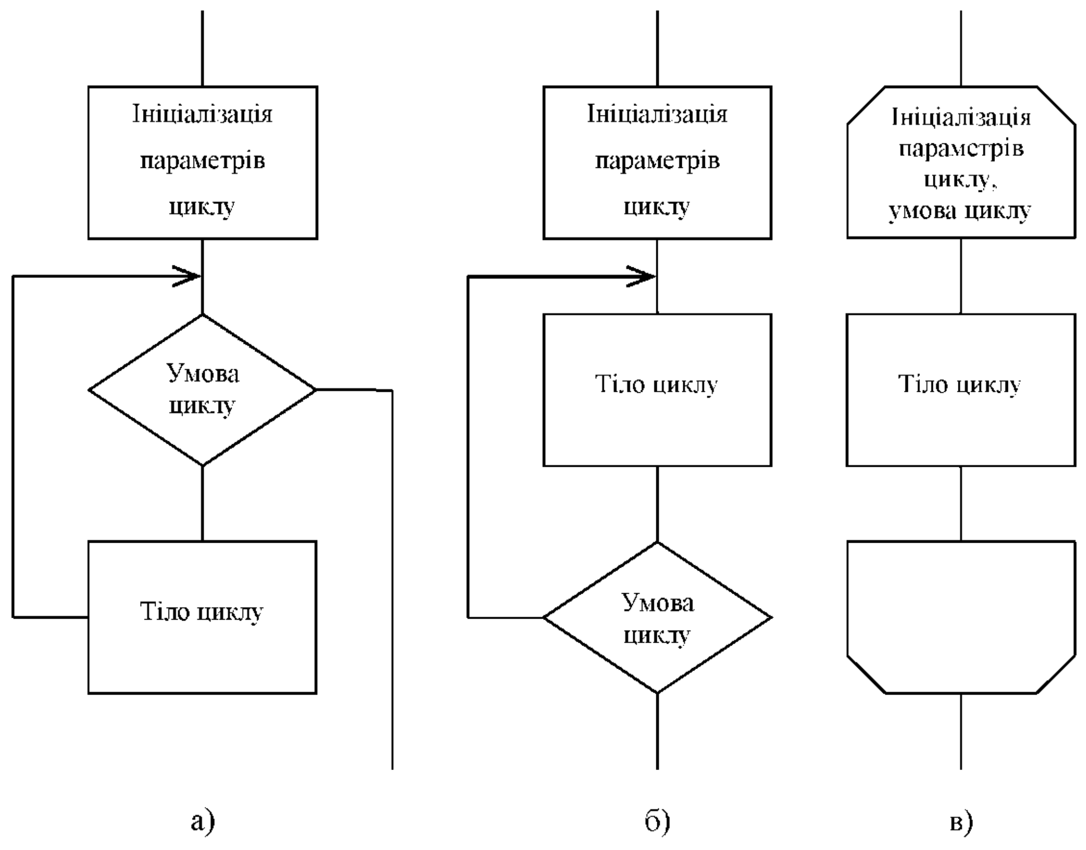


Рис. 2.11 – Схеми циклічних алгоритмів різної структури

Всередині символу, який відкриває цикл з параметром, записується інформація про лічильник в наступному форматі:

назва, початкове значення, кінцеве значення, крок зміни.

Крок зміни лічильника, що дорівнює одиниці, можна не вказувати

Приклади алгоритмів із розгалуженням

<u>Постановка задачі 1</u>: обчислити експоненту за допомогою ряду

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + ... + \frac{x^k}{k!} + ... , \qquad (2.14)$$

та оцінити кількість доданків, необхідних для отримання заданої точності обчислень.

Розв’язання:

Вхідними даними алгоритму є:

- значення показника експоненти - *x;*
- точність обчислень - ε.

Вихідними даними алгоритму є:

- значення експоненти - *y;*

- кількість кроків, витрачена на досягнення заданої точності – n

Спільний елемент даного ряду визначається виразом $\frac{x^p}{p!}$, де p - показник степеню, а підфакторіальне число - змінюється від нуля до нескінченності (факторіал нуля дорівнює одиниці) з кроком одиниця. Якщо б кількість доданків, необхідних для обчислення експоненти із заданою точністю була відома, циклічний алгоритм можна було б будувати на основі структури із параметром, однак, в нашому випадку останнє значення лічильника є невідомим, а, отже, алгоритм слід будувати на основі більш "м'якої" структури, наприклад - ДО.

Звернемо увагу на те, що факторіал, присутній в знаменнику ряду, який розглядається в задачі, є так званою *рекурсивною* функцією, тобто

$$n! = n\cdot(n-1)!,\ (n-1)! = (n-1)\cdot(n-2)!,\ ...,\ 2! = 2 \cdot 1!. \qquad (2.15)$$

Це означає, що кожне наступне значення факторіалу в знаменнику можна отримувати з попереднього множенням його на наступне значення лічильника. Отже, зникає необхідність у переобчисленні факторіалу "з нуля" на кожному кроці циклу.

Побудуємо алгоритм у словесному вигляді:

- ввести x, ε;
- задати початкове значення параметру циклу: $p = 1$;
- задати початкове значення суми елементів ряду: $S = 1$;
- задати початкове значення знаменника: z = 1;
- **тіло циклу**:
  1. обчислити черговий елемент ряду: $y = \frac{x^p}{z}$
  2. додати обчислений елемент ряду до суми: $S = S + y$;
  3. збільшити параметр циклу на одиницю: $p = p + 1$;
  4. обчислити нове значення знаменника: z = z * p;
  5. якщо черговий елемент ряду менше заданої точності: $y < \epsilon$ - вийти з тіла циклу;
  6. перейти до кроку 1 тіла циклу;
- кінець тіла циклу
- записати кількість витрачених кроків: $n = p$;
- вивести S і n.

Важливу роль в даному алгоритмі грає змінна *S,* яка виконує функцію суматора з накопиченням. Зазвичай, початкове значення суматора з накопиченням повинно дорівнювати нулю, що забезпечує правильне обчислення суми елементів деякої послідовності. Однак, в нашому випадку встановлення цього значення рівним одиниці дозволило почати цикл з обчислення другого елемента ряду і не вводити в алгоритм засоби, які б забезпечили правильне обчислення факторіалу нуля.

Схема алгоритму розв'язанні задачі представлена на наступному рис. 2.12.

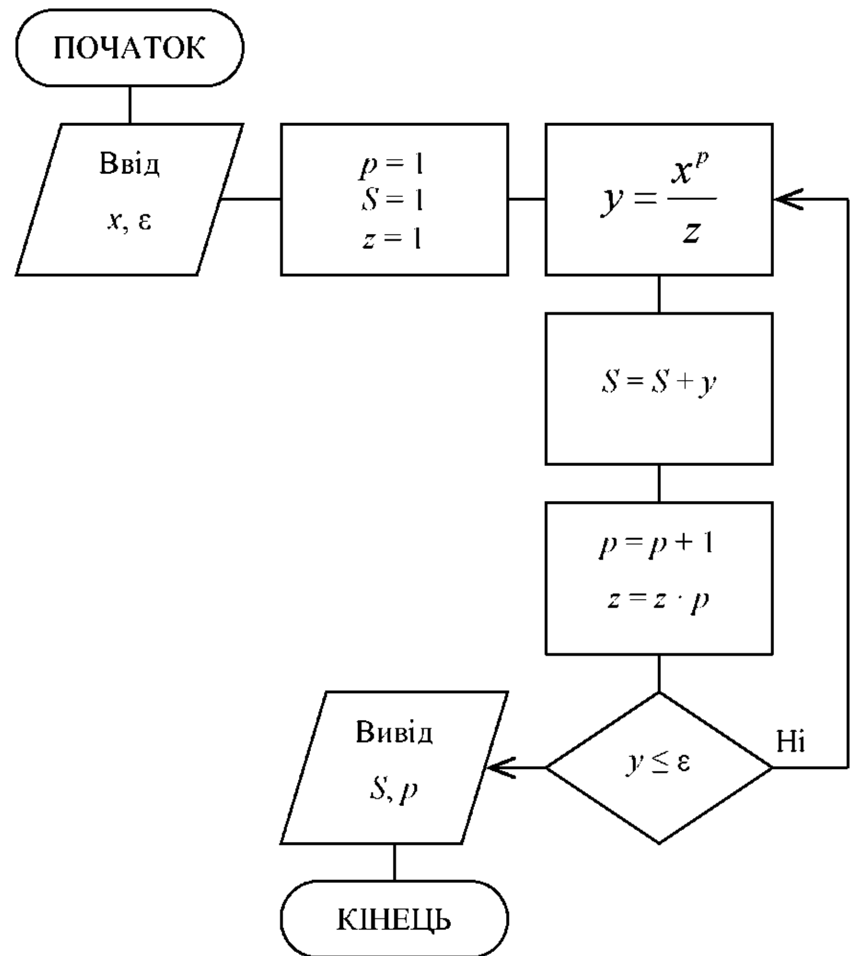


Рис. 2.12 - Схема алгоритму обчислення експоненти

В наступній таблиці представлено звіт з процесу тестування алгоритму для наступних вхідних значень:

- $x = 2$ ($e^2=7{,}39$);
- $\varepsilon = 0{,}01$.

p	**x^p**	**z**	**y**	**S**
1	2	1	2,00	3,00
2	4	2	2,00	5,00
3	8	6	1,33	6,33
4	16	24	0,67	6,99
5	32	120	0,27	7,27
6	64	720	0,09	7,36
7	128	5040	0,03	7,38

Таким чином, задана точність була досягнута за сім кроків циклу. Результат обчислення: 7,38.

Постановка задачі 2: вивести всі точки з цілочисельними координатами, які потрапляють в коло з радіусом R і центром на початку координат.

Розв’язання:

Вхідні дані алгоритму - радіус кола - R.

Вихідні дані - множина пар цілочисельних координат точок *(x,* у), які потрапляють в коло.

Будемо вважати, що точка потрапляє в коло тоді, коли відстань від центру кола до неї суворо менше ніж радіус кола.

Найпростіший (і достатньо ефективний) спосіб розв'язання цієї задачі - прямий перебір усіх точок з цілочисельними координатами, які знаходяться у прямокутній області, яка обмежує коло (рис. 2.13) та перевірка кожної з них на знаходження всередині кола. Перебір здійснюється для кожної координати окремо, тобто, для кожного значення координати *x* (або у) необхідно перебрати всі значення координати *y* (або х). Значення перебираються від мінус *R* до *R* з кроком одиниця.

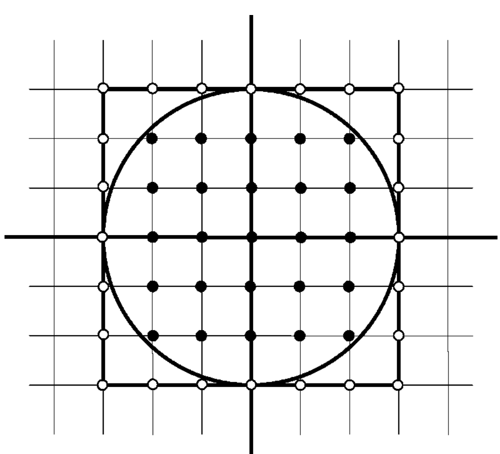


Рис. 2.13 - Ілюстрація до задачі про точки з цілочисельними координатами

Складемо алгоритм у текстовому вигляді:

ввести *R*;

• кінець тіла зовнішнього циклу

Зауважимо, що в даному алгоритмі використовується так званий *вкладений цикл,* тобто один цикл (внутрішній) є тілом іншого (зовнішнього). Під час виконання алгоритму на кожен крок зовнішнього циклу припадають всі кроки внутрішнього циклу. Якщо кількість кроків обох циклів однакова і дорівнює N, то загальна кількість кроків, за яку буде повністю виконано зовнішній цикл складатиме N1N.

Схема алгоритму представлена на рис. 2.14.

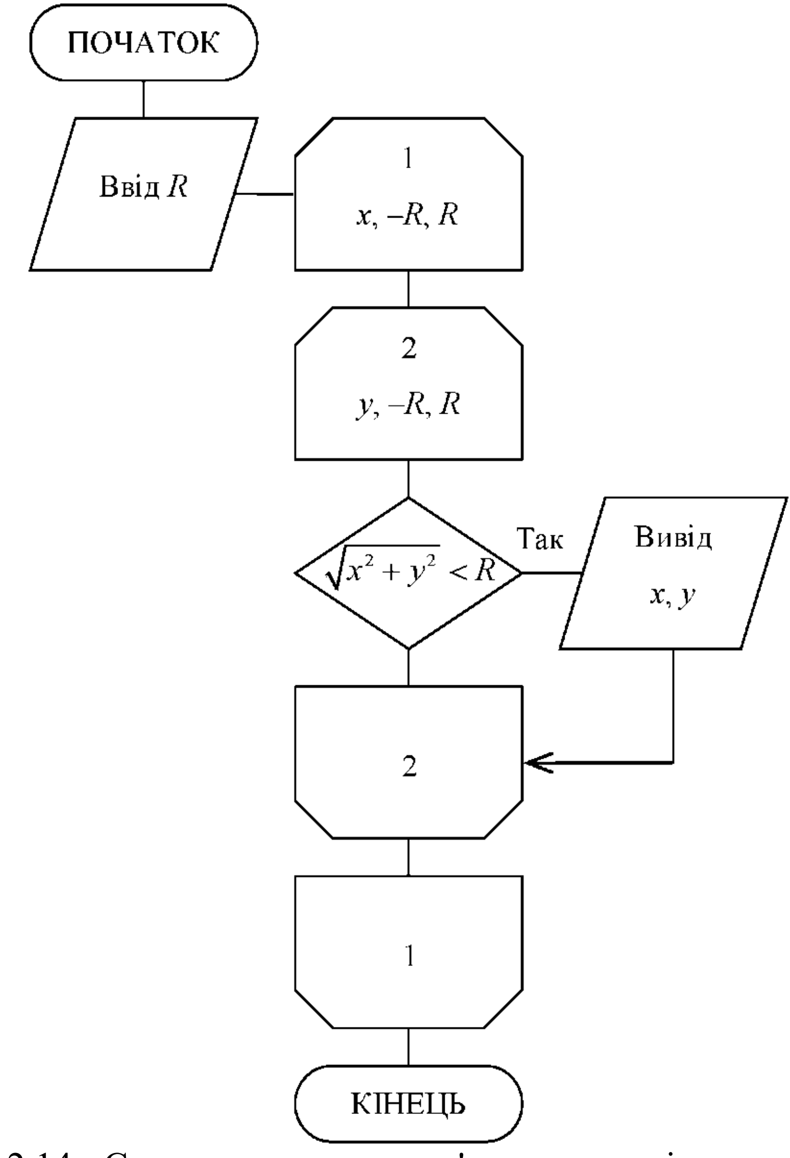


Рис2.14 - Схема алгоритму розв'язання задачі про точки з цілочисельними координатами

III. ОСНОВИ ТЕОРІЇ ОБЧИСЛЮВАЛЬНОСТІ

3.1 Принцип дедукції

Кажуть, що формула B логічно випливає з формули A, якщо формула B має значення I при всіх інтерпретаціях, при яких формула A має значення I. Кажуть, що формули A і B логічно еквівалентні (позначається A? B або просто A = B), якщо вони є логічним наслідком один одного. Логічно еквівалентні формули мають однакові значення істинності при будь-якій інтерпретації.

Найбільш короткий і простий спосіб виведення заснований на теоремі дедукції.

Основні схеми логічно правильних міркувань.

Поряд з алфавітом і правилами побудови складних висловлювань - логічних формул, мови логіки висловлювань містять *правила прео рення логічних формул.* В алгебрі логіки - це еквівалентні співвідношення, а також правило підстановки і правило заміни; в обчисленні висловлювань - це загальні логічні аксіоми і правила підстановки і висновку, звані правилами виведення. Правила перетворення реалізують загальні логічні закони і забезпечують логічно правильні міркування. Коректність допустимих в логіці перетворень є фундаментальним властивістю формальної (математичної) логіки.

Якщо опис системи (процесу, явища і т.п.) представлено сукупністю складних висловлювань - логічних формул, дійсних для даної системи (в даній інтерпретації її простих висловлювань), то за допомогою допустимих перетворень наявних логічних уявлень про систему може бути виконаний їх аналіз (синтез), можуть бути отримані нові уявлення, що характеризують зазначену систему (істинні для даної системи) і т.п. Таким чином, за допомогою допустимих в логіці перетворень з'являється можливість отримання нових знань з відомостей, вже наявних.

Процес отримання нових знань, виражених висловлюваннями, з інших знань, також виражених висловлюваннями, називається *міркуванням (умовиводом).* Вихідні висловлювання називаються *посилками (гіпотезами, умовами),* а одержувані висловлювання - *ув'язненням (наслідком).*

Наведемо приклади найбільш уживаних *схем логічно правильних міркувань:*

1. Правило висновку - стверджує модус (Modus Ponens):

"Якщо з вислову *A* слід вислів *B* і справедливо (істинно) висловлювання *A,* то справедливо *B"* (Спосіб спуску). Позначається:

$$\frac{A \to B, A}{B}.$$

2. Правило заперечення - негативний модус (Modus Tollens):

"Якщо з *A* випливає *B,* але вислів *У* невірно, то невірно *A "(Доказ від протилежного). Позначається:*

$$\frac{A \to B, \neg B}{\neg A}.$$

3. Правила твердження-заперечення (Modus Ponendo-Tollens):

"Якщо справедливо чи висловлювання А, або висловлювання *B* (в розділовому значенні) і істинно одне з них, то інше брехливо" (розділовий силогізм). Позначається:

$$\frac{A \oplus B, A}{\neg B}; \frac{A \oplus B, B}{\neg A}.$$

4. Правила заперечення-ствердження (Modus Tollen-Ponens):

а) "Якщо істинно або *A* або *B* (в розділовому значенні) і невірно одне з них, то істинне інше":

$$\frac{A \oplus B, \neg A}{B}; \frac{A \oplus B, \neg B}{A}.$$

б) "Якщо істинно *A* або *B* (в неразделітельном сенсі) і невірно одне з них, то істинно інше "(діз'юнктівную силогізм). Позначається:

$$\frac{A \vee B, \neg A}{B}; \frac{A \vee B, \neg B}{A}.$$

5. Правило транзитивності (спрощене правило силогізму):

"Якщо з *A* випливає *B,* а з *B* слід *C,* то з *A* випливає *C"* (гіпотетичний силогізм). Позначається:

$$\frac{A \to B, B \to C}{A \to C}.$$

6. Закон суперечності:

"Якщо з *A* випливає *B* і ¬ *B,* то невірно *A":*

$$\frac{A \to B, A \to \neg B}{\neg A}.$$

7. Правило контрапозиции:

"Якщо з *A* випливає *B,* то з того, що невірно *B,* випливає, що невірно *A":*

$$\frac{A \to B}{\neg B \to \neg A}.$$

8. Правило складної контрапозиции:

"Якщо з *A* і *B* слід *C,* то з *A* і ¬ *C* слід ¬ *B":*

$$\frac{(A \& B) \to C}{(A \& \neg C) \to \neg B}.$$

9. Правило перетину:

"Якщо з *A* випливає *B,* а з *B* і *C* слід *D,* то з *A* і C слід D":

$$\frac{A \to B, (B \& C) \to D}{(A \& C) \to D}.$$

Наведемо без пояснень ще кілька правил умовиводів.

10. Правило імпортації (об'єднання посилок):

$$\frac{A \to (B \to C)}{(A \& B) \to C}.$$

11. Правило експортації (роз'єднання посилок):

$$\frac{(A \& B) \to C}{A \to (B \to C)}.$$

12. Правила дилем:

а) $\frac{A \to C, B \to C, A \vee B}{C}$ (Проста конструктивна дилема);

б) $\frac{A \to B, C \to D, A \vee C}{B \to D}$ (Складна конструктивна дилема);

в) $\frac{A \to B, A \to C, \neg B \vee \neg C}{\neg A}$ (Проста деструктивна дилема);

г) $\frac{A \to B, C \to D, \neg B \vee \neg D}{\neg A \vee \neg C}$ (Складна деструктивна дилема).

Примітка. Для побудови логічних формул, що відображають зазначені вище логічно правильні міркування, слід все посилки з'єднати зв'язкою "І" (&) і отриману таким чином узагальнену посилку - зв'язкою "якщо ..., то ..." (?). Наприклад, правило ув'язнення (Modus Ponens) має бути представлено логічною формулою:

$$\frac{A \to B, A}{B} \Rightarrow ((A \to B) \,\&\, A) \to B.$$

Прикладами міркувань, які не є правильними, можуть служити:

а) $\frac{A \to B, B}{A}$

б) $\frac{A \to B, \neg A}{\neg B}$

в) $\frac{A \to B, A}{\neg B}$ та ін

Для того щоб перевірити, чи є даний умовивід логічно правильним, слід відновити схему міркування і визначити, чи відноситься вона до схем логічно правильних міркувань. Однак така перевірка ускладнюється тим, що схем логічно правильних міркувань нескінченна безліч. Для перевірки правильності міркувань може бути використаний метод докази від протилежного (закон протиріччя - правило 6).

Метод резолюцій в численні висловів. Метод резолюції є одним з методів доказу від супротивного. Метод, запропонований Дж. Робінсоном, в даний час є теоретичною базою більшості методів доказу. Хоча загальноприйняті правила виводу, наприклад, правило modus ponens, дозволяють людині простежити за кожним кроком процедури докази, існує більш сильний правило резолюцій, яке важко піддається сприйняттю, але ефективно реалізується на комп'ютері.

В даний час не існує ефективних критеріїв перевірки виконуваності КНФ. Метод резолюцій дозволяє виявити нездійсненність безлічі диз'юнктів.

3.2 Числення предикатів

Основні поняття та визначення.

В даному курсі лекцій опис формальної теорії числення предикатів носить конспективний характер, зокрема, багато складні докази опущені. У той же час, основна увага приділена прагматичним аспектам теорії, які можуть

принести велику користь інженеру-програмісту. Нагадаємо ще раз найбільш важливі аспекти обчислення предикатів.

Предикат - оповідної пропозицію, що містить *предметні змінні,* визначені на відповідних множинах. При заміні змінних конкретними значеннями (елементами) цих множин пропозицію звертається в висловлювання, тобто приймає значення "істинно" або "хибно". Позначення предиката, що містить *n* змінних *(n-місного* предиката): $P(x_1, x_2,..., x_n)$, при цьому передбачається, що $x_1 \in M_1, x_2 \in M_2,..., x_n \in M_n$.

За допомогою логічних зв'язок (і дужок) предикати можуть об'єднуватися в різноманітні логічні формули - *предикатні формули.* Дослідження предикатних формул і способів встановлення їх істинності є основним предметом *логіки предикатів.* Логіка предикатів разом з вхідною в неї логікою висловлень є основою логічного мови математики. З її допомогою вдається формалізувати і точно дослідити основні методи побудови математичних теорій. Логіка предикатів є важливим засобом побудови розвинених логічних мов і формальних систем (формальних теорій).

Логіка предикатів, як і логіка висловлювань, може бути побудована у вигляді *алгебри логіки* предикатів і *числення* предикатів. Тут, як і у випадку логіки висловлювань, для знайомства з основними поняттями логіки предикатів скористаємося мовою алгебри, а не числень. Такий вибір обумовлений рядом причин:

- Дослідження предикатних формул алгебри логіки, виконання їх перетворень значно простіше, ніж в численні предикатів.
- Обмеження у використанні апарата алгебри обумовлені тим, що предметні області (множини, на яких визначені предметні змінні предикатів) теоретично можуть бути і нескінченними. У таких випадках стандартний метод перевірки істинності предикатів і формул в цілому, що вимагає підстановки всіх можливих значень предметних змінних, не може бути здійснений в строгому сенсі (точніше, процедура обчислення істинності може бути нескінченною і не дати відповіді за кінцевий час). Однак у практичних ситуаціях при описі реальних систем, процесів, явищ як предметних областей, як правило, використовуються кінцеві безлічі. Тому проблема нескінченності в значній мірі втрачає свою актуальність.

n - місцевий предикат - це функція $P(x_1, x_2,..., x_n)$ від *n* змінних, що приймають значення з деяких *заданих* предметних областей, так що $x_1 \in M_1, x_2 \in M_2,.., x_n \in M_n$, а функція Р приймає два логічних значення - "істинно" або "хибно" (позначення: {І, Л}, {1, 0}). Таким чином, предикат $P(x_1, x_2,..., x_n)$ є функцією типу $P: M_1$ x M_2 x ... x M_n ? *B,* де безлічі $M_1, M_2,..., M_n$ називаються *предметними областями* предиката; $x_1, x_2,..., x_n$ - *предметними змінними* предиката; *B* - двійкове (бінарне) безліч: *B* = {І, Л} або {1,0}. Якщо предикатні змінні приймають значення на одному безлічі, то $P: M^{n?}$ *B*.

Відповідності між предикатами, відносинами і функціями:

1. Для будь-яких *M* і *n* існує взаємно однозначна відповідність між *n-місцевими* відносинами $R \subseteq M^n$ і *n* - місцевими предикатами *P* $(x_1, x_2,..., x_n)$, *P*: $M^{n?}$ *B:*

- кожному *n* - місцевому відношенню *R* відповідає предикат *P* $(x_1, x_2,..., x_n)$ такий, що *P* $(a_1, a_2,..., a_n) = 1$, якщо і тільки якщо $(a_1, a_2,..., a_n) \in R$;
- всякий предикат *P* $(x_1, x_2,..., x_n)$ визначає відношення *R* таке, що $(a_1, a_2,..., a_n) \in R$, якщо і тільки якщо *P* $(a_1, a\, x_2,..., a_n) = 1$.

При цьому *R задає область істинності предиката P.*

2. Усякої функції $f(x_1, x_2,..., x_n)$, f: M^n ? *M,* відповідає предикат *P* $(x_1, x_2,..., x_n, x_{n+1})$, *P*: $M^{n+1?}$ *B,* такий, що *P* $(a_1, a_2,..., a_n, a_{n+1}) = 1$, якщо і тільки якщо $f(a_1, a\, x_2,..., a_n) = a_{n+1}$.

Поняття предиката ширше поняття функції, тому зворотне відповідність (від $(n+1)$ - місцевого предиката до *n* - місцевої функції) можливо не завжди, а тільки для таких предикатів *P '* для яких виконується умова, пов'язане з вимогою однозначності функції.

3.3 Машини Тьюрінга. Обчислюваність за Тьюрінгом

Під (детермінованою) *машиною Тьюрінга* (скорочено МТ) будемо розуміти впорядковану 5-ку (Q, T, δ, q_0, q^*), де:

– Q – скінченна множина внутрішніх станів;
– T – скінченний алфавіт символів стрічки, причому T містить спеціальний символ порожньої клітки λ;
– $\delta : Q \times T \to Q \times T \times \{R, L, \varepsilon\}$ – однозначна функція переходів;
– $q_0 \in Q$ – початковий стан;
– $q^* \in Q$ – фінальний стан.

Функцію переходів на практиці задають скінченною множиною команд одного з 3-х видів: $qa \to pbR$, $qa \to pbL$ та $qa \to pb$, де $p, q \in Q$, $a, b \in T$, $\to \notin Q \cup T$. При цьому, як правило, не для всіх пар $(q,a) \in Q \times T$ існує команда з лівою частиною qa. Це означає, що функція δ не є тотальною. Проте зручніше вважати функцію δ тотальною, тому для всіх пар $(q,a) \notin D_\delta$ неявно (не додаючи відповідні команди вигляду $qa \to qa$) вводимо довизначення $\delta(q,a) = (q,a,\varepsilon)$.

Неформально МТ складається зі скінченної пам'яті, розділеної на клітки нескінченної з обох боків стрічки та головки читання-запису. В кожній клітці стрічки міститься єдиний символ із T, причому в кожен даний момент стрічка містить скінченну кількість символів, відмінних від символа λ. Головка читання-запису в кожен даний момент оглядає єдину клітку стрічки.

Якщо МТ знаходиться в стані q та головка читає символ a, то при виконанні команди $qa \to pbR$ (команди $qa \to pbL$, команди $qa \to pb$) МТ переходить у стан p, замість символу a записує на стрічці символ b та зміщує головку на 1 клітку направо (відповідно на 1 клітку наліво, залишає головку на місці).

Конфігурація, або *повний стан МТ*, – це слово вигляду xqy, де $x,y \in T^*$, $q \in Q$. Неформально це означає, що на стрічці записане слово xy, тобто зліва і справа від xy можуть стояти тільки символи λ, МТ знаходиться в стані q, головка читає 1-ий символ підслова y.

Конфігурацію вигляду q_0x, де 1-й та останній символи слова x відмінні від λ, називають *початковою*. Конфігурацію вигляду xq^*y називають *фінальною*. Після переходу до фінального стану, отже, до фінальної конфігурації МТ спиняється.

Нехай МТ знаходиться в конфігурації $xcqay$, де $x,y \in T^*$, $a, c \in T$, $q \in Q$. Після виконання команди $qa \to pbR$ (команди $qa \to pbL$, команди $qa \to pb$) МТ перейде до конфігурації $xcbpy$ (відповідно до конфігурації $xpcby$, конфігурації $xcpby$).

Кожна МТ задає вербальне відображення $T^* \to T^*$ таким чином.

МТ M переводить слово $u \in T$ у слово $v \in T^*$, якщо вона з початкової конфігурації q_0u переходить до фінальної конфігурації xqy, де $q \in F^*$, $xy = \alpha v \beta$, $\alpha, \beta \in \{\lambda\}^*$. При цьому перший та останній символи слова v відмінні від λ, або $v = \varepsilon$. Цей факт записуємо так: $v = M(u)$.

Якщо МТ M, починаючи роботу з початкової конфігурації q_0u, ніколи не спиниться, кажуть, що M *зациклюється* при роботі над словом u. Тоді $M(u)$ не визначене.

МТ M_1 та M_2 *еквівалентні*, якщо вони задають одне і те ж вербальне відображення.

МТ M *обчислює* часткову функцію $f: N^k \to N$, якщо вона кожне слово вигляду $|^{x_1}\#|^{x_2}\#...\#|^{x_k}$ переводить у слово $|^{f(x_1,...,x_k)}$ у випадку $(x_1,...,x_k) \in D_f$, та $M(|^{x_1}\#|^{x_2}\#...\#|^{x_k})$ не визначене при $(x_1,...,x_k) \notin D_f$.

Функція називається *обчислюваною за Тьюрінгом*, або *МТ-обчислюваною*, якщо існує МТ, яка її обчислює.

Зауважимо, що кожна МТ обчислює *безліч* функцій натуральних аргументів та значень, але, зафіксовуючи наперед *арність* функцій, дістаємо, що кожна МТ обчислює *єдину* функцію заданої арності.

Розглянемо приклади МТ.

Приклад 1. МТ, яка обчислює функцію $x+y$:

$$q_0| \to q_0|R$$
$$q_0\# \to q_0|R$$
$$q_0\lambda \to q_1\lambda L$$
$$q_1| \to q^*\lambda$$

Приклад 2. МТ, яка обчислює функцію $f(x)=\mathrm{sg}(x)$:

$$q_0\lambda \to q^*\lambda$$
$$q_0| \to q_1|R$$
$$q_1| \to q_1\lambda R$$
$$q_1\lambda \to q^*\lambda$$

Приклад 3. МТ, яка обчислює функцію $f(x, y) = x-y$:

$$q_0| \to q_1\lambda R$$
$$q_1| \to q_1|R$$
$$q_1\# \to q_1\#R$$
$$q_1\lambda \to q_2\lambda L$$
$$q_2| \to q_3\lambda L$$
$$q_3| \to q_3|L$$
$$q_3\# \to q_3\#L$$
$$q_3\lambda \to q_0\lambda R$$
$$q_2\# \to q^*|$$
$$q_0\# \to q_4\lambda R$$
$$q_4\lambda \to q^*\lambda$$

Приклад 4. МТ, яка обчислює функцію $f(x, y)= x \dot{-} y$

$$q_0| \to q_1\lambda R$$
$$q_1| \to q_1|R$$
$$q_1\# \to q_1\#R$$
$$q_1\lambda \to q_2\lambda L$$
$$q_2| \to q_3\lambda L$$
$$q_3| \to q_3|L$$
$$q_3\# \to q_3\#L$$
$$q_3\lambda \to q_0\lambda R$$
$$q_2\# \to q^*|$$
$$q_0\# \to q_4\lambda R$$
$$q_4| \to q_4\lambda R$$

(єдина відмінність від МТ для $f(x, y) = x-y$)

$$q_4\lambda \to q^*\lambda$$

3.4 Нормальні алгоритми Маркова. Обчислюваність за Марковим

Під *нормальним алгоритмом* (скорочено НА) *в алфавіті* T розуміють упорядковану послідовність продукцій (правил) вигляду $\alpha\to\beta$ або $\alpha\to\cdot\beta$, де $\alpha, \beta\in T^*$ та $\cdot\,, \to\notin T$. Продукції вигляду $\alpha\to\cdot\beta$ називають *фінальними*.

Кожен НА в алфавіті T задає деяке вербальне відображення $T^*\to T^*$. Слово, яке є результатом обробки слова x нормальним алгоритмом D, позначимо D(x). Обробка слова x нормальним алгоритмом D проводиться поетапно таким чином.

Покладемо $x_0=x$ і скажемо, що x_0 отримане із x після 0 етапів. Нехай слово x_n отримане із слова x після n етапів. Тоді $(n+1)$-й етап виконується так.

Шукаємо першу за порядком продукцію $\alpha\to\beta$ або $\alpha\to\cdot\beta$ таку, що α – підслово x_n. Застосуємо цю продукцію до x_n, тобто замінимо в x_n найлівіше входження α на β. Отримане слово позначимо x_{n+1}. Якщо застосована на $(n+1)$-му етапі продукція нефінальна, тобто $\alpha\to\beta$, то переходимо до $(n+2)$-го етапу. Якщо ця продукція фінальна, тобто $\alpha\to\cdot\beta$, то після її застосування $\mathtt{D}$ спиняється і $\mathtt{D}(x)=x_{n+1}$. Якщо ж на $(n+1)$-му етапі жодна продукція $\mathtt{D}$ не застосовна до x_{n+1}, тобто в $\mathtt{D}$ немає продукції, ліва частина якої – підслово слова x_{n+1}, то $\mathtt{D}$ спиняється і $\mathtt{D}(x)=x_n$.

Якщо в процесі обробки слова x НА $\mathtt{D}$ не спиняється ні на якому етапі, то вважаємо, що $\mathtt{D}(x)$ не визначене.

Нормальний алгоритм називають нормальним алгоритмом *над алфавітом* T, якщо він є нормальним алгоритмом у деякому розширенні $T'\supseteq T$. НА над T задає певне відображення $T^*\to T^*$, використовуючи в процесі обробки слів допоміжні символи поза алфавітом T. Зупинка НА $\mathtt{D}$ над T при роботі над словом $x\in T^*$ *результативна,* коли вона відбулася на слові $y\in T^*$, інакше результат роботи $\mathtt{D}$ над x не визначений.

НА $\mathtt{D}$ і $\mathtt{E}$ *еквівалентні відносно алфавіту* T, якщо для всіх $x\in T^*$ $\mathtt{D}(x)$ та $\mathtt{E}(x)$ одночасно визначені або не визначені, та у випадку визначеності $\mathtt{D}(x)=\mathtt{E}(x)$.

Відомо, що для кожного НА над алфавітом T існує еквівалентний йому відносно T НА в алфавіті $T\cup\{\mathrm{s}\}$ з єдиним допоміжним символом $\mathrm{s}\notin T$. Відомо також, що вербальне відображення, яке кожне слово $x\in T^*$ переводить у слово xx, не може бути заданим жодним НА в алфавіті T. У той же час маємо НА, який кожне $x\in T^*$ переводить у слово xx (тут $\#\notin T$):

Приклад 1.

$\#\#a\to a\#a\#\#$ для всіх $a\in T$

$\#ab\to b\#a$ для всіх $a, b\in T$

$\#a\to a$ для всіх $a\in T$

$\#\#\to\cdot\varepsilon$

$\varepsilon\to\#\#$

НА $\mathtt{D}$ *обчислює* часткову функцію $f{:}N^k\to N$, якщо він кожне слово вигляду $|^{x_1}\#|^{x_2}\#...\#|^{x_k}$ переводить у слово $|^{f(x_1,...,x_k)}$ у випадку $(x_1,...,x_k)\in D_f$ та $\mathtt{D}(|^{x_1}\#|^{x_2}\#...\#|^{x_k})$ не визначене при $(x_1,...,x_k)\notin D_f$.

Функція називається *обчислюваною за Марковим*, або *НА-обчислюваною*, якщо існує НА, який її обчислює.

Зауважимо, що кожний НА обчислює *безліч* функцій натуральних аргументів та значень, але, зафіксовуючи наперед *арність* функцій, дістаємо, що кожний НА обчислює *єдину* функцію заданої арності.

Приклад 2. НА для функції $f(x, y)=x+y$:

$\#\to\varepsilon$

Приклад 3. НА для функції $f(x, y)=x-y$:

$$|\#|\to\#$$
$$\#|\to\#|$$
$$\#\to\varepsilon$$

Приклад 4. НА для функції $f(x)=x/2$:

$$\#||\to|\#$$
$$\#|\to\#|$$
$$\#\to\cdot\varepsilon$$
$$\varepsilon\to\#$$

3.5 Система Поста. Обчислюваність за Постом

Канонічною системою Поста над алфавітом T назвемо формальну систему (T^*, A, P), у якій множина аксіом A є скінченною підмножиною множини T^*, а множина правил виведення P складається зі слів вигляду $\alpha_0 S_1\alpha_1...\alpha_{m-1}S_m\alpha_m\to\beta_0 S_{j_1}\beta_1..\beta_{n-1}S_{j_n}\beta_n$. Тут $\to\notin T$, усі α_k та β_i – фіксовані слова із T^*, усі символи $S_k\notin T$, причому всі $j_i\in\{1,...,m\}$.

Символи S_k призначені для позначення довільних слів із T^*.

Системи Поста звичайно позначають у вигляді $\mathrm{S}=(T, A, P)$.

Множина правил P визначає на словах із T^* відношення *безпосереднього* виведення таким чином: $\sigma\Rightarrow_P\tau$, якщо існує правило

$$\alpha_0 S_1\alpha_1...\alpha_{m-1}S_m\alpha_m\to\beta_0 S_{j_1}\beta_1..\beta_{n-1}S_{j_n}\beta_n\in P,$$

таке, що для деяких слів $\varphi_1, ..., \varphi_m\in T^*$ маємо

$$\sigma=\alpha_0\varphi_1\alpha_1...\varphi_m\alpha_m,\ \tau=\beta_0\varphi_{j_1}\beta_1..\beta_{n-1}\varphi_{j_n}\beta_n.$$

Рефлексивно-транзитивне замикання відношення $\Rightarrow_P$ позначаємо $\Rightarrow_P^*$. Інакше кажучи, $\sigma\Rightarrow_P^*\tau$ означає, що слово τ отримане зі слова σ за допомогою скінченної кількості застосувань правил із P.

Слово τ породжується системою Поста S, якщо $\alpha\Rightarrow_P^*\tau$ для деякої $\alpha\in A$. Цей факт записуємо $_P|\!-\tau$ та називаємо таке слово τ *теоремою* системи Поста S.

Множину $Th(\mathrm{S})=\{\tau\in T^*|\ _P|\!-\tau\}$ називатимемо *множиною теорем* системи Поста S.

Для завдання системи Поста достатньо вказати множину правил та множину аксіом. У випадку необхідності вказуємо й алфавіт T.

Приклад 1. Система Поста із $A=\{a,b,\varepsilon\}$ та $P=\{S\to aSa,\ S\to bSb\}$ породжує всі слова-паліндроми в алфавіті $\{a,b\}$, тобто слова, які читаються однаково зліва направо і справа наліво.

Множина $X\subseteq T^*$ *породжувана за Постом*, якщо існують алфавіт $T'\supseteq\ \supseteq T$ та система Поста $\mathrm{S}=(T', A, P)$, такі, що $Th(\mathrm{S})\cap(T^*)=X1$

Обчислюваність функцій за Постом – це породжуваність за Постом графіків таких функцій.

Часткова функція $f: N^k \rightarrow N$ *обчислювана за Постом,* якщо породжуваною за Постом є множина $\{|^{x_1}\#|^{x_2}\#...\#|^{x_k}\#|^{f(x_1,...,x_k)} \mid (x_1,...,x_k) \in D_f\}$.

Наведемо приклади функцій і предикатів, обчислюваних за Постом.

Приклад 2. Система Поста для функції $f(x, y)=x+y$:

$$A = \{\#\#\};$$
$$P = \{X\#Y\#R \rightarrow X|\#Y\#R|,$$
$$X\#Y\#R \rightarrow X\#Y|\#R|\ \}.$$

Приклад 3. Система Поста для функції $f(x, y)=x-y$:

$$A = \{\#\#\};$$
$$P = \{X\#Y\#R \rightarrow X|\#Y\#R|,$$
$$X\#Y\#R| \rightarrow X\#Y|\#R\ \}.$$

Приклад 4. Ще одна система Поста для функції $f(x, y)=x-y$:

$$A = \{\#\#\};$$
$$P = \{X\#Y\#R \rightarrow X|\#Y|\#R,$$
$$X\#\#R \rightarrow X|\#\#R|\ \}.$$

Приклад 5. Система Поста для предиката "$x=y$":

$$A = \{\#\#|\};$$
$$P = \{X\#Y\#R \rightarrow X|\#Y|\#R|,$$
$$X\#\#R \rightarrow X|\#\#,$$
$$\#Y\#R \rightarrow \#Y|\#\ \}.$$

IV. АЛГОРИТМІЧНІ СТРАТЕГІЇ

4.1 Поняття та види стратегій

Поняття алгоритмічної стратегії дозволяє класифікувати базові алгоритми обробки даних по групах їх використання. Сучасніше розуміння терміну відноситься до програмних реалізацій алгоритмів.

Стратегія, Strategy - поведінковий шаблон проектування, призначений для визначення сімейства алгоритмів, інкапсуляції кожного з них і забезпечення їх взаємозамінюваності. Це дозволяє вибирати алгоритм шляхом визначення відповідного класу. Шаблон Strategy дозволяє міняти вибраний алгоритм незалежно від об'єктів- клієнтів, які його використовують.

Базові алгоритми обробки даних є результатом досліджень і розробок, що проводилися упродовж десятків років. Але вони, як і раніше, продовжують відігравати важливу роль в застосуванні обчислювальних процесів, що все розширюється.

До базових алгоритмів програмування можна віднести:

- Алгоритми роботи із структурами даних. Вони визначають базові принципи і методологію, використовувані для реалізації, аналізу і порівняння алгоритмів. Дозволяють отримати уявлення про методи представлення даних. До таких структур відносяться зв'язні списки і рядки, дерева, абстрактні типи даних, такі як стеки і черги.
- Алгоритми сортування, призначені для впорядкування масивів і файлів, мають особливу важливість. З алгоритмами сортування пов'язані, зокрема, черги по пріоритету, завдання вибору і злиття.
- Алгоритми пошуку, призначені для пошуку конкретних елементів у великих колекціях елементів. До них відносяться основні і розширені методи пошуку з використанням дерев і перетворень цифрових ключів, у тому числі дерева цифрового пошуку, збалансовані дерева, хешування, а також методи, які підходять для роботи з дуже великими файлами.
- Алгоритми на графах корисні при рішенні ряду складних і важливих завдань. Загальна стратегія пошуку на графах розробляється і застосовується до фундаментальних завдань зв'язності, у тому числі до завдання відшукування найкоротшого шляху, побудови мінімального остовного дерева, до завдання про потоки в мережах і завданні про паросочетаниях. Уніфікований підхід до цих алгоритмів показує, що в їх основі лежить одна і та ж процедура, і що ця процедура базується на основному абстрактному типі даних черги по пріоритету.
- Алгоритми обробки рядків включають ряд методів обробки (довгих) послідовників символів. Пошук в рядку призводить до зіставлення з еталоном, що у свою чергу веде до синтаксичного аналізу. До цього ж класу завдань можна віднести і технології стискування файлів.

- Геометричні алгоритми - це методи рішення завдань з використанням точок і ліній (і інших простих геометричних об'єктів), які увійшли до вживання досить нещодавно. До них відносяться алгоритми побудови опуклих оболонок, заданих набором точок, визначення перетинів геометричних об'єктів, рішення завдань відшукування найближчих точок і алгоритму багатовимірного пошуку. Багато хто з цих методів доповнює прості методи сортування і пошуку.

Виділимо алгоритмічні стратегії, які використовуються в алгоритмах :

- алгоритми грубої сили;
- жадібні алгоритми;
- "Розділяй і володарюй";
- алгоритми з поверненням;
- евристичні алгоритми;
- зіставлення із зразком і алгоритми обробки рядків/текстів;
- алгоритми чисельної апроксимації;
- online- і offline-алгоритми;
- динамічне програмування; і інші.

Розглянути реалізацію усіх вище приведених стратегій у рамках даного курсу представляється неможливо складним, та і потреби більшості завдань обмежуються лише вузьким кругом алгоритмічних стратегій, які, так або інакше, зустрінуться нам нижче.

Представимо основні парадигми.

Рекурсія - фундаментальне поняття в математиці і комп'ютерних науках. У мовах програмування рекурсивною програмою називається програма, яка звертається сама до себе. Рекурсивна програма не може викликати себе до безкінечності, отже, друга важлива особливість рекурсивної програми - наявність умови завершення, що дозволяє програмі припинити викликати себе.

Таблиця 4.1. Основні алгоритмічні стратегії

Тип алгоритму	**Ідея алгоритму і "при родящего ефективності**	**Діапазон трудаем-кісток**
Рекурсивний або звичайний розподіл	"Розділяй і володарюй": завдання розбивається на ідентичні підзадачі, результати яких об'єднуються в загальне рішення	$N \ldots N \log N \ldots N^2$
Повний перебір	"Гірше не буває"(без коментарів)	$2^N \ldots N^N \ldots N!$
Динамічне програмування	"Де жа ию": запам'ятовування результатів підзадач, що повторюються, збільшення продуктивності за рахунок додаткової пам'яті.	
Жадібний яскраво-червоний горитм	"Лицар на роздоріжжі": локальний вибір єдиної з підзадач на кожному кроці дає глобальне оптимальне рішення	$\log N \ldots N$

Рекурсивне рішення задачі полягає в декомпозиції великого завдання на дрібніші підзадачі того ж виду, рішенні цих підзадач і в наступному об'єднанні отриманих рішень для формування рішення початкової задачі. Підзадачі вирішуються рекурсивно тим же самим алгоритмом. В результаті декомпозиції

мають бути, кінець кінцем, отримані підзадачі такі прості (малій розмірності), що їх рішення може бути отримане безпосередньо.

Така алгоритмічна парадигма називається рекурсивною декомпозицією. Алгоритми, засновані на рекурсивній декомпозиції, аналізуються за допомогою рекурентних стосунків. Приклад рекурентного відношення визначення функції факторіалу :

$$n! = n*(n-1)! \quad \text{При } n>1$$
$$n! = 1 \quad \text{при } n=0.$$

Рекурсивну програму завжди можна перетворити в нерекурсивну (ітеративну, використовуючу цикли), яка виконує ті ж обчислення. І навпаки, використовуючи рекурсію, будь-яке обчислення, що припускає використання циклів, можна реалізувати, не прибігаючи до циклів.

Рекурентне співвідношення це рекурсивна функція з цілочисельними значеннями. Значення будь-якої такої функції можна визначити, обчислюючи усі її значення починаючи з найменшого, використовуючи на кожному кроці раніше вичислені значення для підрахунку поточного значення. Рекурентні вирази використовуються, зокрема, для визначення складності рекурсивних обчислень.

Наприклад, при спробі вичислити числа Фібоначчі за рекурсивною схемою $F(i) = F(i - 1) + F(i - 2)$, при $N >= 1$; $F(0) = 0$; $F(1) = 1$;

Текст програми буде приблизно таким:

```
Function F( n : integer ) : longint;
 begin
  if n < 2 then F := n
         else F := F(n-1) + F(n-2)
 end;
```

Кількість рекурсивних викликів при обчисленні значення F(N) за такою схемою може бути отримана з рішення рекурентного виразу $T_N = T_{N-1} + T_{N-2}$, при $N>=1$; $T_0 = 1$; $T_1 = 1$, де

T_N приблизно рівний Φ^N, де $\Phi \sim 1.618$ - золота пропорція, тобто приведена вище програма зажадає експоненціальних тимчасових витрат на обчислення.

Рекурсивне програмування дає загальний підхід до рішення завдань - розбиття їх на аналогічні підзадачі меншої розмірності, або на завдання, що є кроками в можливих напрямах її рішення. Так або інакше, рекурсивні виклики утворюють деревовидну структуру, кількість вершин в якій визначає ефективність алгоритму (виразиму зазвичай через трудомісткість). Різниця між різними типами алгоритмів полягає в способі отримання підзадач, їх розмірності, способі з'єднання отриманих результатів.

Ідея рекурсивного або звичайного розподілу сходить до технологічного прийому - модульного програмування. У своєму початковому варіанті вона припускає розбиття на завдання різної природи. Не рекурсивний розподіл дозволяє досягти певного ефекту за рахунок розбиття завдання на безліч ідентичних завдань меншої розмірності з наступним об'єднанням результату. Застосування того ж самого алгоритму рекурсивне до отриманих підзадач дає

наступний клас алгоритмів - рекурсивний розподіл. Як правило, незалежність отриманих підзадач має на увазі відповідний розподіл початкових даних завдання на підмножини, що не перетинаються, - цьому і відповідає сам термін розподіл. Ефективність (і трудомісткість) таких алгоритмів, залежить від витрат на само розподіл і від пропорцій частин, що розділяються : кращому випадку відповідає ділення на рівні частини (логарифмічні залежність), гіршому - виділення єдиного елементу (лінійні залежності).

Жадібні алгоритми. Ідеальним випадком можна вважати алгоритм, здатний "вибрати з декількох зол" єдино правильне. У основі його так само лежить принцип розподілу, але в кожній точці він має основу вибрати одну з підзадач. Зазвичай це робиться на підставі особливостей організації оброблюваних даних або їх надмірності. Основою жадібних алгоритмів є завжди досить спірне твердження: рух "по лінії найменшого опору" в кожній точці приведе до бажаного результату.

Повний перебір (вичерпний, комбінаторний перебір). Перелічені вище підходи засновані на всіляких "хитрощах", заснованих на особливостях предметної області алгоритму. Якщо ж нічого не допомагає, то залишається повний перебір усіх можливих варіантів рішення задачі.

Динамічне програмування. В процесі породження дерева рекурсивних викликів можливе повторення підзадач з одними і тими ж даними. Якщо запам'ятовувати результат їх виконання, то ефективність алгоритму може бути значно збільшена. Вивчення рекурсії нерозривно пов'язане з вивченням рекурсивно визначуваних структур даних, званих деревами (trees). Дерева використовуються як для спрощення розуміння і аналізу рекурсивних програм, так і в якості явних структур даних. У свою чергу, рекурсивні програми використовуються для побудови дерев. Глобальний зв'язок між ними (і рекурентними стосунками) використовується при аналізі алгоритмів. Багато алгоритмів використовують два рекурсивні виклики, кожен з яких працює приблизно з половиною вхідних даних. Така рекурсивна схема, по- видимому є найбільш важливим випадком добре відомого методу "розділяй і володарюй" (divide and conquer) розробки алгоритмів. В якості прикладу розглянемо завдання відшукування максимального з N елементів, збережених в масиві a[1], . . ., a[N] з елементами типу Item. Це завдання легко може бути вирішене за один прохід масиву

```
Max:=a[1];
For i:=1 to N do
  if a[i] > Max then Max:=a[i];
```

Рекурсивне рішення типу "розділяй і володарюй ще один простий (хоча абсолютно інший) спосіб рішення тієї ж задачі :

```
Function Max (a array of Item; l, r : integer) : Item;
var u, v : Item; m : integer;
begin
  m := (l+r) / 2;
  if (l = r)
    then Max := a[l]
    else begin
        u := Max (a, l, m);
        v := Max (a, m+1, r);
        if (u > v) then Max := u else Max := v
    end
end;
```

Більшість сучасних мов високого рівня підтримують механізм рекурсивного виклику, коли функція, як елемент структури мови програмування, повертає вичислене значення по своєму імені, може викликати сама себе з іншим аргументом. Ця можливість дозволяє безпосередньо реалізовувати обчислення рекурсивно певних функцій. Відмітимо, що через тезу Черча - Тюринга апарат рекурсивних функцій Черча равний за потужністю машині Т'юринга, і, отже, будь-який рекурсивний алгоритм може бути реалізований ітераційно.

Найчастіше підхід "розділяй і володарюй" використовують через те, що він забезпечує швидші рішення, ніж ітераційні алгоритми. Основним недоліком алгоритмів типу "розділяй і володарюй" являється те, що ділять завдання на незалежні підзадачі. Коли підзадачі незалежні, це часто призводить до неприпустимо великих витрат часу, оскільки одні і ті ж підзадачі починають вирішуватися багато разів.

Аналіз трудомісткості рекурсивних реалізацій алгоритмів, очевидно, пов'язаний як з кількістю операцій, що виконуються при одному виклику функції, так і з кількістю таких викликів. Графічне представлення породжуваного цим алгоритмом ланцюжка рекурсивних викликів називається деревом рекурсивних викликів. Детальніший розгляд призводить до необхідності обліку витрат як на організацію виклику функції і передачі параметрів, так і на повернення вичислених значень і передачу управління в точку виклику. Можна помітити, що деяка гілка дерева рекурсивних викликів обривається досягши такого значення передаваного параметра, при якому функція може бути вичислена безпосередньо. Таким чином, рекурсія еквівалентна конструкції циклу, в якому кожен прохід є виконання рекурсивної функції із заданим параметром. Дерево рекурсивних викликів може мати і складнішу структуру, якщо на кожному виклику породжується декілька звернень - фрагмент дерева рекурсій для чисел Фібоначчі представлений на рис. 4.1.

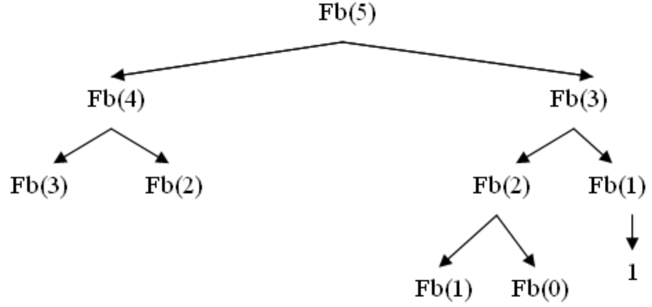


Рис. 4.1. Фрагмент дерева рекурсії при обчисленні чисел Фібоначчі

4.2 Методи розробки алгоритмів

Розкладання завдання в послідовність різнорідних підзадач

Цей метод називають іноді методом **"розділяй і володарюй"**.

У цьому методі зазвичай виділяється відносно невелике число підзадач. Наприклад: завдання - виконати програму на ЕОМ; підзадачі - ввести початковий текст програми; транслювати програму в машинні команди; приєднати до машинного коду стандартні процедури з бібліотеки; завантажити програму в оперативну пам'ять; запустити про-цесс виконання; завершити процес виконання програми.

Результати рішення першої підзадачі стають початковими даними для другої підзадачі і т. д. Таким чином, тут використаний другий підхід в чистому вигляді - декомпозиція функції, завдання її суперпозицією простіших. Помітимо також, що така суперпозиція може бути задана послідовним з'єднанням машин Т'юринга.

На алгоритмічній мові цей метод може бути виражений записом процедур, що послідовно викликаються.

Розкладання завдання в послідовність однорідних підзадач (ітерація)

Важливий окремий випадок попереднього методу, що придбаває, однак, нова якість за рахунок того, що завдання P зводиться до n екземплярів більш простого завдання R і до простого завдання Q, що об'єднує n рішень.

Дуже простий приклад: обчислення скалярного твору двох векторів A і B :

S:=0; {Завдання Q - підготовка місця для підсумовування.}

for i:= 1 to n do_

S:=S+A[i]*B[i]; {Завдання R - перемножування компонент і сумування.}

Цей алгоритм використовує розбиття початкових даних на частини - окремі компоненти векторів.

Однорідність підзадач дозволяє значно скоротити довжину тексту алгоритму за рахунок застосування операторів повторення. Ітерація на рівні великих підзадач або окремих невеликих операторів зустрічається у більшості реальних алгоритмів і служить основним джерелом ефективного використання комп'ютера в порівнянні з іншими обчислювальними засобами (наприклад, непрограмованим калькулятором).

Зведення завдання до самої собі (рекурсія)

Завдання також, як і в попередньому методі зводиться до більше за просту. Але це простіше завдання має те ж формулювання, що і початкова, з тією лише різницею, що вирішуватися вона повинна для простіших початкових даних. Це чистий варіант спрощення початкових даних.

Метод послідовних наближень

Спочатку яким-небудь чином вгадується значення x0, близьке до рішення. Завдання P знаходження рішення зводиться до багатократного рішення завдання R поліпшення рішення. Метод припускає, що якимсь чином може бути оцінена "якість" рішення (зазвичай - точність). Найчастіше абсолютна точність недосяжна, тому процес потенційно нескінченний, т. е. не виконується властивість скінченності алгоритму. Для того, щоб цього уникнути, дещо змінюють первинне формулювання завдання: вимагають відшукати не точне

рішення Y, а будь-яке рішення, отли-чающееся від Y не більше ніж на деяку величину Δ - тобто наближене рішення. Характерний приклад - завдання відшукування кореня рівняння або завдання пошуку кореня p -ої міри з x.

Основною проблемою є побудова завдання R по початковій задачі P, доказ факту збіжності процесу до шуканого рішення і забезпечення досить високої швидкості збіжності.

Цей варіант методу ітерацій використовується зазвичай для завдань, в яких шуканий результат Y виражається за допомогою речових або комплексних чисел. В усякому разі, на безлічі рішень повинна існувати метрика і повинно бути гарантовано, що задовільні наближені рішення існують. Передбачається, що початкові дані не розбиваються на частини, не спрощуються, а використовуються на кожному кроці ітерації. Причому, якщо в загальному методі ітерацій з розбиттям вихідних даних зазвичай не важливо, в якому порядку проводяться окремі кроки ітерації (вони можуть бути проведені одночасно, якщо дозволяє апаратура ЕОМ), то в методі послідовних наближень цей порядок дуже важливий.

Рішення зворотної задачі

Іноді зворотне завдання, т. е. завдання, що відповідає функції f, - 1(Y)=X, вирішується значно простіше, ніж початкове завдання. Тоді наявний алгоритм рішення зворотної задачі R іноді можна використовувати для побудови алгоритму рішення прямої задачі P._

Метод повного перебору

Коли говорили про рішення задачі обчислення f(X) шляхом декомпозиції функції f або розбиття початкових даних X на частини, то малося на увазі, що існують математичні результати або інші підстави для таких перетворень. Інакше кажучи, завдання аддитивне - її рішення може бути отримане об'єднанням рішень приватних завдань.

У ряді випадків це не так. Розглянемо, наприклад, завдання про рюкзак.

Дані ціле ненегативне число N і k чисел {n1, n2, n3, ..., nk}; знайти підмножину в {n1, n2, n3, ..., nk}, сума чисел якого рівна N, якщо така підмножина існує.

Немає ніяких підстав для того, щоб розділити початкові дані на частини і вирішувати завдання для спрощених початкових даних. Саме формулювання завдання також не вказує ніякої можливості декомпозиції.

Вихід виявляється одночасно і простим і складним. Можна узяти деяку підмножину і безпосередньою перевіркою (сумуванням чисел з цієї підмножини і порівнянням суми з N) дізнатися, чи задовольняє цю підмножину поставленій умові. Оскільки різних підмножин є кінцева кількість 2k, то потенційно можна перебрати усі підмножини і знайти рішення.

Складність полягає в тому, що зі збільшенням кількості вихідних даних (переході від k до k+1) швидко збільшується необхідне число перевірок. При k = 10 їх буде 1024, а при k = 40 вже більше 1012.

Метод повного перебору застосовний в тих випадках, коли шукане рішення Y =f(X) належить деякій кінцевій області і може бути знайдена проста функція quality(Y) для перевірки правильності (чи якості) вибраного рішення. Тоді завдання P обчислення функції f замінюється на багатократне рішення задачі R обчислення функції quality (стільки разів, скільки елементів є в області

рішень). Причому, в загальному випадку, проглянути треба усю область і порядок, в якому проглядаються елементи, не важливий.

Евристичні методи розробки алгоритмів

Під евристичними розуміються методи, правильність яких не доведена. Вони виглядають правдоподібними, здається, що у більшості випадків вони повинні давати вірне рішення. Іноді не вдається побудувати контрприклад, що демонструє помилковість або неуніверсальність метода. Але не вдається довести математичними засобами і правильність методу. Проте, практика використання евристичних методів дає позитивні результати.

Евристичні методи різноманітні, тому не можна описати загальну схему розробки таких методів. Найчастіше евристичні методи застосовують спільно з методами перебору для скорочення кількості варіантів, що перевіряються : деякі підмножини варіантів згідно з вибраною евристикою вважаються свідомо неприйнятними і не перевіряються. Таким чином, алгоритм перебору з евристикою виконується значно швидше, ніж алгоритм повного перебору. Платою за це є відсутність гарантії правильності рішення або гарантії того, що з усіх можливих вибрано найкраще рішення.

В якості прикладу можна привести завдання розфарбовування вершин графа: розфарбувати вершини графа так, щоб суміжні вершини були розфарбовані в різні кольори, а кількість використаних в графі фарб була мінімальною.

Точне рішення задачі можливе методом повного перебору, але воно виходитиме за занадто великий час для графів, декількох десятків вершин, що містять. Проте можна помітити, що у формулюванні завдання містяться дві умови - перше - обов'язкове і друге (мінімальності), яке частенько можна ослабити. Наприклад, якщо мінімальне число фарб рівне десяти, а алгоритм швидко знайде розфарбовування, що використовує одинадцять фарб, то часто таке рішення можна розглядати як прийнятне.

Динамічне програмування

Найбільш загальною формою називають процес покрокового рішення задач, коли на кожному кроці вибирається одно значення з безлічі допустимих на цьому кроці, причому таке, яке оптимізує задану мету. У основі програмування лежить принцип оптимальності Беномена. Суть методу :

Часто не вдається розбити завдання на невелику кількість підзадач, об'єднане рішення яких дозволяє отримати рішення шуканих задач.

Виникають 2 можливості:

1. Можна спробувати розділити завдання на стільки завдань, скільки необхідно, потім кожну на ще дрібніші і так далі. Якщо алгоритм зводиться до саме такої послідовності, то отримаємо завдання з експоненціальним часом рішенням.
2. Іноді вдається отримати алгоритм з поліноміальним числом підзадач, і ту або іншу доводитися вирішувати багаторазово. Якщо би відслідковували кожну підзадачу і просто відшукували їх рішення, то можна отримали поліноміальне рішення.

Іноді простіше створити таблицю рішення усіх підзадач незалежно від того, потрібна вона або ні.

Заповнення таблиць - рішення підзадач для отримання рішення певної задачі - в теорії алгоритмів дістало назву динамічного програмування. Форми алгоритмів динамічного програмування можуть бути різними, загальними можуть бути лише заповнені таблиці і порядок заповнення її елементів.

Метод балансування

При проектуванні деяких алгоритмів доводитися йти на раз-особисті компроміси, тобто по можливості збалансувати обчислювальні витрати на використання різних частин алгоритми. Метод балансування розглядається як балансування дерев. Дерево - важлива структура даних, вживана для зберігання, обробки і представлення інформації. Дерево складається з елементів^ вершин і зв'язків між ними (дуг). Серед вершин виділяється одна, яка називається коренем. Вона є батьківською по відношенню до інших пов'язаних з нею вершин. Усі вершини, пов'язані з коренем дугами, називаються нащадками. Кожна вершина в дереві, окрім кореня, має точно одну батьківську вершину і більше нащадків. Дерево має дві властивості:

- зв’язність: з кореня треба пройти по дугах до кожної вершини;
- воно не містить циклів, тобто замкнутих послідовностей вершин і дуг.

У комп'ютерних науках часто використовуються дерева, в яких усі вершини містять обмежене число нащадків (не більше двох).

Якщо це число 2 (0,1,2), то такі дерева називаються бінарними. Якщо у вершини два нащадки, те дерево ділиться на ліве піддерево і праве. Розрізняють ідеальні і вироджені дерева.

Ідеальне дерево - дерево, в якому усі вершини розташовуються на k рівнях : корінь - на 1-му, дві вершини на 2-му, чотири - на 3-му. Нова вершина може бути поміщена тільки на k+1 рівень.

Вироджене дерево - дерево, яке є лінійним списком елементів, впорядкованих за збільшенням або зменшенням інформаційних полів.

Критерій якості дерева - це довжина найдовшої гілки. На одиницю від цього значення відрізняється кількість рівнянь дерева, тобто її висота (на одиницю менше). Дерева мінімальної висоти мають максимальну кількість вершин на одному рівні. Таким чином хороші з точки зору пошуку і вставки вершини - це дерева мінімальної висоти. Вони мають мінімальну кількість вершин на кожному рівні.

Метод балансування - метод що дозволяє не допустити занадто поганих дерев.

Суть: дерево називається збалансованим, якщо висота hl лівого піддерева і висота hr правого піддерева для кожної вершини відрізняються не більше ніж на одиницю. Кращі зі збалансованих дерев є ідеальними. При включенні нової вершини в дерево може (але не обов’язково) збільшуватися висота одного з дерев.

Можливі 3 ситуації.

1. Вершина включається в піддерево меншої висоти, його висота збільшується на одиницю, дерево стає збалансованим (ідеальним).
2. Піддерева мають однакову висоту. При включенні нової вершини, висота однієї з них збільшується на одиницю, дерево залишається збалансованим.

3. Вершина включається в піддерево більшої висоти, різниця висот стає рівною двом. Дерево стає незбалансованим. Варіанти рішення : треба гілці одного з піддерев підняти на одну одиницю і одночасно на одиницю опустити інше. Ця модифікація призводить до зміни форми дерева. Піднімаючи ліве піддерево, включаючи його корінь, тим самим, робиться рівень кореня лівого піддерева вищим, ніж корінь самого дерева. Отже, корінь лівого піддерева стає коренем усього дерева. В цьому випадку доведеться змінити деякі зв'язки між вершинами, оскільки якщо корені міняються ролями, то на зворотні міняються і відношення пращур-нащадок.

V. КЛАСИ СКЛАДНОСТІ Р І NP

5.1 Поняття складності алгоритмів

Аналіз алгоритму полягає в тому, щоб передбачити потрібні для його виконання ресурси. Іноді оцінюється потреба в таких ресурсах, як пам'ять, пропускна спроможність мережі або необхідне апаратне забезпечення, але найчастіше визначається час обчислення. Шляхом аналізу декількох алгоритмів, призначених для вирішення одного і того ж завдання, можна без зусиль вибрати найбільш ефективний. В процесі такого аналізу може також виявитися, що декілька алгоритмів приблизно рівноцінні, а усі інші треба відкинути.

Під складністю алгоритму розуміють одну з асимптотичних оцінок функції складності алгоритму (зазвичай використовується О-оцінка). Наприклад, говорять: "Складність алгоритму А є $O(n^3)$.

По складності алгоритми ділять на категорії. Приклади деяких категорій складності алгоритмів :

- алгоритми поліноміальної складності, $F(n) = O(n^p)$;
- алгоритми експоненціальної складності, $F(n) = O(e^{a\,n})$;
- алгоритми факторіальної складності, $F(n) = O(n!)$.

Алгоритми, що мають поліноміальну функцію складності, математики називають ефективними.

Метою аналізу трудомісткості алгоритмів є знаходження оптимального алгоритму для вирішення цього завдання. В якості критерію оптимальності алгоритму вибирається трудомісткість алгоритму, що розуміється як кількість елементарних операцій, які необхідно виконати для вирішення завдання за допомогою цього алгоритму. Функцією трудомісткості називається відношення, що зв'язують вхідні дані алгоритму з кількістю елементарних операцій.

Трудомісткість алгоритмів по-різному залежить від вхідних даних. Для деяких алгоритмів трудомісткість залежить тільки від об'єму даних, для інших алгоритмів — від значень даних, в деяких випадках порядок вступу даних може впливати на трудомісткість. Трудомісткість багатьох алгоритмів може в тій чи іншій мірі залежати від усіх перелічених вище чинників.

Одним із спрощених видів аналізу, використовуваних на практиці, є асимптотичний аналіз алгоритмів. Метою асимптотичного аналізу є порівняння витрат часу і інших ресурсів різними алгоритмами, призначеними для вирішення одного і того ж завдання, при великих об'ємах вхідних даних. Використовувана в асимптотичному аналізі оцінка функції трудомісткості, звана складністю алгоритму, дозволяє визначити, як швидко росте трудомісткість алгоритму зі збільшенням об'єму даних. У асимптотичному аналізі алгоритмів використовуються позначення, прийняті в математичному асимптотичному аналізі. Нижче перераховані основні оцінки складності.

Основною оцінкою функції складності алгоритму $f(n)$ є оцінка Θ. Тут n — величина об'єму даних або довжина входу. Ми говоримо, що оцінка складності

алгоритму f(n)=Θ(g(n)), якщо при g>0 при n>0 існують позитивні c_1 c_2, n_0, такі, що $c_1g(n) \leq f(n) \leq c_2g(n)$ при $n>n_0$, інакше кажучи, можна знайти такі c_1 і c_2, що при досить великих n, f(n) буде між $c_1g(n)$ і $c_2g(n)$.

У такому випадку говорять ще, що функція g(n) є асимптотично точною оцінкою функції f(n), оскільки за визначенням функція f(n) не відрізняється від функції g(n) з точністю до постійного множника. Наприклад, для методу сортування heapsort оцінка трудомісткості складає

$f(n) = \Theta(n \log n)$ то есть $g(n) = n\log n$

Из $f(n) = \Theta(g(n))$ следует, что $g(n) = \Theta(f(n))$.

Важливо розуміти, що Θ(g(n)) є не функцією, а множиною функцій, що описують ріст f(n) з точністю до постійного множника.

Θ дає одночасно верхню і нижню оцінки росту функції. Часто необхідно розглядати ці оцінки окремо. Оцінка *O* представляет собою верхню асимптотичну оцінку трудомісткості алгоритму. Ми говоримо, що f(n)=O(g(n)) якщо $\exists\, c > 0, n_0 > 0 : 0 \leq f(n) \leq cg(n), \forall\, n > n_0$

Інакше кажучи, запис f(n)=O(g(n)) означає, що f(n) належить класу функцій, які ростуть не швидше, ніж функція g(n) з точністю до постійного множника.

Оцінка Ω задає нижню асимптотичну оцінку росту функції f(n) і визначає клас функцій, які ростуть не повільніше, ніж g(n) з точністю до постійного множника f(n)=Ω(g(n)). якщо $\exists\, c > 0, n_0 > 0 : 0 \leq cg(n) \leq f(n), \forall\, n > n_0$

Наприклад, запис *f(n)=O(n* log n*)* означає клас функцій, які ростуть не повільніше, ніж *g(n)=n log n*, в цей клас потрапляють усі поліноми з мірою більшої одиниці, так само як і усі степенні функції з основою більшою одиниці. Рівність f(n)=Θ(g(n)) виконується тоді і тільки тоді, коли f(n)=O(g(n)) і f(n)=Ω(g(n)).

Асимптотичний аналіз алгоритмів має не лише практичне, але і теоретичне значення. Так, наприклад, доведено, що усі алгоритми сортування, засновані на попарному порівнянні елементів, відсортують n елементів за час, не менший Ω(n log n).

Класи складності

У рамках класичної теорії здійснюється класифікація завдач за класами складності (P -складні, NP -складні, експоненціально складні та ін.). До класу P відносяться задачі, які можуть бути вирішені за час, поліноміальний залежний від об'єму початкових даних, за допомогою детермінованої обчислювальної машини (наприклад, машини Т'юринга), а до класу NP — задачі, які можуть бути вирішені за поліноміальний виражений час за допомогою недетермінованої обчислювальної машини, тобто машини, наступний стан якої не завжди однозначно визначається попередніми. Роботу такої машини можна представити як процес, що розгалужується на кожній неоднозначності: задача вважається вирішеною, якщо хоч би одна гілка процесу прийшла до відповіді. Інше визначення класу NP: до класу NP відносяться задачі, рішення яких за допомогою додаткової інформації поліноміальної довжини, ми можемо перевірити за поліноміальний час. Зокрема, до класу NP відносяться усі задачі,

рішення яких можна *перевірити* за поліноміальний час. Клас P міститься в класі NP. Класичним прикладом NP-задачі є завдання про комівояжера.

Оскільки клас P міститься в класі NP, приналежність задачі до класу NP часто відбиває наше поточне уявлення про способи рішення цієї задачі і носить неостаточний характер. У загальному випадку немає підстав вважати, що для NP-задачі не може бути знайдене P-рішення. Питання про можливу еквівалентність класів P і NP (тобто про можливість знаходження P-рішення для будь-якої NP-задачі) вважається одним з основних питань сучасної теорії складності алгоритмів. Відповідь на це питання не знайдена досі. Сама постановка питання про еквівалентність класів P і NP можлива завдяки введенню поняття NP-повних задач. NP-повні задачі складають підмножину NP-задач і відрізняються тією властивістю, що усі NP-задачі можуть бути тим або іншим способом зведені до них. З цього виходить, що якщо для NP-повної задачі буде знайдено P-рішення, то P–рішення буде знайдене для усіх завдань класу NP. Прикладом NP-повної задачі є задача про кон'юнктивну форму.

Дослідження складності алгоритмів дозволили по-новому поглянути на рішення багатьох класичних математичних завдань і знайти для ряду таких завдань (множення многочленів і матриць, рішення лінійних систем рівнянь та ін.) рішення, що вимагають менше ресурсів, ніж традиційні.

Складність завдання і поліноміального сходження

Складнощі завдання характеризують складністю найкращого алгоритму, що вирішує задачу з найважчим значенням входу.

Розрізняють 3 основні класи складності задач :

1) задачі, для яких відомі алгоритми поліноміальної складності;

2) задачі, для яких не відомі алгоритми поліноміальної складності, але для яких і не доведено неіснування таких алгоритмів;

3) задачі, для яких встановлено, що вони не можуть бути вирішені алгоритмом поліноміальної складності.

Задачі, які не належать першому класу називають важко-розв’язуваними.

Другий клас задач є найзагадковішим. Для більшості задач цього класу справедливо наступне: існування поліноміального алгоритму для однієї з них означає існування поліноміального алгоритму для усіх інших.

Великою різноманітністю відносної складності відрізняються задачі дискретної математики. Алгоритми рішення багатьох з них мають переборний характер, що часто призводить до важко розв’язаності задач. Нерідкі випадки, коли така задача може бути вирішена тільки шляхом застосування алгоритму повного перебору. У такому разі складність задачі визначається тим, наскільки швидко росте безліч варіантів зі збільшенням розміру вхідних даних.

Сходженість задач

Говорять, що задача Z1 поліноміальний зводиться до задачі Z2, якщо рішення задачі Z1 можна отримати з рішення відповідної задачі Z2 за поліноміальний час.

Відмітимо, що якщо задача Z2 має поліноміальну складність і поліноміальне сходження задачі Z1 до задачі Z2 доведена, то тим самим доведена поліноміальна складність задачі Z1.

Класи задач P і NP

Клас P є безліччю задач, для кожної з яких існує детермінований алгоритм з поліноміальною функцією складності.

Клас NP визначається як безліч задач, які можуть бути вирішені недетермінованим алгоритмом з поліноміальною функцією складності.

Очевидно, що P$\subseteq$NP.

NP-складні і NP-повні завдання

Задача називається NP-складною, якщо будь-яка задача з NP поліноміальних зводиться до неї.

Завдання називається NP-повною, якщо вона є NP-складною і в той же час належить до класу NP.

Таким чином, мають місце наступні співвідношення:

{NP-повні} $\subseteq$ {NP-складні}
складність(NP-повна)≤складність(NP-складна).

Приклад NP-складної задачі: знайти оптимальний маршрут в симетричному завданні комівояжера (тобто з симетричною матрицею вартості).

Приклади NP-повних задач:

- завдання про здійснимість булевого виразу, представленого в КНФ;
- визначити, чи містить заданий граф повний підграф з k –вершинами.

5.2 Правила аналізу складності алгоритмів

У загальному випадку час виконання оператора або групи операторів можна розглядати як функцію з параметрами – розміром вхідних даних і/або одної чи декількох змінних. Але для часу виконання програми в цілому допустимим параметром може бути лише розмір вхідних даних.

Час виконання операторів присвоєння, читання і запису звичайно має порядок $O(1)$.

Час виконання послідовності операторів визначається за правилом сум. Тому міра росту часу виконання послідовності операторів без визначення констант пропорційності співпадає з найбільшим часом виконання оператора в даній послідовності.

Час виконання умовних операторів складається з часу виконання умовно виконуваних операторів і часу обчислення самого логічного виразу. Час обчислення логічного виразу часто має порядок $O(1)$. Час для всієї конструкції if-then-else складається з часу обчислення логічного виразу і найбільшого з часів, який необхідний для виконання операторів, що виконуються при різних значеннях логічного виразу.

Час виконання циклу є сумою часів усіх часів виконуваних конструкцій циклу, які в свою чергу складаються з часів виконання операторів тіла циклу і часу обчислення умови завершення циклу (часто має порядок $O(1)$). Часто час виконання циклу обчислюється, нехтуючи визначенням констант

пропорційності, як добуток кількості виконуваних операцій циклу на найбільший можливий час виконання тіла циклу. Час виконання кожного циклу, якщо в програмі їх декілька, повинен визначатися окремо.

VI. ЛАБОРАТОРНІ РОБОТИ

6.1 Лабораторна робота – «Математичні основи аналізу алгоритмів»

Мета: усвідомлення та раціональне використання понять, законів та методів математичної логіки як засобу для побудови та аналізу алгоритмів та формальних систем.

Теоретичні відомості.

Під **висловленням** розумітимемо речення, про зміст якого можна сказати: істинний він чи хибний, і притому лише одне з двох. Звичайно, це не означення. Поняття висловлення є в логіці висловлень вихідним, неозначуваним.

Саме ця властивість — бути істинним чи хибним — є характеристичною для висловлення як предмета вивчення логіки. Поняття істинності і хибності в логіці висловлень не аналізуються, а беруться як дані. Наприклад, “7 — просте число” є висловлення істинне, “ТНЕУ — не вищий навчальний заклад” — висловлення хибне.

Розглянемо вираз “*x* більше від одиниці”. Цей вираз не е висловленням, бо немає смислу твердити про його істинність чи хибність доти, поки символ “*x*” не буде замінено назвою певного дійсного числа.

Визначення 1 Вираз, який не є висловленням, але стає ним після заміни всіх символів змінних, що входять до цього виразу, назвами відповідних предметів, називають ***висловлювальною формою*** або ***невизначеним висловленням***.

Розглядаючи висловлення, виходять з двох основних *припущень*:

а) кожне висловлення є або істинним, або хибним, тобто третього не дано (***закон виключеного третього***);

б) жодне висловлення не є одночасно істинним і хибним (***закон виключення суперечності***).

Позначимо значення “істинне” та “хибне” відповідно через “1” та “0”. Звичайно, “1” і “0” тут не є назвами чисел, а лише символами значень введеної ***функції істинності***. Значення “1” і “0” називають ***значеннями істинності*** чи ***істинісними значеннями***.

Висловлювальні змінні позначають так само, як числові змінні в математиці: $p, q, r, p_1, p_2, p_3, \ldots$ Замість цих символів можна підставляти довільні висловлення. Звичайно, символи $p, q, r, p_1, p_2, p_3, \ldots$ не є висловленнями, вони є ***змінними для висловлень*** (їх також називають ***змінними висловленнями*** або ***пропозиційними буквами*** чи ***пропозиційними змінними***).

Значення функції істинності для даного значення аргументу p позначатимемо $| p|$. Так, позначивши через p висловлення “2 — найменше

просте число”, а через q — висловлення: “Число π дорівнює 3,14”, матимемо: $|p|= 1$, $|q|=0$.

Операції над висловленнями

Однією з основних задач логіки висловлень є дослідження операцій, за допомогою яких з певних вихідних висловлень утворюють нові висловлення. Такі операції і називають ***логічними***.

Означення кожної логічної операції задаватимемо відповідною матрицею (таблицею), в перших стовпчиках якої записуються всі можливі істинісні значення компонентів, а в останньому стовпчику — істинісне значення результату операції.

Визначення 2 Бінарну логічну операцію, яка відповідає зв’язці “і” звичайної мови, позначається символом “$\wedge$” і задається наступною матрицею, називають ***кон'юнкцією***, або ***логічним множенням.***

p	q	$p \wedge q$
0	0	0
0	1	0
1	0	0
1	1	1

Вищенаведене табличне означення операції “кон'юнкція” рівнозначне такому словесному означенню: “Кон'юнкція $p \wedge q$ істинна тоді і тільки тоді, коли обидва компоненти p і q є одночасно істинними”.

Визначення 3 Бінарну логічну операцію, відповідну зв’язці “або нероздільне”, що позначається символом “$\vee$” і задається наступною таблицею, називають ***диз’юнкцією*** або ***логічним додаванням***.

p	q	$p \vee q$
0	0	0
0	1	1
1	0	1
1	1	1

Словесне означення диз'юнкції: “Диз’юнкція “$p \vee q$” істинна тоді і тільки тоді, коли принаймні один з її компонентів p або q є істинним, у протилежному випадку диз’юнкція є хибною”.

Визначення 4 Унарну операція, відповідну виразу “неправильно, що”, яку позначають символом “—“ і задають наступною таблицею, називають ***логічним запереченням***. Вираз „$\overline{p}$” читають “не p”.

p	$\overline{p}$
0	1
1	0

Словесне означення заперечення: “Заперечення $\overline{p}$ істинне тоді і тільки тоді, коли p — хибне, у протилежному випадку $\overline{p}$ — хибне”.

Визначення 5 Операцію, яка відповідає сполучнику “якщо..., то...”, позначається символом “$\rightarrow$”, називають ***імплікацією***. Її задають таблицею:

p	q	$p\rightarrow q$
0	0	1
0	1	1
1	0	0
1	1	1

В імплікації p називають ***антецедентом*** або ***посиланням (умовою)***, q — ***консеквентом*** або ***висновком***.

Словесне означення операції “імплікація” таке: “Імплікація „$p\rightarrow q$” хибна тоді і тільки тоді, коли її антецедент — істинний, а консеквент — хибний, в усіх інших випадках імплікація — істинна”.

Визначення 6 Бінарну логічну операцію, яка відповідає зв’язці “тоді і тільки тоді”, позначається символом “$\leftrightarrow$”, називають ***еквіваленцією***. Її табличне означення:

p	q	$p\leftrightarrow q$
0	0	1
0	1	0
1	0	0
1	1	1

Словесне означення еквіваленції можна сформулювати так: “Еквіваленція “$p\leftrightarrow q$” істинна тоді і тільки тоді, коли p і q набувають однакових значень істинності, в протилежному випадку еквіваленція — хибна”.

Висловлення, які не містять логічних зв'язок, називають ***елементарними*** чи ***атомарними***. Наприклад, висловлення “Рейк’явік — столиця Ісландії” і “3 — просте число” — елементарні висловлювання. Висловлення, яке містить хоча б одну логічну зв’язку, називають ***складним***. Наприклад, висловлення “10 не є простим числом”.

Пропозиційні літери, символи логічних операцій та дужки – це ***вихідні символи*** алгебри висловлень.

Довільну послідовність вихідних символів називають ***виразом мови***.

З множини виразів виділяють підмножину формул.

Формулами алгебри висловлень називають пропозиційні літери і вирази виду

$$\overline{F},\ F\wedge G,\ F\vee G,\ F\rightarrow G,\ F\leftrightarrow G,$$

де F,G – формули.

При цьому пропозиційні літери є елементарними (атомарними) формулами. Наприклад, $(((p\rightarrow(\overline{q}))\vee r)\oplus s)$ – формула.

Дужки у формулі означають порядок виконання операції.

Символу кожної логічної операції відповідає пара дужок. Щоб запобігти громіздкості формул, використовують такі правила скорочення:

1) зовнішні дужки у записі кінцевої формули можна опустити;

2) всім логічним операціям приписують відповідний ранг, який знижується зліва на право: $\overline{}$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$.

Лівіша операція є сильнішою за правішу.

Знаючи ранг операції можна не використовувати дужки.

Таблиці істинності

У таблиці істинності формули $f(p,q,r)$ кожному символу логічної операції в формулі $f(p,q,r)$ відповідає окремий стовпчик таблиці, останній стовпчик відповідає істинісному значенню, яке визначається даною формулою (її головною операцією). Звернемо увагу на те, що кожен стовпчик таблиці істинності для формули $f(p,q,r)$ відповідає певному кроку процесу її побудови або, як кажуть, певній підформулі $f(p,q,r)$.

Наприклад, нехай задано функцію $f(p,q,r)=(\overline{p}\rightarrow q \vee r)\wedge(q\rightarrow p\vee \overline{r})$.

p	q	r	$\overline{p}$	$q\vee r$	$\overline{p}\rightarrow q\vee r$	$\overline{r}$	$p\vee \overline{r}$	$q\rightarrow p\vee \overline{r}$	$f(p,q,r)$
0	0	0	1	0	0	1	1	1	0
0	0	1	1	1	1	0	0	1	1
0	1	0	1	1	1	1	1	1	1
0	1	1	1	1	1	0	0	0	0
1	0	0	0	0	1	1	1	1	1
1	0	1	0	1	1	0	1	1	1
1	1	0	0	1	1	1	1	1	1
1	1	1	0	1	1	0	1	1	1

Часто таблицею істинності формули $f(p_1, ..., p_n)$ називають скорочену таблицю, в якій з вищенаведеної залишають перших n стовпчиків (значень аргументів $p_1, ..., p_n$) і останній стовпчик.

Степінь складності таблиці істинності для формули f швидко зростає із збільшенням кількості різних пропозиційних букв, що входять до f. Так, при n = 3 кількість рядків таблиці дорівнює $2^3 = 8$, при $n = 4$ воно становить $2^4 = 16$, при $n = 5$ дорівнює $2^5 = 32$, а при $n = 10$ — вже 1024. Практично побудувати таблицю істинності в останньому випадку вже неможливо.

Означення 7 Формули алгебри висловлень $f(p_1, ..., p_n)$, які на всіх наборах $(p_1, …, p_n)$, тобто при всіх можливих розподілах істинісних значень пропозиційних літер $p_1, ..., p_n$, набувають значення 1 (останній стовбець таблиці істинності – лише 1), називають ***тавтологіями***, ***тотожно істинними формулами*** або ***законами алгебри висловлень***.

Те, що формула алгебри висловлень f є тавтологією, позначають так: $\models f$.

Означення 8 Формулу алгебри висловлень $f\ (p_1\ ,\ \ldots,\ p_n)$, яка набуває значення істинності 0 на всіх 2^n наборах, називають ***суперечністю***. Найпростішим прикладом суперечності є формула $p\wedge\bar{p}$.

Означення 9 Формулу алгебри висловлень, яка не є ні тавтологією, ні суперечністю, називають ***нейтральною***. Прикладом нейтральної формули є $p\to q$.

Множини тавтологій, суперечностей і нейтральних формул попарно не перетинаються і разом становлять множину всіх формул алгебри висловлень.

Означення 10 Формулу алгебри висловлень, яка не є суперечністю, називають ***виконуваною***.

Так, формула $p\to p$ — виконувана і формула $p\to\bar{p}$ теж виконувана при $|\mathrm{p}| = 0$; $|\mathrm{p}\to\bar{p}|=1$.

Означення 11 Висловлення називають ***логічно істинним*** (на базі алгебри висловлень) тоді і тільки тоді, коли його логічна структура є тавтологією.

Прикладом логічно істинного твердження є “Трикутник *ABC* — рівнобедрений або трикутник *ABC* — не рівнобедрений” (логічна структура цього твердження — $p\vee\bar{p}$).

Означення 12 Формули алгебри висловлень $f\ (p_1\ ,\ \ldots,\ p_n)$ і $g\ (\ p_1\ ,\ \ldots,\ p_n)$ називають ***рівносильними*** або ***логічно еквівалентними*,** якщо їх функції істинності $|f|$ і $|g|$ тотожно рівні, тобто, якщо їх значення на всіх 2^n наборах збігаються.

Рівносильність формул f і g позначатимемо $f\equiv g$ Символ “$\equiv$” не є символом операції алгебри висловлень, а означає певне відношення між формулами.

Основні рівносильності (закони) алгебри висловлень:

23.$p\wedge q\equiv q\wedge p$ – комутативність кон’юнкції.

24.$p\vee q\equiv q\vee p$ – комутативність диз’юнкції.

25.$p\wedge(q\wedge r)\equiv(p\wedge q)\wedge r$ – асоціативність кон’юнкції.

26.$p\vee(q\vee r)\equiv(p\vee q)\vee r$ – асоціативність диз’юнкції.

27.$p\wedge(q\vee r)\equiv p\wedge q\vee p\wedge r$ – перший дистрибутивний закон.

28.$p\vee(q\wedge r)\equiv(p\vee q)\wedge(p\vee r)$ – другий дистрибутивний закон.

29.$\overline{(p\wedge q)}\equiv\ \bar{p}\vee\ \bar{q}$ } закони
30.$\overline{(p\vee q)}\equiv\bar{p}\wedge\bar{q}$ } де Моргана.

31.$p\to q\equiv\bar{p}\vee q$.

32.$p\leftrightarrow q\equiv(p\to q)\wedge(q\to p)$.

33. $\bar{\bar{p}}\equiv p$ – закон подвійного заперечення.

34.$p\wedge p\equiv p$. } закони
35.$p\vee p\equiv p$. } ідемподентності.

36.$p\to q\equiv\ \bar{q}\to\bar{p}$.

37. $p\vee p\wedge q\equiv p$. } закони
38.$p\wedge(p\vee q)\equiv p$. } поглинання.

39.$\mathrm{u}\wedge1\equiv\mathrm{u}$. } правила
40.$\mathrm{u}\wedge0\equiv0$. } співвідно-
41.$\mathrm{u}\vee1\equiv1$. } шення
42.$\mathrm{u}\vee0\equiv\mathrm{u}$. } констант

43. $p \vee \bar{p} \equiv 1$ – закон виключення третього.

44. $p \wedge \bar{p} \equiv 0$ – закон протиріччя.

Алгоритм побудови формули за таблицею істинності

4. Виділити ті рядки таблиці істинності, в останніх стовпчиках яких міститься величина функції 1.
5. Виписати для кожного такого виділеного рядка кон'юнкцію (логічне «і») таким чином: якщо величина змінної цьому рядку дорівнює 1, то в кон'юнкцію записати позначення цієї змінної, інакше — її заперечення.
6. Всі отримані на попередньому кроці кон'юнкції записати елементами диз'юнкції (логічного «або»).

Приклад скорочення логічних виразів (спрощення логічних формул):

$$X \wedge \overline{Y} \wedge Z \vee X \wedge \overline{Y \wedge Z} \vee X \wedge Y \wedge Z \vee X \wedge \overline{Y} = X \wedge Z \wedge (\overline{Y} \vee Y) \vee X \wedge (\overline{Y \wedge Z} \vee \overline{Y}) =$$

$$X \wedge Z \vee X \wedge (\overline{Y} \vee \overline{Z} \vee \overline{Y}) = X \wedge (Z \vee \overline{Y} \vee \overline{Z}) = X \wedge (1 \vee \overline{Y}) = X \wedge 1 = X .$$

<u>Завдання на лабораторну роботу</u>

1. Скласти формулу математичної логіки з 4 змінними $f(p, q, r, s)$ та 8 різними логічними операціями та за допомогою таблиць істинності визначити її тип (тавтологія, суперечність, нейтральна, виконувана).
2. Використовуючи алгоритм побудови формули за таблицею істинності побудувати формулу та спростити її для функції свого варіанту.
3. Оформити звіт виконання лабораторної роботи.

ВАРІАНТИ ЗАВДАНЬ

p	q	r	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}	f_{16}	f_{17}	f_{18}	f_{19}	f_{20}	f_{21}	f_{22}	f_{23}	f_{24}	f_{25}	f_{26}	f_{27}	f_{28}	f_{29}	f_{30}
0	0	0	0	1	1	0	0	0	0	1	1	0	0	1	1	1	0	1	0	0	0	0	1	0	1	1	0	1	1	0	1	0
0	0	1	1	0	0	1	1	1	0	1	0	0	1	0	0	0	0	1	0	0	0	1	0	0	0	0	0	1	1	0	1	0
0	1	0	1	1	0	1	1	0	1	0	0	1	1	1	0	0	1	1	1	0	0	0	1	1	1	1	0	0	0	1	0	0
0	1	1	0	1	1	0	0	1	0	1	0	1	1	1	1	0	0	1	1	1	0	1	0	0	0	0	1	0	0	1	0	1
1	0	0	0	0	1	1	0	0	1	0	1	0	1	1	1	1	0	0	1	1	1	0	1	1	1	1	1	0	1	0	0	0
1	0	1	1	0	0	1	1	1	1	1	0	1	0	0	1	1	1	0	1	1	1	1	0	0	0	1	0	0	0	0	1	1
1	1	0	0	1	0	0	1	1	1	0	1	0	0	0	0	1	1	0	0	1	1	0	1	1	0	0	1	1	1	1	1	1
1	1	1	1	0	1	0	0	0	0	0	1	1	0	0	0	0	1	0	0	0	1	1	0	1	1	0	1	1	0	1	0	1

3. Скориставшись законами алгебри логіки, максимально спростити вираз свого варіанту:
   1) $\left(x \vee \left(\bar{k} \wedge y\right)\right) \wedge \left(\left(\bar{x} \wedge \left(\bar{y} \vee k\right)\right) \vee z\right) \vee \bar{z} \vee \left(x \vee \left(y \wedge \bar{k}\right)\right)$;
   2) $\left(\left(x \vee z\right) \wedge \left(x \vee k\right)\right) \wedge \left(\left(\left(z \vee \left(z \wedge y\right)\right) \wedge \bar{z}\right) \vee \bar{x}\right)$;
   3) $\left(\bar{y} \vee k\right) \wedge \left(\left(\bar{k} \wedge z\right) \vee \left(x \wedge z\right) \vee \left(\bar{k} \wedge \bar{z}\right) \vee \left(x \wedge \bar{z}\right)\right) \wedge \left(y \vee k\right)$;
   4) $\left(x \vee \bar{z}\right) \wedge \left(\bar{x} \vee \bar{y}\right) \wedge \left(\bar{y} \vee z\right) \wedge \left(\bar{x} \vee y\right) \wedge \left(y \vee z\right)$;
   5) $\left(x \wedge z\right) \vee \left(\left(y \vee \bar{k}\right) \wedge \left(\bar{x} \vee \bar{k}\right) \wedge \left(\bar{x} \vee k\right) \wedge \left(k \vee y\right)\right) \vee \left(x \wedge \bar{z}\right)$;
   6) $\left(\left(\bar{y} \vee \bar{z}\right) \wedge \left(x \vee y\right)\right) \vee \left(k \wedge \bar{z}\right) \vee \left(\left(\left(\bar{y} \wedge \bar{x}\right) \vee z\right) \wedge \left(x \vee y\right)\right)$;
   7) $\overline{X \wedge Y} \vee \overline{X} \wedge Y \wedge Z \wedge (\overline{X} \vee \overline{X \wedge Y} \vee \overline{Y})$;

8) $\overline{\overline{\overline{X}} \wedge \overline{\overline{Y}}} \vee \overline{X} \wedge (X \vee \overline{\overline{\overline{X}} \vee Y}) \vee \overline{\overline{\overline{X}} \vee Y \vee Z \wedge X \wedge \overline{\overline{Y}}}$;

9) $\overline{\overline{\overline{X}} \wedge \overline{\overline{Y}}} \vee \overline{X} \wedge \overline{X \vee \overline{\overline{\overline{X}} \vee Y}}$;

10) $\overline{X} \wedge \overline{Y} \wedge \overline{Z} \vee \overline{X} \wedge \overline{Y} \wedge Z \vee \overline{X} \wedge Y \wedge \overline{Z} \vee X \wedge \overline{Y} \wedge Z \vee X \wedge Y \wedge \overline{Z}$;

11) $(X \vee Y) \wedge Z \vee X \wedge Y \wedge Z \wedge (\overline{Z} \vee X \wedge Y) \vee \overline{Z} \wedge X \wedge Y \vee \overline{X} \wedge Y \wedge \overline{Z}$;

12) $X \wedge \overline{Y} \wedge Z \vee X \wedge \overline{Y \wedge Z} \vee X \wedge Y \wedge Z \vee X \wedge \overline{Y}$;

13) $\overline{X \wedge Y} \vee \overline{X} \wedge Y \wedge Z \wedge (\overline{X} \vee \overline{X \wedge Y} \vee \overline{Y})$;

14) $(\overline{A} \vee B) \wedge (\overline{B} \vee C \vee \overline{A}) \wedge (\overline{\overline{D}} \vee A \wedge \overline{C}) \wedge (\overline{D} \vee A)$;

15) $(A \rightarrow B \wedge C) \wedge (C \rightarrow B \wedge A) \wedge (B \rightarrow C \wedge A)$;

16) $\left(\overline{a \vee \overline{b} \wedge b \wedge (a \rightarrow b)}\right) \vee \left(\overline{a \rightarrow b} \rightarrow \overline{b}\right)$;

17) $(a \wedge b \leftrightarrow \overline{a}) \wedge \left((\overline{a} \wedge \overline{b}) \vee b\right)$;

18) $\left(a \wedge \overline{\overline{\overline{b}} \rightarrow a \vee \overline{\overline{b}}}\right) \wedge \left(a \wedge (\overline{a} \vee b)\right)$;

19) $\left(\overline{\overline{\overline{a}} \rightarrow \overline{\overline{b}}} \vee \overline{\overline{\overline{b}} \wedge a}\right) \vee \left((\overline{a} \wedge b) \vee b\right)$;

20) $\left(a \wedge \overline{(b \vee \overline{c})} \rightarrow a\right) \wedge \left(\overline{a} \wedge (\overline{a} \vee \overline{b})\right)$;

21) $\left(A \wedge (\overline{B \wedge C \vee \overline{\overline{A}}})\right) \wedge \left(A \wedge B \vee \overline{B} \vee \overline{A} \wedge C\right)$;

22) $\left(A \wedge \overline{B} \wedge C \vee A \vee B \vee \overline{C}\right) \vee \left(\overline{A} \vee C \vee \overline{A \wedge B}\right)$;

23) $\left(\overline{\overline{\overline{x}} \wedge \overline{\overline{y}}} \vee \overline{x}\right) \wedge \overline{\overline{(\overline{\overline{x}} \wedge y)} \vee x}$;

24) $\left(\overline{A \rightarrow B} \rightarrow A \vee B\right) \vee \left(A \vee B \leftrightarrow \overline{B}\right)$;

25) $\overline{A} \vee \overline{B} \vee \overline{\overline{\overline{A \vee B}} \leftrightarrow A}$;

26) $\overline{A} \vee B \rightarrow \overline{\overline{\overline{A}} \vee B} \leftrightarrow \overline{A}$;

27) $\overline{A} \rightarrow B \leftrightarrow \overline{A} \vee B \rightarrow A$;

28) $\overline{A \vee B} \wedge \overline{B \vee \overline{\overline{A}}} \leftrightarrow \overline{A}$;

29) $A \wedge B \rightarrow \overline{A \leftrightarrow B}$;

30) $\left(A \wedge \overline{A \leftrightarrow B} \vee A\right) \wedge \left(\overline{A \rightarrow B} \vee A \wedge \overline{B}\right)$.

6.2 Лабораторна робота – «Складність алгоритмів»

Мета: навчитися визначати складність алгоримтів.

Теоретичні відомості.

У процесі вирішення прикладних задач вибір потрібного алгоритму викликає певні труднощі. І справді, на чому базувати свій вибір, якщо алгоритм повинен задовольняти наступні протиріччя.

1. Бути простим для розуміння, перекладу в програмний код і наладки.
2. Ефективно використовувати комп’ютерні ресурси і виконуватися швидко.

Якщо написана програма повинна виконуватися лише декілька разів, то перша вимога найбільш важлива. Вартість робочого часу програміста, звичайно, значно перевищує вартість машинного часу виконання програми,

тому вартість програми оптимізується за вартістю написання (а не виконання) програми. Якщо мати справу з задачею, вирішення якої потребує значних обчислювальних затрат, то вартість виконання програми може перевищити вартість написання програми, особливо якщо програма повинна виконуватися багаторазово. Тому, з економічної точки зору, перевагу буде мати складний комплексний алгоритм (в надії, що результуюча програма буде виконуватися суттєво швидше, ніж більш проста програма). Але і в цій ситуації розумніше спочатку реалізувати простий алгоритм, щоб визначити, як повинна себе вести більш складна програма. При побудові складної програмної системи бажано реалізувати її простий прототип, на якому можна провести необхідні виміри й змоделювати її поведінку в цілому, перш ніж приступати до розробки кінцевого варіанту. Таким чином, програмісти повинні бути обізнані не тільки з методами побудови швидких алгоритмів, але й знати, коли їх потрібно застосувати.

Існує декілька способів оцінки складності алгоритмів. Програмісти, звичайно, зосереджують увагу на швидкості алгоритму, але важливі й інші вимоги, наприклад, до розмірів пам'яті, вільного місця на диску або інших ресурсів. Від швидкого алгоритму може бути мало толку, якщо під нього буде потрібно більше пам'яті, ніж встановлено на комп'ютері.

Важливо розрізняти практичну складність, яка є точною мірою часу обчислення і об'єму пам'яті для конкретної моделі обчислювальної машини, і теоретичну складність, яка більш незалежна від практичних умов виконання алгоритму і дає порядок величини вартості.

Більшість алгоритмів надає вибір між швидкістю виконання і ресурсами. Задача може виконуватися швидше, використовуючи більше пам'яті, або навпаки – повільніше з меншим обсягом пам'яті.

Прикладом в даному випадку може слугувати алгоритм знаходження найкоротшого шляху. Задавши карту вулиць міста у вигляді мережі, можна написати алгоритм, що обчислює найкоротшу відстань між будь-якими двома точками цієї мережі. Замість того, щоб кожного разу заново перераховувати найкоротшу відстань між двома заданими точками, можна наперед прорахувати її для всіх пар точка і зберегти результати в таблиці. Тоді, щоб знайти найкоротшу відстань для двох заданих точка, достатньо буде просто взяти готові значення з таблиці. При цьому отримують результат, практично, миттєво, але це зажадає великий обсяг пам'яті. Карта вулиць для великого міста може містити сотні тисяч точок. Для такої мережі таблиця найкоротших відстаней містила б більше 10 мільярдів записів. В цьому випадку вибір між часом виконання і обсягом необхідної пам'яті очевидний.

Із цього зв'язку випливає ідея просторово-часової складності алгоритмів. При цьому підході складність алгоритму оцінюється в термінах часу і простору, і знаходиться компроміс між ними.

При порівнянні різних алгоритмів важливо розуміти, як складність алгоритму співвідноситься із складністю вирішуваної задачі. При розрахунках за одним алгоритмом сортування тисячі чисел може зайняти 1 секунду, а сортування мільйона – 10 секунд, тоді як розрахунки за іншими алгоритмами можуть зайняти 2 і 5 секунд відповідно. У цьому випадку не можна однозначно сказати, яка із двох програм краща – це буде залежати від вихідних даних.

Ефективність алгоритмів

Одним із способів визначення часової ефективності алгоритмів полягає в наступному: на основі даного алгоритму потрібно написати програму і виміряти час її виконання на певному комп’ютері для вибраної множини вхідних даних. Хоча такий спосіб популярний і, безумовно, корисний, він породжує певні проблеми. Визначений час виконання програми залежить не тільки від використаного алгоритму, але й від архітектури і набору внутрішніх команд даного комп’ютера, від якості компілятора, і від рівня програміста, який реалізував даний алгоритм. Час виконання також може суттєво залежати від вибраної множини тестових вхідних даних. Ця залежність стає очевидною при реалізації одного й того ж алгоритму з використанням різних комп’ютерів, різних компіляторів, при залучені програмістів різного рівня і при використанні різних тестових даних. Щоб підвищити об’єктивність оцінки алгоритмів, учені, які працюють в галузі комп’ютерних наук, прийняли *асимптотичну часову складність* як основну міру ефективності виконання алгоритму.

Часто говорять, що час виконання алгоритму має порядок $T(N)$ від вхідних даних розміру N. Одиниця вимірювання $T(N)$ точно не визначена, але в більшості випадків розуміють під нею кількість інструкцій, які виконуються на ідеалізованому комп’ютері.

Для багатьох програм час виконання дійсно є функцією вхідних даних, а не їх розміру. У цьому випадку визначають $T(N)$ як час виконання в найгіршому випадку, тобто, як максимум часів виконання за всіма вхідними даними розміру N. Поряд з тим розглядають $T_{ср}(N)$ як середній (в статистичному розумінні) час виконання за всіма вхідними даними розміру N. Хоча $T_{ср}(N)$ є достатньо об’єктивною мірою виконання, але часто неможливо передбачити, або обґрунтувати, рівнозначність усіх вхідних даних. На практиці середній час виконання знайти складніше, ніж найгірший час виконання, так як математично це зробити важко і, крім цього, часто не буває простого визначення поняття „середніх” вхідних даних. Тому, в основному, користуються найгіршим часом виконання як міра часової складності алгоритмів.

Продуктивність алгоритму оцінюють за порядком величини. Говорять, що алгоритм має складність порядку $O\left(f\left(N\right)\right)$, якщо час виконання алгоритму росте пропорційно функції $f\left(N\right)$ із збільшенням розмірності початкових даних N. O – позначає „величина порядку”.

Приведемо деякі функції, які часто зустрічаються при оцінці складності алгоритмів. Функції приведемо в порядку зростання обчислювальної складності зверху вниз. Ефективність степеневих алгоритмів звичайно вважається поганою, лінійних – задовільній, логарифмічних – хорошою.

Функція	Примітка
f(N)=C	C – константа
f(N)=log(log(N))	
f(N)=log(N)	
f(N)=NC	C – константа від нуля до одиниці

f(N)=N	
f(N)=N*log(N)	
$f(N)=N^C$	C – константа більша одиниці
$f(N)=C^N$	C – константа більша одиниці
f(N)=N!	тобто 1*2* … N

Оцінка з точністю до порядку дає верхню межу складності алгоритму. Те, що програма має певний порядок складності, не означає, що алгоритм буде дійсно виконуватися так довго. При певних вхідних даних, багато алгоритмів виконується набагато швидше, ніж можна припустити на підставі їхнього порядку складності.

У числових алгоритмах точність і стійкість алгоритмів не менш важлива, ніж їх часова ефективність.

Аналіз складності алгоритму корисний для розуміння особливостей алгоритму і звичайно знаходить частини програми, що витрачають велику частину комп'ютерного часу. Надавши увагу оптимізації коду в цих частинах, можна внести максимальний ефект в збільшення продуктивності програми в цілому.

Іноді тестування алгоритмів є найбільш відповідним способом визначити якнайкращого алгоритму. При такому тестуванні важливо, щоб тестові дані були максимально наближені до реальних даних. Якщо тестові дані сильно відрізняються від реальних, результати тестування можуть сильно відрізнятися від реальних.

Правила аналізу складності алгоритмів

У загальному випадку час виконання оператора або групи операторів можна розглядати як функцію з параметрами – розміром вхідних даних і/або одної чи декількох змінних. Але для часу виконання програми в цілому допустимим параметром може бути лише розмір вхідних даних.

Час виконання операторів присвоєння, читання і запису звичайно має порядок $O(1)$.

Час виконання послідовності операторів визначається за правилом сум. Тому міра росту часу виконання послідовності операторів без визначення констант пропорційності співпадає з найбільшим часом виконання оператора в даній послідовності.

Час виконання умовних операторів складається з часу виконання умовно виконуваних операторів і часу обчислення самого логічного виразу. Час обчислення логічного виразу часто має порядок $O(1)$. Час для всієї конструкції if-then-else складається з часу обчислення логічного виразу і найбільшого з часів, який необхідний для виконання операторів, що виконуються при різних значеннях логічного виразу.

Час виконання циклу є сумою часів усіх часів виконуваних конструкцій циклу, які в свою чергу складаються з часів виконання операторів тіла циклу і часу обчислення умови завершення циклу (часто має порядок $O(1)$). Часто час

виконання циклу обчислюється, нехтуючи визначенням констант пропорційності, як добуток кількості виконуваних операцій циклу на найбільший можливий час виконання тіла циклу. Час виконання кожного циклу, якщо в програмі їх декілька, повинен визначатися окремо.

Завдання на лабораторну роботу

1. Обчислити складність алгоритму з попередньої лабораторної роботи.
2. Вказати шляхи зменшення його складності.
3. Оформити звіт виконання лабораторної роботи.

ЛІТЕРАТУРА

1. Алферова. З.В. Теория алгоритмов.- М.: Статистика, 1973.- 164 с.
2. Вирт. Н Алгоритмы + структуры данных = программы.- М.: Мир, 1985.- 406с.
3. Глушков В. М., Цейтлин Г. Е., Ющенко Е. Л. Алгебра, языки, программирование. — 3-е изд., перераб. и доп. — К.: Наук. думка, 1989. 14
4. Грин Д., Кнут Д. Математические методы анализа алгоритмов.- М.: Мир, 1987.- 120 с.
5. Грин Д., Кнут Д. Математические методы анализа алгоритмов.- М.: Мир, 1987.- 120 с.
6. Гудман С., Хидетниеми С. Введение в разработку и анализ алгоритмов. — М.: Мир, 1981.
7. Гуц А.К. Математическая логика и теория алгоритмов: Учебное пособие. – Омск: Диалог – Сибирь, 2003. – 108 с.
8. Дж. Макконелл. *Анализ алгоритмов*. Техносфера. Москва, 2002.
9. Игошин В.И., Математическая логика и теорія алгоритмов , М.:Изд.центр «Академия», 2008. – 448с.
10. Калужнін Л. А., Королюк В. С. Алгоритми і математичні машини. — К.: Вища шк., 1964.
11. Кнут Д. Искусство программирования. Т.1. Основные алгоритмы. 2000, Вильямс.
12. Кнут Д. Искусство программирования. Т.2. Получисленные алгоритмы. 2000, Вильямс.
13. Кнут Д. Искусство программирования. Т.3. Сортировка и поиск. 2000, Вильямс.
14. Кормен Т., Лейзерсон Ч., Ривест Р., *Алгоритмы*: *построение и анализ.* М.: МЦНМО, 2000
15. Лавров И. А., Максимова Л. Л. Задачи по теории множеств, математической логике и теории алгоритмов. — М.: Наука, 1975.
16. Лісовик Л.П., Шкільняк С.С. Теорія алгоритмів: Навч. посібник.- К.: Видавничий поліграфічний центр “Київський університет”, 2003.-163 с.
17. Мендельсон Э. Введение в математическую логику. — М.: Мир, 1976.
18. Минский М. Вычисления и автоматы. — М.: Мир, 1971.
19. Роберт Седжвик. *Фундаментальные алгоритмы на JAVA.* Части 1-4. Анализ, структуры данных, сортировка, поиск. DiaSoft. Москва, Санкт-Петербург, Киев, 2003.
20. Роберт Седжвик. *Фундаментальные алгоритмы на С++*. Часть 5. Алгоритмы на графах. DiaSoft. Москва, Санкт-Петербург, Киев, 2002.
21. Трахтенброт Б. А. Алгоритмы и вычислительные машины. — М.: Сов. радио, 1974.
22. Тьюринг А. Может ли машина мыслить? — М.: Физматгиз, 1960.
23. Шкільняк С.С. Математична логіка: приклади і задачі. – Київ: ВПЦ "Київський університет", 2002. – 56 с.

Підписано до друку 4.09.2017 р.
Формат 84х108\32. Папір офсетний Друк на різографі.
Умов.-друк.арк. 4,3. Обл..-вид. арк. 4,8. Зам. №7.
Тираж 100 прим.

Віддруковано ФОП Шпак В.О.
м.Тернопіль, вул. Бродівська, 44
тел./факс 520563