Київський національний університет імені Тараса Шевченка
факультет комп'ютерних наук та кібернетики

ОСНОВИ КОМП'ЮТЕРНИХ АЛГОРИТМІВ:

МЕТОДИЧНІ ВКАЗІВКИ ДО ВИКОНАННЯ ЛАБОРАТОРНИХ РОБІТ

ТВОРИ – 2021

УДК 004.43.421(042.3)

Рекомендовано вченою радою факультету комп’ютерних наук та кібернетики Київського національного університету імені Тараса Шевченка (протокол №15 від 7 червня 2021 р.).

Укладач: кандидат фіз.-мат. наук, професор, доктор габілітації Вергунова І.М.

Рецензенти: доктор фіз.-мат. наук, доцент І.О. Завадський
доктор наук, професор В.Р. Стебловська

В 52 Вергунова І.М. Основи комп'ютерних алгоритмів: методичні вказівки до виконання лабораторних робіт. – Вінниця : ТВОРИ, 2021. – 40 с.

Викладено методичні вказівки до виконання лабораторних робіт з дисципліни «Основи комп'ютерних алгоритмів» для студентів факультету комп’ютерних наук та кібернетики спеціальності 122 «Комп'ютерні науки» ОП «Інформатика».

УДК 004.43.421(042.3)

Лабораторна робота 1. Використання структур даних

Лабораторна робота є підготовчою як до виконання наступних робіт, так і до засвоєння лекцій щодо працеємності та інших ресурсних характеристик комп'ютерних алгоритмів.

Класичний аналіз обчислювальних алгоритмів пов'язаний з аналізом їх часової складності. Його результатом є асимптотична оцінка кількості операцій, що задається алгоритмом, як функції довжини входу, що корельовано з асимптотичною оцінкою часу виконання програмної реалізації алгоритму. Але асимптотичні оцінки вказують лише порядок зростання функції, а тому результати порівняння алгоритмів за цими оцінками будуть виконуватися за дуже великих довжин входів. Для порівняння алгоритмів у діапазоні реальних довжин входів, що визначаються областю застосування програмної системи, необхідно знати точну кількість операцій, заданих алгоритмом, тобто його функцію працеємності.

Нехай D_A – множина припустимих конкретних проблем задачі, що вирішується алгоритмом A, а його елемент $D \in D_A$ є конкретною проблемою (вхід алгоритму A) вимірності n. Множина D є кінцевою впорядкованою множиною з n елементів d_i, що являють собою слова фіксованої довжини в абетці {0,1}: $D = \{d_i , i = 1,\ldots, n,\}$, $|D| = n$.

Працеємністю алгоритму A на вході D називають кількість базових операцій у прийнятій моделі обчислень, що задаються алгоритмом на цьому вході (позначимо $f_A(D)$). Значенням функції працеємності для будь-якого припустимого входу D є ціле додатне число [1].

Іноді при детальному аналізі алгоритмів виявляється, що не завжди працеємність алгоритму на одному вході D довжини n, де $n = |D|$, співпадає з його працеємністю на іншому вході такої же довжини. Розглянемо припустимі входи алгоритму довжини n. У загальному випадку існує підмножина, для більшості алгоритмів власна, множини D_A, що включає всі входи вимірності n, яку позначимо D_n:

$D_n = \{D \mid |D| = n\}$. Т.я. елементи d_i є словами фіксованої довжини в абетці {0,1}, множина D_n є скінченою (позначимо її потужність M_{Dn},

$M_{Dn}=|D_n|$). Тоді алгоритм А, одержуючи різні входи D з множина D_n, буде, можливо, задавати у деякому випадку найбільшу, а в деякому – найменшу кількість операцій. Виключення складають алгоритми, для яких працеємність визначається тільки довжиною входу (тоді $f_A(n)$).

Найгіршим випадком є найбільша кількість операцій, що задаються алгоритмом А на всіх входах вимірності n, тобто на всіх входах D ϵ D_n:

$$f_A^{\wedge}(n) = \max_{D \in D_n} \{f_A(D)\}.$$

Найкращим випадком є найменша кількість операцій, що задаються алгоритмом А на всіх входах вимірності n, тобто на всіх входах D ϵ D_n:

$$f_A^{\vee}(n) = \min_{D \in D_n} \{f_A(D)\}.$$

Працеємністю алгоритму А у середньому є середня кількість операцій, що задаються алгоритмом А на всіх входах вимірності n:

$$\overline{f}_A(n) = \sum_{D \in D_n} p(d) f_A(D),$$

де $p(D)$ – частота зустрічей входу D для області застосування алгоритму, що аналізується. Якщо всі входи D ϵ D_n є рівноймовірними, то

$$\overline{f}_A(n) = \frac{1}{M_{D_n}} \sum_{D \in D_n} f_A(D).$$

Часова складність алгоритму – асимптотична оцінка у класах функцій, що визначаються позначеннями O або Θ, функції працеємності алгоритму для найгіршого випадку $f_A^{\wedge}(n) = O(g(n))$ або $f_A^{\wedge}(n) = \Theta(g(n))$, де $g(n)$ – функція, що задає клас O або Θ для $f_A^{\wedge}(n)$ [1].

Маємо $f_A^{\wedge}(n) \le \overline{f}_A(n) \le f_A^{\vee}(n)$.

Стан пам'яті моделі обчислень визначається значеннями, записаними у комірках цієї пам'яті. Механізм реалізації моделі обчислень, виконуючи операції, задані алгоритмом, переводить початковий стан пам'яті моделі обчислень (вхідні дані задачі – вхід алгоритму) у кінцевий стан (знайдене алгоритмом рішення задачі). В ході рішення задачі може бути задіяно деяку додаткову кількість комірок пам'яті.

Під об'ємом пам'яті, що потрібний алгоритму А для входу, заданого множиною D, розуміють максимальну кількість комірок пам'яті моделі обчислень, задіяних у ході виконання алгоритму. Функція об'єму пам'яті алгоритму для входу D, що позначимо $V_A(D)$, є теж цілим додатним числом [1]. Для функції об'єму пам'яті алгоритму для входу D

аналогічним чином визначають найгірший ($V_A^{\wedge}(D)$), середній ($\overline{V}_A(D)$) та найкращий ($V_A^{\vee}(D)$) випадки.

Ємнісна складність алгоритму – це асимптотична оцінка в класах функцій, що визначаються позначеннями O або Θ, функції об'єму пам'яті алгоритму для найгіршого випадку $V_A^{\wedge}(n) = O(h(n))$ або $V_A^{\wedge}(n) = \Theta(h(n))$, де $h(n)$ – функція, що задає клас O або Θ для $V_A^{\wedge}(n)$.

Маємо $V_A^{\wedge}(n) \le \overline{V}_A(n) \le V_A^{\vee}(n)$ [1].

Ресурсною складністю алгоритму у найгіршому, середньому або найкращому випадку називають впорядковану пару класів функцій, заданих асимптотично позначеннями O або Θ (як відповідні випадку часова складність і ємнісна складність алгоритму), - $\Re_c^*(A) = \langle O(g(n)), O(h(n)) \rangle$.

Іноді більш наглядним є перехід в оцінці ємнісної складності від загального об'єму пам'яті моделі обчислень до об'єму додаткової пам'яті, що потрібна алгоритму, якщо об'єми пам'яті для входу й результату однакові для всіх порівнюваних алгоритмів. Тоді позначення всі зберігаються, вони використовують з відповідним уточненням. Наприклад, позначення ресурсної складності для алгоритму сортування вставками, для якого $f_A^{\wedge}(n) = \Theta(n^2)$, а потрібна додаткова пам'ять фіксована й не залежить від довжини масиву, що сортується. Для цього алгоритму $\Re_c^{\wedge}(A) = \langle \Theta(n^2), \Theta(1) \rangle$.

Функції ресурсної ефективності комп'ютерних алгоритмів та їх програмних реалізацій [1]. Розглянемо компоненти функції ресурсної ефективності програмної реалізації алгоритму.

Нехай D_A – конкретна множина початкових даних алгоритму, $T(D_A)$ – потрібний алгоритму ресурс процесора – оцінка часу виконання даного алгоритму на даному комп'ютері. Така оцінка визначається функцією працеємності алгоритму в залежності від характеристичних особливостей множини початкових даних. Перехід від функції працеємності до часової оцінки пов'язаний з визначенням середньозваженого часу $t_{оп}$ виконання узагальненої базової операції в мові реалізації алгоритму на даному процесорі й комп'ютері. Отримання точної функції часу виконання, яка враховує всі особливості архітектури комп'ютера, представляє собою доволі складну задачу. Для оцінки зверху можна використати функцію працеємності для найгіршого випадку за даної

вимірності $f_A^{\wedge}(n)$. У переважній кількості випадків може бути використана функцію працеємності для середнього випадку $\overline{f}_A(n)$. Середньозважений час $t_{оп}$ може бути отриманий дослідним шляхом у середовищі реалізації алгоритму.

Емпіричний метод одержання часових оцінок на основі функції працеємності [1]:

1. Виконується загальний аналіз працеємності алгоритму без поділу на операції (визначають функцію працеємності алгоритму для середнього випадку $\overline{f}_A(n)$ в узагальнених базових операціях прийнятої моделі обчислень).
2. Проводяться обчислювальні експерименти з програмною реалізацією алгоритму (для кожного з обраних значень вимірності n_i, $i = 1,\ldots, m$, виконують деяку кількість експериментів n_e, $j = 1,\ldots, n$, з різними початковими даними, в ході яких визначаються часи виконання $t_{ej}(n_i)$, на основі яких розраховують середній експериментальний час виконання:
$$\bar{t}_e(n_i) = \left(\frac{1}{n_e}\right)\sum_{j=1}^{n_e} \bar{t}_{e\,j}(n_i)\,.$$
3. Розраховують час середній ($\bar{t}_{оп\,A}(n_i)$) та за всім експериментом ($\bar{t}_{оп\,Ae}$), а саме за відомою функцією працеємності алгоритму для середнього випадку $\overline{f}_A(n)$ обчислюють середній час на узагальнену базову операцію, породжуваний даним алгоритмом, компілятором, ОС та комп'ютером для даної вимірності:
$$\bar{t}_{опA}(n_i) = \left(\frac{\bar{t}_e(n_i)}{\bar{f}_A(n_i)}\right), \quad \bar{t}_{опAe}(n_i) = \left(\frac{1}{m}\right)\sum_{i=1}^{m_e} \bar{t}_{опA}(n_i),$$
де m – кількість значень вимірності задачі, що тестується.
4. Виконують прогноз часової ефективності за умови припущення про стійкість середнього часу виконання узагальненої базової операції:
$$\overline{T}_A(n) = \bar{t}_{опAe}(n_i)\,\bar{f}_A(n), \quad n \notin n_i, \quad i = \overline{1,m}.$$

Наприклад, нехай маємо деякі дослідні дані за алгоритмом А (реалізація: конкретна мова, конкретна ОС, конкретний комп'ютер, конкретний процесор) [1], тестові варіанти отримані стандартним генератором псевдовипадкових чисел з рівномірним розподілом. Тоді для кожного фіксованого значення вимірності масиву виконуємо n_e дослідів, їх

зведені результати дають: t_{sum} – час у секундах за всіма дослідами; $\bar{t}_e(n_i)$ – середній час у секундах на виконання за вимірності n_i; $\bar{t}_{\text{оп А}}(n_i)$ – середній час на узагальнену базову операцію в наносекундах вимірності n_i.

***Умови лабораторної роботи*:**
Розробити та реалізувати алгоритм, що з використанням роздільного зв'язування вставляє N випадкових чисел у таблицю розміром N/100, а потім визначає довжину самого довгого та самого короткого списків, N = 103, 104, 105, 106 [2, 3]. Проаналізувати середній час виконання узагальненої базової операції [1].

При проведенні аналізу середнього часу виконання брати до уваги отримання часових оцінок на основі середнього часу виконання узагальненої базової операції. Для цього: - зробити сукупний аналіз працеємнсті алгоритму без розділення на різні операції (визначається в узагальнених базових операціях прийнятої моделі обчислень, тобто по рядках коду маємо просумувати всі виконувані операції); - виконати обчислювальні експерименти з програмною реалізацією алгоритму (на цьому етапі для кожного з вибраних значень вимірності задач N*, проводиться деяка кількість експериментів з різними початковими даними, в ході яких визначається їхній час виконання, на основі чого й розраховується середній дослідний час виконання).*

Лабораторна робота 2. Застосування поліноміальних хеш-функцій

Нехай заданий деякий рядок $S_{0..n-1}$ з n символів.
Поліноміальним хешем (поліноміальною хеш-функцією, функцією поліноміального хешу) при зростанні степенів у поліномі *зліва-направо* для послідовності символів $\{a_0, a_1, ..., a_{n-1}\}$ називають хеш-функцію вигляду

$$h(a, p, m) = (a_0 + a_1 p + a_2 p^2 + ... + a_{n-1} p^{n-1}) \bmod m,$$

де p – точка (основа), що є деяким натуральним числом, взаємно простим з m, m – модуль хешування (наприклад, 2^{64}), що задовольняють нерівності $\max(a_i) < p < m$, a_i – код i-го символу рядку S.

Хеш-функція співставляє послідовності $\{a_0, a_1, ..., a_{n-1}\}$ число довжини n у системі числення p та задає залишок від його ділення на число m, або значення багаточлена $(n-1)$-ї степені з коефіцієнтами a_i в точці p за модулем m (якщо значення $h(a, p, m)$, не взяте за модулем, вміщується у цілочисельний тип даних (наприклад, 64-бітний тип), то можемо кожній послідовності співставити це число, а тоді порівняння типу більше, менше, дорівнює можна виконувати за $O(1)$).

Для такого порівняння достатньо порахувати поліноміальну хеш-функцію на кожному префіксі початкової послідовності $\{a_0, a_1, ..., a_{n-1}\}$. У однакових рядків значення хеш-функції рівні. Крім того, якщо значення хеш-функцій будуть рівні, то можна зробити висновок про рівність підпослідовностей однакової довжини з дуже великою ймовірністю (можливе хибне спрацювання).

Розглянемо як порівняти довільні неперервні підвідрізки послідовності за $O(1)$ (звичайно для порівняння двох рядків довжини n у найгіршому випадку необхідно перевірити усі n символів, а це $O(n)$). Так як значення хеш-функції – це число, то для порівняння підвідрізків потрібно $O(1)$ часу. Але щоб порахувати хеш-функцію одного рядку напряму потрібно $O(n)$ часу. Тому за $O(n)$ шукають хеші одразу усіх підрядків. З цією метою:

1. Можемо визначити поліноміальну хеш-функцію на префіксі як

$$h_{pref}(k) = h_{pref}(k, a, p, m) = (a_0 + a_1 p + a_2 p^2 + ... + a_{k-1} p^{k-1}) \bmod m.$$

Маємо:

$$h_{pref}(0) = 0,\ h_{pref}(1) = a_0,\ h_{pref}(2) = (a_0 + a_1 p) \bmod m,$$

…

$$h_{pref}(n) = (a_0 + a_1 p + a_2 p^2 + ... + a_{n-1} p^{n-1}) \bmod m.$$

Тоді, для знаходження поліноміальної хеш-функції на кожному префіксі за час $O(n)$, можна використати наступні рекурентні співвідношення (за вказаної вище умови):

$$p^k = p^{k-1} p,\ \ h_{pref}(k) = h_{pref}(k-1) + a_k p^{k-1}.$$

Крім того,

$$h_{pref}(n) = h_{pref}(k-1) + p^k h_{pref}(k..n),$$

де $h_{pref}(k..n) = a_k p^k + a_{k+1} p^{k+1} + ... + a_{n-1} p^{n-1}$.

Якщо потрібно порівняти на рівність два підрядки однакової довжини *len*, які починаються в позиціях *i* та *j*, то потрібно розглянути різниці $h_{pref}(i+len) - h_{pref}(i)$ та $h_{pref}(j+len) - h_{pref}(j)$.

Маємо:

$$h_{pref}(i+len) - h_{pref}(i) = a_i p^i + a_{i+1} p^{i+1} + ... + a_{i+len-1} p^{i+len-1}, \quad (2.1)$$

$$h_{pref}(j+len) - h_{pref}(j) = a_j p^j + a_{j+1} p^{j+1} + ... + a_{j+len-1} p^{j+len-1}. \quad (2.2)$$

Приведемо ці рівняння до одного степеня за основою *p*, для цього домножимо справа й зліва на величини p^j та p^i, відповідно:

$$\left(h_{pref}(i+len) - h_{pref}(i)\right)p^j = \left(a_i + a_{i+1} p + ... + a_{i+len-1} p^{len-1}\right)p^{i+j},$$

$$\left(h_{pref}(j+len) - h_{pref}(j)\right)p^i = \left(a_j + a_{j+1} p + ... + a_{j+len-1} p^{len-1}\right)p^{j+i}.$$

В одержаних рівняннях в правій частині в дужках маємо поліноміальні хеш-функції підвідрізків $a_i, a_{i+1},...,a_{i+len-1}$ та $a_j, a_{j+1},...,a_{j+len-1}$. Щоб визначити чи співпадають ці підвідрізки, потрібно перевірити виконання такої рівності:

$$\left(h_{pref}(i+len) - h_{pref}(i)\right)p^j = \left(h_{pref}(j+len) - h_{pref}(j)\right)p^i.$$

Одне таке порівняння можна виконати за час $O(1)$, якщо наперед визначати степені *p* за модулем *m* (використовуючи рекурентний вираз $p^k = p^{k-1} p$). З урахуванням модуля *m* попередня рівність приймає вигляд:

$$p^j \left(h_{pref}(i+len) - h_{pref}(i)\right) \bmod m = p^i \left(h_{pref}(j+len) - h_{pref}(j)\right) \bmod m. \quad (2.3)$$

Але рівність (2.3) показує, що порівняння рядків залежить від параметрів порівнюваних з ними рядків. Для вирішення цієї проблеми рівняння (2.1) множать на p^{-i}, а (2.2) – на p^{-j}. Тоді маємо:

$$\left(h_{pref}(i+len) - h_{pref}(i)\right)p^{-i} = a_i + a_{i+1} p + ... + a_{i+len-1} p^{len-1},$$

$$\left(h_{pref}(j+len) - h_{pref}(j)\right)p^{-j} = a_j + a_{j+1} p + ... + a_{j+len-1} p^{len-1}.$$

В правих частинах отримано хеш-функції для підвідрізків, що дає підставу для наступного рівняння для перевірки рівності цих підвідрізків:

$$p^{-i}\left(h_{pref}(i+len) - h_{pref}(i)\right) = p^{-j}\left(h_{pref}(j+len) - h_{pref}(j)\right) \quad (2.4)$$

Для реалізації такого підходу необхідно знайти обернений елемент для p за модулем m, а так як найбільшим спільним дільником цих чисел є 1, то такий елемент існує. Якщо наперед порахувати степені оберненого елемента за модулем, то порівняння можемо виконати за $O(1)$.

2. Нехай *Mrow* – максимальна довжина порівнюваних підрядків. Домножимо рівняння (2.1) на p в степені $Mrow - i - len + 1$, а рівняння (2.2) – на p в степені $Mrow - j - len + 1$:

$$p^{Mrow-i-len+1}\left(h_{pref}(i+len)-h_{pref}(i)\right)=p^{Mrow-len+1}\left(a_i+a_{i+1}p^1+\ldots+a_{i+len-1}p^{len-1}\right), \tag{2.5}$$

$$p^{Mrow-j-len+1}\left(h_{pref}(j+len)-h_{pref}(j)\right)=p^{Mrow-len+1}\left(a_j+a_{j+1}p^1+\ldots+a_{j+len-1}p^{len-1}\right). \tag{2.6}$$

В рівняннях (2.5), (2.6) справа містяться поліноміальні хеші для заданих підрядків, тому рівність цих підрядків можемо перевірити з наступного рівняння:

$$p^{Mrow-i-len+1}\left(h_{pref}(i+len)-h_{pref}(i)\right)\bmod m = p^{Mrow-j-len+1}\left(h_{pref}(j+len)-h_{pref}(j)\right)\bmod m. \tag{2.7}$$

Рівняння (2.7) дозволяє порівнювати підрядок довжини *len* з іншими підрядками такої ж довжини, причому вони можуть бути підрядками іншого рядку, так як вирази справа та зліва в рівнянні залежать тільки від власних параметрів розглянутого підрядку (*i*, *len*) та сталої *Mrow*.

Отже, для обчислення хешу будь-якого підрядка $h_{pref}(n)$ необхідно знати хеш-функції префіксів підрядків $h_{pref}(0..0)$, $h_{pref}(0..1)$, $h_{pref}(0..2)$, …, $h_{pref}(0..n-1)$:

$$h_{pref}(0..0) = a_0,$$

…,

$$h_{pref}(0..k) = (h_{pref}(0..k-1)p + a_k) \bmod m,\ p(k) = p^k \bmod m, \ldots .$$

Так як хеш кожного наступного префіксу виражається через хеш попереднього, то вважатимемо це лінійним проходом за рядком. Усе разом зможемо обрахувати за $O(n)$ часу. Степені p рахуються наперед і зберігаються у масиві.

Але, так як рядки можуть бути дуже довгими, то при обчисленні хеш-функції можемо отримати переповнення. Тому хеші рахують, наприклад, за mod 2^{64}, $2^{31} - 1$, $p > \max\{a_i\}$ взаємно просте з m, непарне (якщо

маємо справу лише з прописними символами латинської абетки, наприклад, p = 31). Так як всі операції виконуються за mod, то для позбавлення від ділення вирази часто приводять до загального знаменника і замість ділення використовують множення (наприклад, вираз (2.7)). Але можна й використовувати вираз типу (2.4). Одне таке порівняння виконуємо за $O(1)$.

Якщо маємо надто великі значення для модуля m, можна використовувати подвійний поліноміальний хеш. Якщо візьмемо два взаємно простих модуля m_1 та m_2, то кільце залишків за модулем $m = m_1 \cdot m_2$ еквівалентне добутку кілець за модулями m_1 та m_2 (між ними взаємно однозначна відповідність, основана на ідемпотентах кільця вичетів за модулем m). Тобто, можна обчислювати h за модулем m_1 та за модулем m_2, а потім порівнювати два підвідрізки.

Для *порівняння* двох підрядків довжиною n, необхідно спочатку перевірити на співпадіння довжини відрізків, обчислити p^i , i = 0, …, n-1, у масиві, перерахувати хеш-функції для всіх l, l = 0, …, N-n, як

$$h_{pref}(l+1..l+n-1) = (h_{pref}(l..l+n-1) - a_l p^n) \bmod m,$$

$$h_{pref}(l+1..l+n) = (p h_{pref}(l+1..l+n-1) + a_{l+n}) \bmod m.$$

Звідси

$$h_{pref}(l+1..l+n) = ((h_{pref}(l..l+n-1) - a_l p^{n-1}) p + a_{l+n}) \bmod m. \quad (2.8)$$

Вираз (3.8) є основою для алгоритму порівняння підрядків довжини n, але внаслідок наявності можливих хибних спрацювань, маємо ще проводити перевірку результатів (асимптотична оцінка $O(n)$).

У загальному алгоритм може бути наступним:

1. Перевіряємо на співпадіння довжини відрізків.
2. Зберігаємо обчислені p^i у масиві.
3. Обчислюємо хеш-функції префіксів першого рядку.
4. Обчислюємо хеш-функції префіксів другого рядку.
5. Порівнюємо 2 рядки за $O(1)$.
6. Перевірку результат для рядків, що співпали, на наявність хибного спрацювання.

Для *пошуку підрядку* довжиною n у рядку довжиною N знаходимо хеш підрядку та хеші всіх префіксів рядку, а потім рухаємося по рядку вікном довжини n порівнюючи хеші $h_{pref}(i..i+n-1)$.

Для *знаходження* z-функції (z-функція рядку довжиною n – це масив z, i-й елемент якого дорівнює найбільшому числу символів, починаючи з позиції i у рядку, що співпадають з першими символами рядку (асимптотична оцінка $O(n\log(n))$ для $i = 0, 1, \ldots, n\text{-}1$ шукаємо $z[i]$ бінарним пошуком.

Для *пошуку усіх паліндромів* у рядку (асимптотична оцінка $O(n\log(n))$, тобто для пошуку підрядків $S_{L..R}$, в яких $a_L=a_R$, $a_{L+1}=a_{R-1}$, …, поруч із масивом, який містить хеші $h_{pref}(0..0)$, $h_{pref}(0..1)$, $h_{pref}(0..2)$, …, $h_{pref}(0..n-1)$ одержимо подібний масив з хешами для оберненого рядку. Розглянемо позицію i у рядку. Нехай існує паліндром непарної довжини d з центром у позиції i (парна – позиція між $i-1$, i). Якщо обрізати його по краях на 1 символ, він залишиться паліндромом. Діємо так, доки довжина не стане 0. В результаті достатньо для кожної позиції зберігати два значення: скільки існує паліндромів непарної довжини з центром у позиції i та скільки існує їх парної довжини з центром між між $i-1$, i.

При зростанні степенів основ p, тобто *справо-наліво* (використовує алгоритм Рабіна-Карпа [2]) ***поліноміальним хешем*** для рядку $S_{0..n}$ з $n+1$ символів у варіанті зростання степенів у поліномі справо-наліво для послідовності символів $\{a_0, a_1, ..., a_n\}$, називають хеш-функцію

$$h(a, p, m) = (a_0 p^n + a_1 p^n + a_2 p^{n-2} + ... + a_n) \bmod m,$$

m – модуль хешування, що є фіксованим простим доволі великим числом (наприклад, $2^{31}-1$, $2^{61}-1$, $2^{32}-5$, $2^{64}-59$), основу p вибирають випадковим чином перед роботою алгоритму з діапазону від 0 до $m-1$. Наведений у визначенні вираз запишемо у наступному вигляді:

$$h(a, p, m) = (...((a_0 p + a_1) p + a_2) p + ... + a_{n-1}) p + a_n) \bmod m.$$

Вказане дає можливість одержати наступне:

$$h_{pref}(i+1..i+n) = ((h_{pref}(i..i+n-1) - a_i p^{n-1}) p + a_{i+n}) \bmod m. \quad (2.9)$$

Вираз (2.9) і є основою для алгоритму порівняння підрядків.

Оцінимо ймовірність колізій при використанні поліноміальної хеш-функції.

Нехай потрібно обчислювати хеш за модулем m та провести порівняння n рядків. Використовуючи парадокс днів народження одержимо,

ймовірність того, що відбудеться колізія: $p \approx 1 - \exp(\frac{n^2}{2m})$. Наведене показує, що величину m потрібно брати значно більшою за n^2. Тоді $p \approx \frac{n^2}{2m}$, але вказана ймовірність не враховує зламу.

***Умови лабораторної роботи*:**

І. Спільна частина.

Реалізувати структуру даних типу «множина рядків». Рядки – непусті послідовності довжиною до 15 символів з рядкових латинських літер. Структура даних повинна підтримувати операції додавання рядку до множини, вилучення та перевірки належності даного рядку множині. Максимальна кількість рядків у множині, що зберігається, не більше 10^6.

Вхідні дані. Кожен рядок вхідних даних задає одну операцію над множиною. Запис операції складається з типу операції та наступного за ним через пробіл ряду, над яким проводять операцію. Типи операції вказують символи: «+» – додавання рядку, «–» – вилучення, «?» – перевірка на належність. Загальні кількість операцій у вхідному файлі не більше 10^6. Список операцій завершується рядком із символом # (ознака кінця вхідних даних).

При додаванні елементу до множини не гарантується, що він відсутній у множині, а при вилученні елементу з множини не гарантується, що він є у множині.

Вихідні дані. Виводяться для кожної операції типу «?» рядок yes або no, в залежності від того, чи зустрічається дане слово у множині.

ІІ. Частина за варіантами.

Варіант 1. Знайти усі повторювані рядки та поділити їх на групи, щоб у кожній групі виводився повторюваний рядок та кількість його повторювань. Оцінити час виконання.

Варіант 2. Знайти усі паліндроми в множині рядків. Оцінити час виконання.

Варіант 3. Знайти z-функції рядків. Оцінити час виконання.

Лабораторна робота 3. Фільтри Блума

Фільтр Блума – це структура даних для перевірки належності елемента до множини (наприклад, якоїсь небажаної інформації). На практиці фільтр Блума застосовують для фільтрації запитів до зовнішньої пам'яті або даних, що передаються по мережі, щоб уникнути високовартісних операцій доступу, у проксісерверах, БД тощо. У фільтрі Блума можливі *хибнопозитивні* спрацювання й немає хибнонегативних.

Класичний фільтр Блума дозволяє виконувати тільки операції вставки елементу та перевірки належності.

Фільтр Блума містить масив з m біт та декілька хеш-функцій $h_0, h_1, h_2, ..., h_{l-1}$, що належать класу універсальних хеш-функцій H_{pm} та є незалежними [5]. Коли фільтр не містить жодного елемента, то всі m бітів є нульовими. При додаванні елементу з ключом k обчислюються позиції $h_0(k), h_1(k), h_2(k), ..., h_{l-1}(k)$. Елемент з ключом k можливо міститься у фільтрі, тільки якщо на всіх цих позиціях у масиві стоять 1 (рис. 3.1), інакше елементу немає в множині (що задається фільтром). Така структура звичайно займає менший об'єм пам'яті, ніж хеш-таблиця, але не є детермінованою множиною.

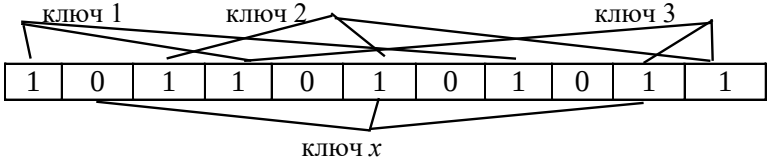

Рис. 3.1. Перевірка наявності ключа у фільтрі (три хеш-функції)

Чим більший об'єм пам'яті, що є наперед заданим нами, тим менша ймовірність хибного спрацювання. Недоліком є те, що у класичному варіанті підтримуються лише операції додавання та перевірки наявності елементів (пошуку не потрібно), а вилучення елементу немає (для уникнення цього існують різні модифікації з лічильниками).

Хеш-функції $h_0, h_1, h_2, ..., h_{l-1}$ рівноймовірно відображають елементи початкової множини у множину $\{0,1,..., m-1\}$ відповідним номером

бітів у масиві (результат їх може бути 0,1,…,*m*-1). Для додавання елементу *x* до множини необхідно записати 1-ці на кожну з позицій $h_0(x), h_1(x), h_2(x), ..., h_{l-1}(x)$ бітового масиву. Щоб перевірити, чи елемент *x* належить множині елементів, що зберігаються у фільтрі, необхідно перевірити стани бітів $h_0(x), h_1(x), h_2(x), ..., h_{l-1}(x)$ (якщо l = 3 як на рис. 3.1, то перевіряємо ці 3 біта). Якщо всі вони дорівнюють 1, то структура відповідає, що елемент належить множині (але може бути хибне спрацювання).

Оптимальний розмір масиву в бітах:

$$m = \frac{-n \ln p}{(\ln 2)^2}, \qquad (3.1)$$

n – припустима кількість елементів, що зберігаються у фільтрі-множині, *p* – бажана ймовірність хибного спрацювання.

Оптимальна кількість хеш-функцій:

$$l = \frac{m}{n} \ln 2 \text{, що дає також } l = -\frac{\ln p}{\ln 2}. \qquad (3.2)$$

При роботі фільтру, як вже вказувалося, перевіряється наявність елементу в множині, але він не дістається, тому в фільтрі не витрачається час на пошук. Миттєва перевірка – чи є деякі дані, наприклад, спам, шкідливі файли. Інформацію розбивають на блоки та додають у фільтр. Коли потрібно просканувати дані на наявність шкідливої активності, то беруть блоки сховища даних та перевіряють чи є вони у фільтрі.

Основні операції у фільтрі: додавання елементу до множини, що представлена фільтром; перевірка належності елементу до множини, що представлена фільтром. Виконання вказаних функцій відбувається наступним чином.

Додавання до фільтру [5]: InsBlF(T,x)

Вхід: T – бітовий масив розміром *m*,

x – запис на додавання до множини, що представлена фільтром

Вихід: запис доданий до фільтру, тобто встановлені біти $h_0(x), h_1(x), h_2(x), ..., h_{l-1}(x)$ на 1

for i = 0..*l*-1

обраховуємо $h \leftarrow h_i(x)$

встановлюємо $T(h) \leftarrow 1$

Перевірка [5]: ScinBlF(T,x)
Вхід: T – бітовий масив розміром m,
x – запис на перевірку належності до множини, що представлена фільтром
Вихід: true, false

for i = 0..l-1
обраховуємо $h \leftarrow h_i(x)$
if $T(h) = 0$ then
return false
return true

Розглянемо як отримати формули для оптимальних значень (3.1), (3.2). Якщо є одна хеш-функція, то ймовірність, що конкретний біт у таблиці буде розміщений, коли ми хешуємо елемент дорівнює $1/m$. Тому ймовірність того, що конкретний біт не буде встановлений буде $1 - 1/m$. Якщо маємо l незалежних хеш-функцій, то ймовірність, що конкретний біт не буде розміщений дорівнює $(1 - 1/m)^l = (1 - 1/m)^{ml/m} = (1/e)^{l/m}$. Якщо у фільтрі n елементів, то ймовірність того, що біт не буде розміщений дорівнює $p = (1/e)^{ln/m}$, а ймовірність того, що біт буде розміщений дорівнюватиме $1 - e^{-ln/m}$. Тоді ймовірність одержання хибнопозивного результату дорівнює ймовірності, що після додання n елементів перевіряємо елемент, який не у фільтрі, а всі l хеш-значень цього елементу знаходяться у зайнятих позиціях (дорівнює ймовірності, що l конкретних бітів розміщені) $p = (1 - e^{-ln/m})^l$. Звідси й маємо оптимальні значення для фільтру Блума: $m = \dfrac{-n \ln p}{(\ln 2)^2}$ та $l = -\dfrac{\ln p}{\ln 2}$.

Нехай $m = 1 \cdot 10^{10}$ біт, $n = 1 \cdot 10^9$ елементів у фільтрі. Тоді $l = \dfrac{m}{n} \ln 2 \approx 7$ різних незалежних хеш-функцій, а ймовірність хибнопозитивного результату буде: $p = (1 - e^{-7/10})^7 \approx 0.008 < 1\%$. Проте навіть з 5 хеш-функціями можемо досягти менше 1% для хибнопозитивного результату.
Якщо маємо значення l, задана ймовірність p та кількість елементів n, то визначаємо кількість біт для фільтра. За $p = 1\%$, $n = 1 \cdot 10^9$ елементів, $l = 7$, то $m \approx 1 \cdot 10^{10}$ біт. Якщо середній розмір елемента складає 10 біт,

то можемо опрацювати більше 10 Гбайт даних з обчислювальною швидкістю декількох хеш-функцій.
Можливі об'єднання, перетин та ієрархічна побудова фільтрів Блума. Перетин та об'єднання з однаковим розміром і набором хеш-функцій реалізують на практиці за допомогою відповідно операцій порозрядного ТА й АБО. Операція об'єднання фільтрів не має втрат в тому смислі, що результуючий фільтр Блума співпадає з фільтром Блума, просто створеним з використанням об'єднання двох наборів. Операція перетину має наступну властивість: ймовірність хибного спрацювання у результуючому фільтрі Блума не більш чим ймовірність хибного спрацювання в одному із складових фільтрів, проте може бути більшою, ніж ймовірність хибного спрацювання у фільтрі, просто створеного на основі перетину відповідних двох наборів.

***Умови лабораторної роботи*:**
Реалізувати структуру даних фільтр Блума. Елементи-рядки – непусті послідовності довжиною до 15 символів латинських літер. Максимальна кількість елементів у множині не перевищує $n = 1 \cdot 10^6$. Ймовірність хибнопозитивних спрацювань – 1%.
Структура даних повинна підтримувати операції:

- перевірки належності рядка множині;
- додавання рядка до множини (не гарантується, що цього рядка немає у множині).

Додатково можна реалізувати підтримку операції вилучення (з підрахунком, тобто лічильником).
Вхідні дані: файл, що містить рядки (<символ операції> <до 15 символів латинської абетки>). Типи операцій вказують символи: «+» – додавання рядку, «?» – перевірка на належність («–» – при підтримці вилучення).
Загальна кількість рядків у вхідному файлі не перевищує $n = 1 \cdot 10^6$. Список операцій завершується символом #.
Вихідні дані: для кожної операції перевірки «Y», якщо слово належить множині, «N» – якщо не належить.
Оцінити середній час виконання операцій у побудованому фільтрі.

Лабораторна робота 4. Алгоритми CRC (циклічні надлишкові коди)

Розглянемо некриптографічні CRC, які застосовуються як хеш-функції та як перешкодостійке кодування. Некриптографічні CRC використовують у різних протоколах зв'язку для перевірки цілісності даних, що передаються різними пристроями, при випадкових викривленнях інформації в каналі передачі даних, але не використовують для захисту [6, 7]. Алгоритми таких CRC мають невисокі витрати ресурсів та прості у реалізації. Часто реалізуються на апаратному рівні (архіватор стискує файли у відповідності з деяким алгоритмом архівації, обчислюючи з кожного стиснутого файлу CRC, потім заархівовані файли можна копіювати, пересилати тощо, а інформація може викривитися, навіть у 1 біті; коли архів розпаковують, архіватор у першу чергу перевіряє цілісність файлів (знову обчислює CRC та порівнює з першим значенням CRC, якщо вони рівні, значить цілісність порушена не була та архів розпаковують, інакше – ні)). CRC не обов'язково обчислювати для великих масивів даних (файл), можна це робити для окремих рядків, слів тощо. Для реалізації CRC використовують теорію лінійних циклічних кодів.

Важливим поняттям є поняття **контрольної суми** (КС) – деякого значення, обчисленого за визначеною схемою на основі кодового повідомлення. Використовують в якості перевірочної інформації, що дописується до даних, які передаються. На приймаючій стороні відомий алгоритм обчислення КС і відбувається перевірка прийнятих даних.

Сутність таких дій: за добрих характеристик КС похибка у повідомленні переважно приведе до зміни у КС. Якщо КС початкова та перевірочна не однакові, то приймається рішення про недостовірність прийнятих даних, відбувається запит на повторну передачу пакету. Визначення поодиноких похибок відбувається з приблизно 100 % ймовірністю, інших – з ймовірністю $1 - 2^{-N}$, де N – число розрядів контрольного коду.

Алгоритми CRC базуються на поліноміальній арифметиці: повідомлення, дільник та залишок мають бути представлені у вигляді поліномів з двійковим коефіцієнтами (послідовності бітів, кожен з яких є коефіцієнтом поліному).

Алгоритм CRC використовує властивості ділення із залишком двійкових багаточленів (значення CRC є залишком від ділення багаточлена, що відповідає вхідним даним, на деякий фіксований *породжуючий багаточлен*).

Для кожної послідовності бітів $a_0 \dots a_{N-1}$ взаємно однозначно складається двійковий поліном $a_0x^{N-1}+\dots+a_{N-1}$, послідовність коефіцієнтів якого являє собою вихідну послідовність. Послідовність бітів 1011010 відповідає багаточлену $P(x) = x^6+x^4+x^3+x$.

Кількість різних багаточленів степені менше N дорівнює 2^N, що співпадає з числом усіх двійкових послідовностей довжини N.

Значення КС визначають як бітову послідовність довжини N, що є багаточленом $R(x)$, який є залишком від ділення багаточлена $P(x)$, що є вхідним потоком біт, на багаточлен $G(x)$: $R(x)=P(x)x^N \mathrm{mod} G(x)$, де $G(x)$ – породжуючий багаточлен степені N.

В CRC алгоритмі використовують поліноміальну арифметику за модулем 2, тому дії, що виконуються під час обчислення CRC є арифметичними операціями без урахування переносу. Операції складання та віднімання тут ідентичні операції XOR (виключаюче АБО). Ділення відбувається аналогічно до звичайного, але замість віднімання при діленні стовпчиком виконується операція XOR. На рис. 4.1. представлене виконання операції над повідомленням 1101011011 з поліномом 10011. Отриманою контрольною сумою (CRC) є 1110, що відповідає залишку від ділення без урахування переносу (а результат, тобто частка, не потрібний для використання).

Часто з файлу береться перше слово (для CRC-1 маємо бітовий елемент, для CRC-8 – байтовий), у найпростішому алгоритмі, якщо старший біт у слові «1», то слово зсувають вліво на 1 розряд з наступним виконанням операції XOR з породжуючим поліномом. Якщо старший біт «0», то зсувають вліво на 1 розряд без виконання операції XOR. Так як результат не представляє жодного інтересу, то після зсуву старший біт втрачається. На місце молодшого біту (яке тепер містить «0») загружають наступний біт з файлу і т.д. до останнього біту з файлу. Після проходження усього повідомлення у схові залишається залишок – контрольна сума (КС). Якщо, наприклад, потрібно отдержати контрольну суму 8-розрядну, то утворюючий поліном береться 9-розрядний (тобто має 8-му степінь).

```
                    1100001010   частка
10011       11010110110000   вирівняне повідомлення (1101011011+0000)
Поліном     10011
             10011
             10011
                   1
              00000
                   10
               00000
                   101
                00000
                   1011
                 00000
                   10110
                   10011
                    01010
                    00000
                     10100
                     10011
                        1110    залишок (контрольна сума)
```

Рис. 4.1. Обчислення побітово контрольної суми (КС).

Множення на x^N – це приписування N нулевих бітів до вхідної послідовності, що покращує якість хешування для коротких вхідних послідовностей (можемо одержати 2^N різних залишків від ділення).

Багаточлен G(x) часто є непривідним. Звичайно його підбирають у відповідності з вимогами до хеш-функції для кожного конкретного застосування (існують стандартні, що можуть задати мінімальне число колізій, просто обчислюватися). Але стандартно використовувані поліноми не є самими ефективними.

Важливою характеристикою є степінь поліному (ширина – width). Часто вибирають величини, кратні до розрядності регістрів процесору для спрощення реалізації алгоритмів (16, 32 та ін.). Степінь поліному – це дійсна позиція старшого біту. Позиції бітів відліковують починаючи з 0. Отже, породжуючий поліном, а саме його степінь, визначає кількість бітів, використовуваних для обчислення CRC. Ще також важливим параметром є: початкове (стартове) значення слова.

Існують різні модифікації алгоритму (прямий, дзеркальний, табличний (що використовує асоціативність операції XOR) тощо).
Найпростіший прямий **алгоритм** – **побітовий зсув** [7], який можна представити у наступному вигляді.

1. Створюється масив (регістр), заповнений 00…00, що дорівнює за довжиною розрядності (степені) поліному.
2. Початкове повідомлення заповнюється 00…00 у молодших розрядах, за кількістю рівних числу розрядів поліному.
3. У молодший розряд регістру заносять старший біт повідомлення, а із старшого розряду регістру висувають один біт.
4. Якщо висунутий біт дорівнює 1, то відбувається інверсія бітів у тих розрядах регістру, що відповідають першим в поліномі.
5. Якщо у повідомленні є ще біти, то переходять до виконання 3.
6. Якщо всі біти повідомлення оброблені алгоритмом, у регістрі залишається залишок від ділення, який і є CRC (КС).

Отже, повідомлення T, що передається, є кратним дільнику (останні W бітів повідомлення T – це залишок від ділення повідомлення M на дільник: T = M + CRC). А так як додавання є одночасно й відніманням, воно зменшує M до найближчого кратного. Оскільки повідомлення може бути дуже великим (наприклад, декілька Мбт) та використовується для отримання контрольної суми CRC-арифметика, то використання звичайної комп'ютерної операції ділення неможливе. Найпростішим підходом є наступний. Нехай поліном з N = 4 має вигляд 10111. Тоді регістр буде чотирьохбітним, а алгоритм, що повертає значення регістру, де буде знаходитися КС, у загальному вигляді буде такий:

```
CRC_простий(КС)
Завантажити регістр нульовими бітами
Доповнити кінцеву частину повідомлення N нульовими бітами
while (є необроблені біти)
    begin
    зсунути регістр на 1 біт вліво
    розмістити черговий ще не оброблений біт повідомлення у 0 позицію регістру
    if (з регістру був висунутий біт із значенням «1»)
      регістр = регістр XOR поліном
end
return регістр
```

Якщо ж до регістру на початку обчислень записати не 00…00, а F…F, то це підвищить надійність визначення початку передачі повідомлення, якщо, наприклад, повідомлення на початку має нульові біти. Крім того, необхідно відокремлювати повідомлення та значення КС. Уникнути прямого відслідковування довжини рядку, що приймається, можна шляхом формування джерелом КС на початку повідомлення або, взагалі, окремо від рядку (це характерно для програми, що відслідковує цілісність файлів у деякому каталозі, тоді результати CRC для кожного файлу зберігаються у деякому службовому файлі). В деяких модифікаціях алгоритму може виконуватися й таке: - безпосередньо перед видачою КС результат ділиться на яке-небудь інше число; - байти повідомлення у процесі запису до регістру можуть розміщуватися як старшим бітом уперед, так і молодшим; - результуючий CRC може також видаватися, починаючи від старшого біту або від молодшого (реверс).
Тому на значення КС впливають наступні параметри:

- порядок CRC;
- утворюючий багаточлен;
- початковий вміст регістру;
- значення, з якого відбувається фінальний XOR;
- реверс байтів повідомлення;
- реверс байтів CRC перед фінальним XOR.

Але, взагалі, для прискорення можна виконувати ділення й так:

```
101111001110 | 10011
10011        |------
…10010
  10011
  -----
  00001 0111
      1 0011
      ------
        1000  – КС
```

Хоча й цей підхід, внаслідок дуже великої довжини повідомлень, вважають повільним. Вдосконалення такого підходу до ділення привело до появи табличних алгоритмів пошуку CRC.

Прямий табличний алгоритм. Отже, побітовий алгоритм для практичних повідомлень є надто повільним та неефективним. Асоціативність операції XOR дозволяє проводити обчислення з цілими байтами. Представимо такі обчислення, наприклад, для поліному 32 степені

(N = 32). Розглянемо регістр, який використовують для збереження проміжного результату обчислення CRC. Коли біт, що висунутий з лівої частини регістру, є 1, то виконується операція XOR вмісту регістру з молодшими N бітами поліному (тут N = 32), тобто виконуємо ділення подібно до наведеного вище. Якщо ж зсувати одночасно цілі групи біт, то можемо мати наступне.

Нехай до зсуву у регістрі маємо: 10110100110000100010001000100010001. Потім 16 біт висуваються зліва, а справа надходять нові 16 біт (наприклад, 0010001000100011). Тоді поточний вміст регістру: 00010001000100010010001000100011. Біти, що тільки но висунуті зліва: 1011010011000001. Поліном, наприклад, заданий такий: 10100000000000000000000000010101111.

Обчислимо нове значення регістру:

10110100110000010001000100010001001000100010001100011
10100000000000000000000000010101111

Поліном складаємо по XOR починаючи з 15 біту старшої групи. Отримаємо:

00010100110000010001000101000110.

Далі поліном складаємо по XOR знову.

10100110000010001000101000110101000100010001100011
10100000000000000000000000010101111

Маємо

00000110000010001000101001100010100100010001100011

Загалом старші 8 бітів тепер містять нулі. Теж саме могли б отримати, якщо скласти операнди у першому та другому діленні (поліноми), що прикладають до відповідних бітів старшої групи (0 та 3), й результат застосувати по XOR до початкового значення регістру. Це вказує на можливість наперед розрахувати величини XOR-доданку для кожної комбінації старших бітів.

Якби працювали просто послідовно, то мали б наступне.

10100000000000000000000000010101111

Результат:

01100001000100010100110000000001011

Знову складаємо по XOR.

11000010001000101001100000001011010001100011
10100000000000000000000000010101111

Результат:

01100010001000101001100001011100 1
Знову
11000100010001010011000010111001100011
10100000000000000000000000010101111
Результат:
01100100010001010011000011101110 0
Знову
11001000100010100110000111011100 00011
10100000000000000000000000010101111
Результат:
01101000100010100110000110001011 1
Знову
11010001000101001100001100010111 0011
10100000000000000000000000010101111
Результат:
01110001000101001100001101000000 1
Знову
11100010001010011000011010000001011
10100000000000000000000000010101111
Результат:
01000010001010011000011011010110 1
Знову
10000100010100110000110110101101 11
10100000000000000000000000010101111
Результат:
00100100010100110000110111111010 0
Знову
10010001010011000011011111101001
10100000000000000000000000010101111
Але вже не вистачає біта для ділення. Якщо немає необроблених бітів, то 10010001010011000011011111101001 є CRC, якщо є ще необроблені біти повідомлення то зсуваємо біти вліво й до регістру справа надходять нові біти (32). Але це не ефективно.
Якщо ж для кожної комбінації бітів старшої групи, що висуваються з регістру можемо розрахувати доданок, то одержимо відповідну таблицю (якщо 8 бітів, то з 256 подвійних слів).

Отже, алгоритм можна оптимізувати. Немає потреба постійно протискати байти рядку через регістр. Можна безпосередньо застосовувати операцію XOR байтів, висунених з регістру, з байтами оброблюваного рядку. Результат буде вказувати на позицію в таблиці, значення з якої й буде на наступному кроці складатися з регістром. Обчислення таблиці відбувається одноразово, результати зберігаються.
Тоді, наприклад, для N = 8 можемо одержати такий найпростіший варіант алгоритму.

Прямий табличний алгоритм [5].

1. Вибираємо перший байт масиву, який розглядається як адреса в таблиці. Вибираємо в таблиці число з даним номером (адресою) – одержуємо перший залишок.
2. Беремо другий байт інформаційного масиву та сумуємо його за mod2 із залишком. Одержане число є адресою в таблиці. Для цієї адреси беремо з таблиці залишок два.
3. Далі беремо наступний байт масиву, сумуємо за mod2 із залишком два. Одержане число є адресою в таблиці. Для цієї адреси беремо з таблиці залишок.
4. Виконуємо 3 доки не досягнемо кінця масиву, після його обробки у вихідному регістрі буде знаходитися значення КС.

Або більш докладно:

While (повідомлення повністю не оброблене)
Begin
Перевіримо старший байт регістру
Розрахуємо на його основі контрольний байт
Базуючись на значенні контрольного байту,
виконаємо операцію XOR обраного поліному з різними зміщеннями
Зсунемо регістр на один байт вліво, додавши справа новий байт із повідомлення
Виконаємо операцію XOR вмісту регістру з сумованим поліномом
End

Поки що наведений алгоритм не є кращим за "простий" прямий алгоритм. Але, якщо частину обчислень, як було вказано вище, провести наперед, а результати зібрати у таблицю, то маємо:

While (повідомлення повністю не оброблене)
Begin
Top = Старший_байт(Register);
Register = (зсув Register на 8(16…) біт вліво) **або** Новий_байт_повідомлення;
Register = Register XOR Розрахована_таблиця[Top];
End

Отриманий алгоритм вже став ефективним алгоритмом, він вимагає тільки операцій зсуву, OR, XOR, а також операції пошуку в таблиці (не враховуючи попередні обчислення в таблиці). Графічно схема алгоритму виглядає як на рис. 4.2.
Отже, операції зсуву на 1 байт, доповнення регісту справа новим байтом з повідомлення, XOR табличного значення із вмістом регістру виконуються за константний час кожна. Залишаються ще операції пошуку у наперед сформованій таблиці.

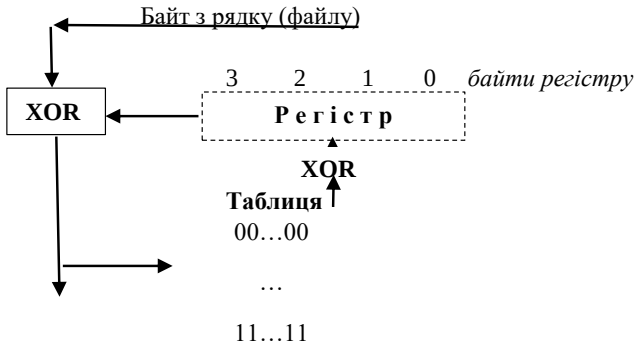

Рис. 4.2. Схема алгоритму [7].

Тому розглянемо як формують таблицю. Розмір таблиці залежить від розрядності КС. Наприклад, маємо утворюючий поліном восьмого порядку, тобто розрядність КС – $N = 8$.
Складається таблиця чисел $2^N \times N$ (залишків від ділення на поліном), тому для восьмирозрядної КС матимемо таке:

Адреса в таблиці	Дані в таблиці, ділення за mod 2
1	Залишок від ділення числа 10000000 на поліном
10	Залишок від ділення числа 100000000 на поліном
11	Залишок від ділення числа 110000000 на поліном
…..	……
1111 1111	Залишок від ділення числа 1111111100000000 на поліном

Звідси бачимо, недоліком є те, що для КС з великим числом розрядів буде потрібно великий обсяг пам'яті. Тому вважають доцільним використання таблиць при $N \leq 16$.

В результаті на практиці, наприклад, для **CRC-16** застосовують подібний *алгоритм*:

1. Висувається старший байт регістру у байтову комірку.
2. Виконується XOR над висунутим байтом регістру та байтом рядку-повідомлення.
3. Одержане значення є індексом для доступу до наперед створеної таблиці.
4. Узяте з таблиці значення об'єднується по XOR із значенням у регістрі та результат заноситься у регістр.
5. До закінчення обробки останнього байту рядку виконуються кроки 3-4, після обробки останнього байту рядку у регістрі буде міститися CRC.

Дзеркальні CRC алгоритми [7]. Найпростіший з них – послідовний, в якому надсилаються біти починаючи найменшого і закінчуючи найстаршим, тобто у зворотному порядку. У «дзеркальному» регістрі всі біти відображаються відносно центру. Замість того, щоб міняти місцями біти перед їх обробкою, дзеркально відображають всі значення та дії, що беруть участь у прямому алгоритмі. При цьому напрямок розрахунків змінюється – байти зсуваються направо, а також додається дзеркальне відображення поліному (рис. 4.3). Для дзеркального табличного алгоритму таблиця стане дзеркальним відображенням CRC-таблиці прямого алгоритму.

Отже, для дзеркального послідовного алгоритму (наприклад, CRC-32) *процес обчислення аналогічний, але інший порядок зсувів та заповнення*

регістру (заповнюється вправо). Проміжний CRC для першої дії може бути 16 бітів. Байти рядку розглядаються *без дописування нулів* у кінці. Дзеркальний табличний алгоритм (CRC-32) може мати вигляд:

1. Заповнюємо регістр та висувається старший байт регістру у байтову комірку (на 1 байт вправо).
2. Виконується XOR над висунутим байтом регістру та правим байтом рядку-повідомлення.
3. Одержане значення є індексом для доступу до наперед створеної таблиці.
4. Узяте з таблиці значення об'єднується по XOR із значенням у регістрі та результат заноситься у регістр.
5. До закінчення обробки останнього байту рядку виконуються кроки 3-4, після обробки останнього байту рядку у регістрі буде міститися CRC.

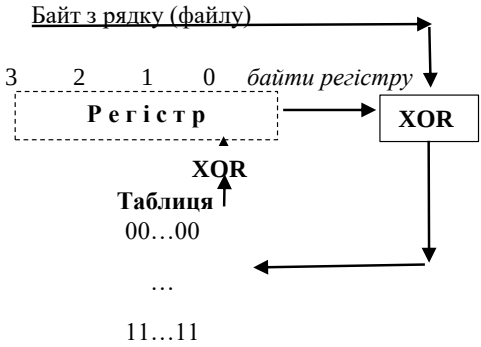

Рис. 4.3. Схема оберненого табличного алгоритму.

Для прямих та дзеркальних алгоритмів на практиці використовують також різні додаткові ускладнення. Наприклад, можливе повернення старших бітів й перемішування з молодшими, додавання віртуального байту, помноженого на деяку емпіричну сталу.

При виборі конкретного CRC алгоритму важливим моментом є врахування типів помилок, що очікувані при передачі повідомлень. Найпростішими типами помилок, які найбільш очікувані при передачі є [7]:

- *Помилка в одному біті.* E = 100...000. Такі помилки можна відловити якщо у поліномі не менше 2-х бітів встановлено в 1. Як би не складали зсуви таких поліномів, завжди мали б рядок, в якому хоча б 2 одиничних біта, а значить E не може бути кратним поліному.
- *Помилка в двох бітах.* Для усунення таких помилок підбирають поліноми, для яких не є кратними числа типу 011, 0101, 01001 і т.д.
- *Помилка у непарній кількості біт.* Такі похибки можна відловити, якщо взяти поліном, в якому кількість біт парна.
- *Помилка пакета.* E = 000...000111...111000...000. Тобто похибка локалізована у неперервному пакеті у повідомленні. Таке E можна представити у виді добутку: E=(1000.....00)*(111111111). Якщо в поліномі молодший біт «1», то лівий множник не може бути дільником поліному, тоді, якщо поліном довший за правий множник, то похибка буде відловлена.

Розроблені для практичного застосування CRC-алгоритми використовують наступні стандартизовані специфікації, що встановлюють повну характеристику конкретного алгоритму (табл. 4.1):

- Назва алгоритму;
- Степінь породжуючого КС багаточлена (width);
- Породжуючий поліном. Його спочатку записують як бітову послідовність (при цьому старший біт випускають – він завжди 1). Наприклад, багаточлен буде записаний числом . Для зручності двійкове представлення записують у шістнадковій формі (або 0x11);
- Стартові дані (init), тобто значення регістрів на момент початку обчислень;
- Прапорець (RefIn), що вказує на початок і напрямок обчислень, для виявлення пакетів помилок має відповідати порядку передачі у каналі. Існує два варіанти: False – починаючи із старшого значущого біту, True – з молодшого;
- Прапорець (RefOut), вказує чи інвертується порядок бітів регістру при вході на елемент XOR;
- Число (XorOut), з яким складається за mod2 одержаний результат;
- Значення CRC (check) для рядку «123456789».

Табл. 4.1. Специфікації поширених CRC алгоритмів.

Назва	Width	Поліном	Init	RefIn	RefOut	XorOut	Check
CRC-3/ROHC	3	0x3	0x7	true	true	0x0	0x6
CRC-4/ITU	4	0x3	0x0	true	true	0x0	0x7
CRC-5/EPC	5	0x9	0x9	false	false	0x0	0x0
CRC-5/ITU	5	0x15	0x0	true	true	0x0	0x7
CRC-5/USB	5	0x5	0x1F	true	true	0x1F	0x19
CRC-6/CDMA2000-A	6	0x27	0x3F	false	false	0x0	0xD
CRC-6/CDMA2000-B	6	0x7	0x3F	false	false	0x0	0x3B
CRC-6/DARC	6	0x19	0x0	true	true	0x0	0x26
CRC-6/ITU	6	0x3	0x0	true	true	0x0	0x6
CRC-7	7	0x9	0x0	false	false	0x0	0x75
CRC-7/ROHC	7	0x4F	0x7F	true	true	0x0	0x53
CRC-8	8	0x7	0x0	false	false	0x0	0xF4
CRC-8/CDMA2000	8	0x9B	0xFF	false	false	0x0	0xDA
CRC-8/DARC	8	0x39	0x0	true	true	0x0	0x15
CRC-8/DVB-S2	8	0xD5	0x0	false	false	0x0	0xBC
CRC-8/EBU	8	0x1D	0xFF	true	true	0x0	0x97
CRC-8/I-CODE	8	0x1D	0xFD	false	false	0x0	0x7E
CRC-8/ITU	8	0x7	0x0	false	false	0x55	0xA1
CRC-8/MAXIM	8	0x31	0x0	true	true	0x0	0xA1
CRC-8/ROHC	8	0x7	0xFF	true	true	0x0	0xD0
CRC-8/WCDMA	8	0x9B	0x0	true	true	0x0	0x25
CRC-10	10	0x233	0x0	false	false	0x0	0x199
CRC-10/CDMA2000	10	0x3D9	0x3FF	false	false	0x0	0x233
CRC-11	11	0x385	0x1A	false	false	0x0	0x5A3
CRC-12/3GPP	12	0x80F	0x0	false	true	0x0	0xDAF
CRC-12/CDMA2000	12	0xF13	0xFFF	false	false	0x0	0xD4D
CRC-12/DECT	12	0x80F	0x0	false	false	0x0	0xF5B
CRC-13/BBC	13	0x1CF5	0x0	false	false	0x0	0x4FA
CRC-14/DARC	14	0x805.	0x0	true	true	0x0	0x82D
CRC-15	15	0x4599	0x0	false	false	0x0	0x59E
CRC-15/MPT1327	15	0x6815	0x0	false	false	0x1	0x2566

CRC-16/ARC	16	0x8005	0x0	true	true	0x0	0xBB3D
CRC-16/AUG-CCITT	16	0x1021	0x1D0F	false	false	0x0	0xE5CC
CRC-16/BUYPASS	16	0x8005	0x0	false	false	0x0	0xFEE8
CRC-16/CCITT-FALSE	16	0x1021	0xFFFF	false	false	0x0	0x29B1
CRC-16/CDMA2000	16	0xC867	0xFFFF	false	false	0x0	0x4C06
CRC-16/DDS-110	16	0x8005	0x800D	false	false	0x0	0x9ECF
CRC-16/DECT-R	16	0x589	0x0	false	false	0x1	0x7E
CRC-16/DECT-X	16	0x589	0x0	false	false	0x0	0x7F
CRC-16/DNP	16	0x3D65	0x0	true	true	0xFFFF	0xEA82
CRC-16/EN-13757	16	0x3D65	0x0	false	false	0xFFFF	0xC2B7
CRC-16/GENIBUS	16	0x1021	0xFFFF	false	false	0xFFFF	0xD64E
CRC-16/MAXIM	16	0x8005	0x0	true	true	0xFFFF	0x44C2
CRC-16/MCRF4XX	16	0x1021	0xFFFF	true	true	0x0	0x6F91
CRC-16/RIELLO	16	0x1021	0xB2AA	true	true	0x0	0x63D0
CRC-16/T10-DIF	16	0x8BB7	0x0	false	false	0x0	0xD0DB
CRC-16/TELEDISK	16	0xA097	0x0	false	false	0x0	0xFB3
CRC-16/TMS37157	16	0x1021	0x89EC	true	true	0x0	0x26B1
CRC-16/USB	16	0x8005	0xFFFF	true	true	0xFFFF	0xB4C8
CRC-A	16	0x1021	0xC6C6	true	true	0x0	0xBF05
CRC-16/KERMIT	16	0x1021	0x0	true	true	0x0	0x2189
CRC-16/MODBUS	16	0x8005	0xFFFF	true	true	0x0	0x4B37
CRC-16/X-25	16	0x1021	0xFFFF	true	true	0xFFFF	0x906E
CRC-16/XMODEM	16	0x1021	0x0	false	false	0x0	0x31C3
CRC-24	24	0x864CFB	0xB704CE	false	false	0x0	0x21CF02
CRC-24/FLEXRAY-A	24	0x5D6DCB	0xFEDCBA	false	false	0x0	0x7979BD
CRC-24/FLEXRAY-B	24	0x5D6DCB	0xABCDEF	false	false	0x0	0x1F23B8
CRC-31/PHILIPS	31	0x4C11DB7	0x7FFFFFFF	false	false	0x7FFFFFFF	0xCE9E46C
CRC-32/zlib	32	0x4C11DB7	0xFFFFFFFF	true	true	0xFFFFFFFF	0xCBF43926
CRC-32/BZIP2	32	0x4C11DB7	0xFFFFFFFF	false	false	0xFFFFFFFF	0xFC891918
CRC-32C	32	0x1EDC6F41	0xFFFFFFFF	true	true	0xFFFFFFFF	0xE3069283
CRC-32D	32	0xA833982B	0xFFFFFFFF	true	true	0xFFFFFFFF	0x87315576
CRC-32/MPEG-2	32	0x4C11DB7	0xFFFFFFFF	false	false	0x0	0x376E6E7
CRC-32/POSIX	32	0x4C11DB7	0x0	false	false	0xFFFFFFFF	0x765E7680
CRC-32Q	32	0x814141AB	0x0	false	false	0x0	0x3010BF7F
CRC-32/JAMCRC	32	0x4C11DB7	0xFFFFFFFF	true	true	0x0	0x340BC6D9
CRC-32/XFER	32	0xAF	0x0	false	false	0x0	0xBD0BE338

CRC називають *n-бітним CRC*, якщо його контрольне значення дорівнює n-бітам. Для заданого n можливі декілька CRC, кожен з різним поліномом. Такий багаточлен має найвищу степінь n, його довжина n + 1. Залишок має довжину n. Кожен алгоритм CRC має ім'я у формі CRC-n-XXX (наприклад, CRC-8-CCIT, CRC-5-USB, CRC-8-GSM, але є й типу CRC-32C (Castagnoli), CRC-8-Dallas/Maxim).

Дізайн поліному CRC залежить від максимальної спільної довжини блоку, який має бути захищений (дані + біти CRC), бажаних функцій захисту від похибок та типу ресурсів для реалізації CRC, а також бажаної продуктивності. Часто вважають, що «кращі» поліноми CRC одержуються або з непривідних поліномів, або з непривідних поліномів, помножених на 1 + x, що додає до коду можливість виявляти всі похибки, що впливають на непарну кількість бітів. Насправді вказані фактори мають входити у вибір поліному й можуть привести до привідного поліному. Але вибір привідного поліному приведе до деякої частини пропущених похибок.

***Умови лабораторної роботи*:**

Реалізувати алгоритми для обчислення CRC із заданим за варіантом утворюючим поліномом (табл. 4.2).

Для вхідного повідомлення довжиною K =1000 (сгенероване випадково з 0 та 1) обчислити CRC за допомогою реалізованих наступних алгоритмів:

1. Простий послідовний алгоритм.
2. Табличний алгоритм.
3. Дзеркальний послідовнй алгоритм.
4. Дзеркальний табличний алгоритм.
5. Стандартизований алгоритм за назвою (додатково за бажанням, див. табл. 4.1).

Провести порівняльний аналіз реалізацій алгоритмів за їх ресурсною складністю. Оцінити середній час обчислення CRC за кожним з алгоритмів для довільно узятої досить великої кількості експериментів.

Таблиця 4.2. Варіанти утворюючих поліномів

Номер варіанту	Назва	Утворюючий поліном	Використання
1	CRC-16 (CRC-16-IBM, CRC-16- ANSI)	$x^{16}+x^{15}+x^2+1$	Bisyne, Molbus, USB, ANSI X3.28 та ін.
2	CRC-8-CCIT	x^8+x^2+x+1	ATM HEC, ISDN Header Error Control and Cell Delineation ITU-T I.432.1(02/99)
3	CRC-24	$x^{24}+x^{22}+x^{20}+x^{19}+x^{18}+x^{16}+x^{14}+x^{13}+x^{11}+x^{10}+x^8+x^7+$ $+x^6+x^3+x+1$	FlaxRay
4	CRC-16-T10-DIF	$x^{16}+x^{15}+x^{11}+x^9+x^8+x^7+x^5+ x^4+x^2+x+1$	SCSI DIF
5	CRC-5-USB	x^5+x^2+1	USB token packets
6	CRC-32C (Castagnoli)	$x^{32}+x^{28}+x^{27}+x^{26}+x^{25}+x^{23}+x^{22}+x^{20}+x^{19}+x^{18}+x^{14}+x^{13}+$ $+x^{11}+x^{10}+x^9+ x^8+x^6 +1$	iSCSI, G.Hn payload
7	CRC-10	$x^{10}+x^9+ x^5+x^4 +x+1$	
8	CRC-24-Radix-64	$x^{24}+x^{23}+x^{18}+x^{17}+x^{14}+x^{11}+x^{10}+x^7+x^6+x^5+x^4+x^3+x+1$	OpenPGP
9	CRC-8-Dallas/Maxim	$x^8+x^5+x^4+1$	1-Wire bus
10	CRC-32K (Koopman)	$x^{32}+x^{30}+x^{29}+x^{28}+x^{26}+x^{20}+x^{19}+x^{17}+x^{16}+x^{15}+x^{11}+x^{10}+$ $+x^7+x^6+x^4+ x^2+x+1$	
11	CRC-15-CAN	$x^{15}+x^{14}+x^{10}+x^8+x^7+x^4+x^3+1$	
12	CRC-4-ITU	x^4+x+1	ITU G.704
13	CRC-32Q	$x^{32}+x^{31}+x^{24}+x^{22}+x^{16}+x^{14}+x^8+x^7+x^5+x^3+x+1$	aviation, AIXM
14	CRC-16-CCITT	$x^{16}+x^{12}+x^5+1$	X.25, HDLC, XMODEM, Bluetooth, SD та ін
15	CRC-8-GSM	$x^8+x^6+x^3+1$	моб. мережі
16	CRC-8	$x^8+x^7+x^6+x^4+x^2+1$	ETSI EN 302 307
17	CRC-32-IEEE 802.3	$x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^8+x^7+x^5+x^4+$ $+x^2+x+1$	V.42, MPEG-2, PNG, POSIX
18	CRC-6-ITU	$x^6+ x+1$	ITU G.
19	CRC-11	$x^{11}+x^9+x^8+x^7+x^2+1$	FlaxRay
20	CRC-16-CCIT	$x^{16}+x^{12}+x^5+1$	X25, HDLC, XMODEM, Bluetooth, SD та ін.
21	CRC-5-EPC	x^5+x^3+1	Gen 2 RFID
22	CRC-31/PHILIPS	$x^{31}+x^{241}+x^{21}+x^{20}+x^{14}+x^{10}+x^9+x^8+x^6+x^5+x^3+x^2+1$	
23	CRC-16-DNP	$x^{16}+x^{13}+x^{12}+x^{11}+x^{10}+ x^8+x^6+x^5+x^2+1$	DNP, IEC 870, M-Bus

24	CRC-7	x^7+x^3+1	системи телекомунікації, ITU-T G.707, ITU-T G.832, MMC, SD
25	CRC-16/ARX	$x^{16}+x^{15}+x^3+1$	1-Wire bus
26	CRC-8-SAE JI850	$x^8+x^4+x^3+x^2+1$	
27	CRC-30	$x^{30}+x^{29}+x^{21}+x^{20}+x^{15}+x^{13}+x^{11}+x^8+x^7+x^6+x^2+x+1$	CDMA
28	CRC-5-ITU	$x^5+x^4+x^2+1$	ITU-T G.704
29	CRC-12	$x^{12}+x^{11}+x^3+x^2+x+1$	системи телекомунікації
30	CRC-8-WCDMA	$x^8+ x^7+x^4+x^3+x+1$	моб. мережі
31	CRC-8-Bluetooth	$x^8+ x^7+x^5+x^2+x+1$	бездротова мережа
32	CRC-10	$x^{10}+x^9+x^5+x^4+x+1$	банкомат; ITU-T I.610
33	CRC-16-DECT	$x^{16}+ x^{10}+x^8+ x^7+x^3+x+1$	бездротові телефони

Лабораторна робота 5 (додаткова). Ідеальне хешування

***Умови лабораторної роботи*:**

Нехай задана деяка архітектура комп'ютера, в межах якої необхідно скласти процедуру, що дає ідеальних хеш для заданої множини чисел.

Комп'ютер має $4 \cdot 2^{20}$ байт пам'яті, що доступна тільки для читання, та містить програму, яку буде виконувати комп'ютер, а також дані. Процесор його містить 256 регістрів (від r0 до r255), причому кожен може зберегти 32-бітне ціле число. Перед виконанням процедури значення кожного регістру крім r0 дорівнює 0, r0 містить вхідні дані процедури.

Комп'ютер має спеціальний регістр – instruction pointer (знаходиться окремо). Процедура зберігається в пам'яті та на початку значення цього регістру дорівнює також 0.

Виконання процедури проходить наступним чином. Спершу зчитується номер інструкції, що зберігається в пам'яті за адресою значення instruction pointer, потім зчитуються аргументи інструкції та виконується інструкція. Якщо під час виконання інструкції не трапився jump,

то значення instruction pointer збільшується на розмір прочитаної інструкції, включаючи аргументи.

Процедура має бути відображенням з множини деяких заданих n цілих чисел $A=\{a_0,\ldots,a_n\}$ у множину цілих чисел від 0 до $2n-1$. Процедура має завершатися після не більше ніж 30 виконаних інструкцій (за будь-яких вхідних даних з множини A процедура не повинна виконувати ділення на 0, а також намагатися звертатися за межі діапазону $4\cdot 2^{20}$ байтів пам'яті). Відображення не повинно мати колізій.

Тому, відповідно до архітектури комп'ютера, має виконуватися:

- якщо процедурі на вхід подати одне з чисел з множини A, вона має завершитися за не більш ніж 30 інструкцій (таблиця інструкцій – табл. 5.1) та вивести ціле число від 0 до $2n-1$;
- для будь-яких двох різних чисел a_i та a_j з множини A значення на виході повинні бути різними.

Крім того:

– всі регістри містять 32-бітні цілі числа;

– порядок збереження для всіх чисел: від молодших байт до старших;

– для додавання, віднімання та бітових операцій не суттєво, числа знакові чи ні;

– команди jump з умовою представляють числа як знакові;

– результатом множення або ділення є знакове 64-бітне число, що зберігається у два регістри, регістр, що одержує молодшу частину результату зберігає його як беззнакове число;

– ділення та одержання залишку беруть ділене в якості 64-бітних чисел, де молодша частина зберігається як беззнакове число, а старша – із знаком; дільник є знаковим числом, ділення та одержання залишку для знакових чисел відбувається аналогічно архітектурі x86.

Вхідні дані та їх формат:

- Перший рядок містить ціле число n ($1 \le n \le 1\cdot 10^5$);
- Другий рядок містить n різних цілих чисел $a_0,\ldots,a_n$ ($1 \le n \le 1\cdot 10^9$).

Вихідні дані та їх формат:

Це дані, що містяться в пам'яті комп'ютера. Можна вивести будь-яку кількість байт – від 1 до 4194304 (виводимо тільки ненульову частину). Частина пам'яті, що залишилася, заповниться нулями. Кожен біт має бути виведений як число у системі счислення за основою 16 (тобто дві цифри). Значення виводяться без пробілів.

Таблиця 5.1. Таблиця інструкцій

Назва	Розмір байт	Opcode	Зміст інструкції
nop	1	00	пуста інструкція, нічого не відбувається
add	4	01 r1 r2 r3	r3:= r1 + r2
sub	4	02 r1 r2 r3	r3:= r1 – r2
mul	5	03 r1 r2 r3 r4	(r3, r4):= r1 * r2 (r3 містить молодші біти добутку, r4 – старші)
div	6	04 r1 r2 r3 r4 r5	(r3, r4):= (r1, r5) div r2 (r1 містить молодші біти діленого, r5 – старші; r3 містить молодші біти результату, r4 – старші)
mod	5	05 r1 r2 r3 r4	r3:= (r1, r4) mod r2 (r1 містить молодші біти діленого, r4 – старші)
and	4	10 r1 r2 r3	r3:= r1 and r2
or	4	11 r1 r2 r3	r3:= r1 or r2
xor	4	12 r1 r2 r3	r3:= r1 xor r2
neg	2	20 r1	r1:= - r1
not	2	21 r1	r1:= ¯r1
load	3	30 r1 r2	r1:= memory[r2] (4 байти, починаючи з адреси r2 копіюються до регістру r1, першим копіюється молодший байт)
put	6	31 r1 b0 b1 b2 b3	r1:= (b0, b1, b2, b3) (b0 – молодший байт числа, що покладене до регістру, b3 – старший)
jmp	5	40 b0 b1 b2 b3	виконується jump до інструкції (b0, b1, b2, b3)
jz	6	41 r1 b0 b1 b2 b3	виконується jump якщо r1 == 0 до інструкції (b0, b1, b2, b3)
jnz	6	42 r1 b0 b1 b2 b3	виконується jump якщо r1 ¬= 0 до інструкції (b0, b1, b2, b3)
jg	6	43 r1 b0 b1 b2 b3	виконується jump якщо r1 >0 до інструкції (b0, b1, b2, b3)
jge	6	44 r1 b0 b1 b2 b3	виконується jump якщо r1 >= 0 до інструкції (b0, b1, b2, b3)
jl	6	45 r1 b0 b1 b2 b3	виконується jump якщо r1 < 0 до інструкції (b0, b1, b2, b3)
jle	6	46 r1 b0 b1 b2 b3	виконується jump якщо r1 <= 0 до інструкції (b0, b1, b2, b3)

ret	1	ff	завершується процедура, вихідні дані містяться у регістрі r0

Тестові результати

Вхідні дані	Виведені біти
3 2 3 9	3101040000000500010003ff

Рекомендована література

[1]. Ульянов М. В. Ресурсноэффективные компьютерные алгоритмы / М. В. Ульянов. – М.: Наука, 2007. – 376 с.

[2]. Кормен Т. Алгоритмы. Построение и анализ. 3-е изд. / Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн. – М. : ИД "Вильямс", 2013. – 1328 с.

[3]. Кнут Д. Искусство программирования, том 3. Сортировка и поиск. 3-е изд. / Д. Кнут. – М.: Вильямс, 2006. – С. 822.

[4]. Седжвик Р. Алгоритмы на C++. Фундаментальные алгоритмы и структуры данных / Р. Седжвик. – М. : ИД "Вильямс", 2011. – 1056 с.

[5]. Панос Л. Алгоритмы для начинающих: Теория и практика для разработчика. / Л. Панос. – М.: ЭКСМО, 2018. – 608 с.

[6]. Блейхут Р. Теория и практика кодов, контролирующих ошибки / Р. Блейхут. – М. : Мир, 1986. – 576 с.

[7]. Ross N. W. A painless guide to CRC error detection algorithms. – 1993. – 90 с. Ел. ресурс. Режим доступу: https://ceng2.ktu.edu.tr/~cevhers/ders_materyal/bil311_bilgisayar_mimarisi/supplementary_docs/crc_algorithms.pdf.

ЗМІСТ

Навчальне видання

Вергунова І.М.,

Основи комп'ютерних алгоритмів: методичні вказівки до виконання лабораторних робіт.

Підписано до друку 21.08.2021.
Формат 60х84/16. Папір офсетний.
Друк цифровий.
Друк. арк. 2,5. Умов. друк. арк. 2,3.
Обл.-вид. арк. 1,5.
Наклад 100 прим. Зам. № 5376/1.
Віддруковано з оригіналів замовника.
ФОП Корзун Д.Ю.
Свідоцтво про державну реєстрацію фізичної особи-підприємця
серія В02 № 818191 від 31.07.2002 р.

Видавець та виготовлювач ТОВ «ТВОРИ».
Свідоцтво про внесення суб'єкта видавничої справи
до Державного реєстру видавців, виготовлювачів і розповсюджувачів
видавничої продукції серія ДК № 6188 від 18.05.2018 р.
21034, м. Вінниця, вул. Немирівське шосе, 62а.
Тел.: 0 (800) 33-00-90, (096) 97-30-934, (093) 89-13-852,
(098) 46-98-043.
e-mail: info@tvoru.com.ua
http://www.tvoru.com.ua