С.Д.Паращук, І.В.Лупан

ПРАКТИКУМ
із ТЕОРІЇ АЛГОРИТМІВ
Частина І

Навчально-методичний посібник
для студентів закладів вищої освіти

Кропивницький 2025

С.Д.Паращук, І.В.Лупан

ПРАКТИКУМ
із ТЕОРІЇ АЛГОРИТМІВ
Частина І

Навчально-методичний посібник
для студентів закладів вищої освіти

Кропивницький 2025

УДК 004.421

Рецензенти:

О.В.Авраменко, доктор фізико-математичних наук, професор кафедри математики Національного університету «Києво-Могилянська академія»

В.В.Нарадовий, кандидат технічних наук, доцент кафедри інформатики, програмування, штучного інтелекту та технологічної освіти Центральноукраїнського державного університету імені Володимира Винниченка.

Рекомендовано рішенням кафедри інформаційних технологій Економіко-технологічного інституту імені Роберта Ельворті (протокол № 6 від 23.01.25 р.)

Паращук С.Д., Лупан І.В.

Практикум із теорії алгоритмів. Частина І. Навчально-методичний посібник. [Електронне видання]. Кропивницький: ЕТІ імені Роберта Ельворті, 2025, 110 с.

У посібнику подані матеріали для проведення практичних занять, що охоплюють основи алгоритмізації, формальні моделі алгоритмів, обчислювальну складність алгоритмів. До кожного заняття подані необхідні теоретичні відомості, велика кількість прикладів розв’язання задач, завдання для індивідуальної роботи.

Посібник призначений для студентів, які навчаються за галуззю знань «Інформаційні технології».

Зміст

Тема 1: Поняття алгоритму. Лінійні алгоритми

Теоретичні відомості

Існують різні змістовні тлумачення поняття алгоритму. Наведемо одне з них.

Алгоритм – це деякий скінченний набір дій, виконання яких одна за однією через скінчене число кроків приводить до поставленої мети (розв'язку задачі).

Алгоритм характеризується такими властивостями:

1) масовість,
2) визначеність, (детермінованість),
3) дискретність,
4) результативність,
5) формальність.

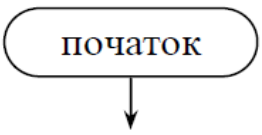
Алгоритми можна записувати в різних формах. Ці форми в значній мірі визначаються тим, на якого виконавця розрахований алгоритм. Можна виділити такі форми подання алгоритмів:

- словесний запис алгоритму;
- словесно-формульна форма;
- графічна форма;
- псевдокоди;
- мови програмування.

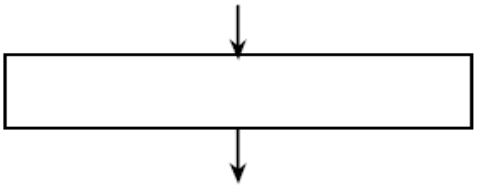
При графічному поданні алгоритмів часто використовують блок-схеми – це графічне зображення блоків, зв'язаних між собою лініями і стрілками. Лінії і стрілки позначають послідовність обчислень, а кожний блок відповідає одному крокові алгоритму і містить опис відповідної дії.

Блок-схеми будують відповідно до міждержавного стандарту ISO 5807-85 (ГОСТ 19.701-90) «Єдина система програмної документації. Схеми алгоритмів, програм, даних і систем».

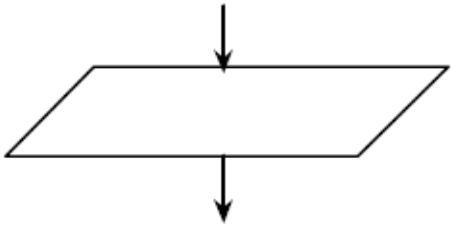
Блок-схема має містити по одному блоку початку (входу) й кінця (виходу) алгоритму:



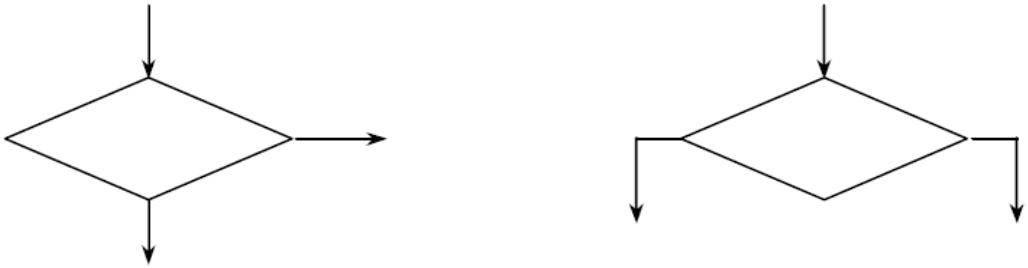
Прямокутниками зображають виконувані алгоритмом обчислювальні дії. Стрілки використовуються для зображення входу і виходу з блоку.



Паралелограмами зображають блоки введення-виведення.



Логічні блоки зображають ромбами. Всередині ромбу записують умову, яка може виконуватися або ні. На підставі цього визначають два можливих варіанти подальших обчислень.



Існують ще інші елементи блок-схем, однак ми їх не використовуватимемо.

У 1966 році італійські математики Коррадо Бьом і Джузеппе Якопіні в своїй статті опублікували структурну теорему, яка лежить в основі парадигми структурного програмування. Відповідно до цієї теореми, будь-який алгоритм можна задати з використанням лише трьох конкретних структур керування (базових алгоритмічних конструкцій): слідування, розгалуження і повторення.

Слідування. Ця конструкція подається у вигляді послідовності двох (або більше) виконуваних одна за однією дій D_1, D_2, …, D_N для опрацювання інформації (рис. 1.1).

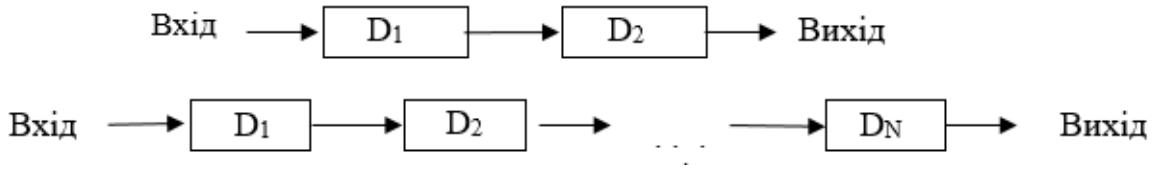


Рис. 1.1. Структура слідування

Дії D_1, D_2, …, D_N – це або окремі вказівки, наприклад, вказівка присвоювання або виведення, або будь-яка можлива скінченна послідовність вказівок для опрацювання інформації. Кожна дія має один вхід (подається вхідна інформація) і один вихід (результат опрацювання інформації за допомогою дії). Стрілки також визначають напрям виконання дій. Конструкція слідування дозволяє зображати складну дію у вигляді більш простіших дій або навпаки замінювати послідовність простіших дій на одну складнішу дію.

Розгалуження (вибір). Ця конструкція служить для вибору одної із двох можливих дій, в залежності від деякої умови (рис. 1. 2).

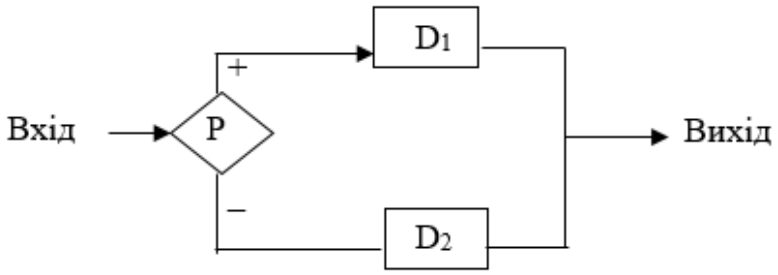


Рис. 1.2. Структура розгалуження.

Конструкція розгалуження виконується так. Спочатку обчислюється умова P, яка може приймати два логічні значення: істина або хибність. Якщо умова P виявиться істинною, то управління передається в напрямку стрілки «+», тобто виконується дія D_1. Якщо ж умова P виявиться хибною, то буде виконуватись дія D_2.

Окремим випадком розгалуження є неповне розгалуження (корекція), яка зображена на рис 1.3.

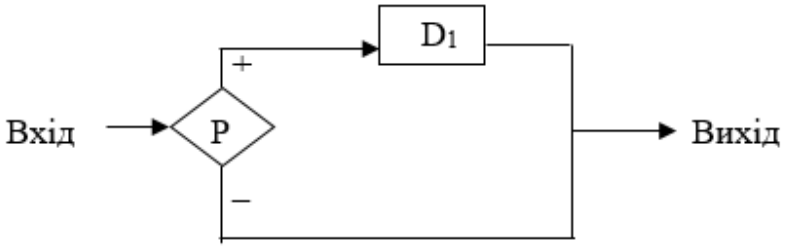


Рис. 1.3. Неповне розгалуження.

На відміну від розгалуження в неповному розгалуженні при хибності умови P не виконується ніякої дії.

Повторення (цикл). Конструкції повторення використовуються для позначення багатократно повторюваних дій. Повторення ще називають циклами. Розрізняють два види циклів – цикл-ПОКИ (з передумовою) і цикл-ДО (з після умовою).

Структура циклу-ПОКИ зображена на рис. 1.4.

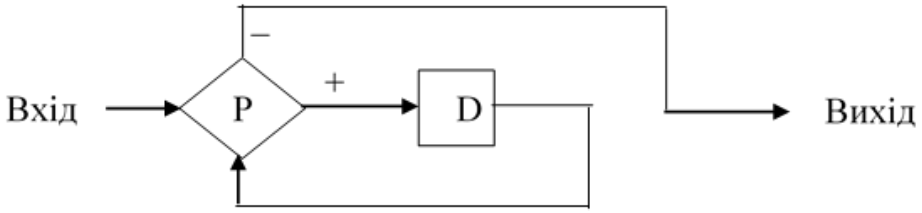


Рис. 1. 4. Структура циклу-ПОКИ.

Конструкція циклу –ПОКИ працює так. Спочатку здійснюється перевірка умови P – чи є вона істинною чи хибною. Якщо P істинна, то виконуємо дію D (рухаємось за стрілкою зі знаком»+») і знову повертаємось до перевірки умови P. Виконання дії D буде повторюватись до тих пір, поки умова P не стане хибною. Як тільки P стане хибною, то управління передається в напрямку стрілки зі знаком «–», тобто виконання дії D припиняється і кажуть, що цикл припинив свою роботу.

Дія D може не виконатись жодного разу, виконатись скінчену кількість разів або виконуватись нескінченну кількість разів. Дійсно, якщо з самого початку умова P виявиться хибною, то дія D не виконається жодного разу, бо управління буде передане в напрямку стрілки «–». Дія D буде виконуватись скінчену кількість разів, якщо ж таку кількість разів умова P буде істинною, а потім стане хибною. Якщо умова P буде завжди істинною, то дія D буде виконуватись нескінченну кількість разів. У цьому випадку кажуть, що цикл-ПОКИ побудований некоректно і такі варіанти циклу в алгоритмі не використовуються.

Структура циклу-ДО зображена на рис. 1.5.

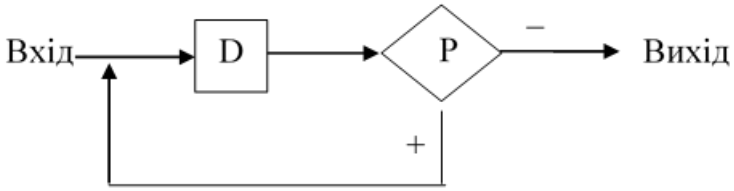


Рис. 1.5. Структура циклу ДО.

У циклі-ДО спочатку виконується дія D. Потім здійснюється перевірка умови P – чи істинна вона чи хибна. Якщо P істинна, то управління передається за стрілкою «+» і знову виконується дія D. Виконання дії D буде повторюватися до тих пір, поки умова P залишатиметься істинною. Як тільки умова стане хибною, управління буде передане за стрілкою «-» і цикл припинить свою роботу.

У циклі-ДО дія D буде виконуватись скінчену кількість разів або нескінченну. В останньому випадку вважають, що цикл побудовано некоректно.

Як бачимо, в циклі-ПОКИ спочатку перевіряється умова P, а потім виконується дія D. А в циклі-ДО спочатку виконується дія D, а потім перевіряється умова P. Тому в першому циклі дія D може не виконатись жодного разу, а в другому – обов’язково хоч раз виконається.

Важливою особливістю розглянутих структур є те, що кожна з них має єдиний вхід і єдиний вихід. При конструюванні алгоритму вихід кожної базової структури приєднується до входу іншої, таким способом весь алгоритм подається у вигляді лінійної послідовності базових алгоритмічних структур.

Алгоритми, у яких використовується тільки структура «слідування» називають лінійними. Алгоритми, в основі яких лежить структура «розгалуження», називають алгоритмами з розгалуженням. Алгоритми, в основі яких лежить структура «повторення», називають циклічними. На практиці, зазвичай, алгоритми містять усі три типи базових структур.

Приклад 1

Дано площу круга. Знайти сторону правильного трикутника, вписаного в коло. Розробити алгоритм розв’язання задачі та подати його в текстовій формі та у вигляді блок-схеми.

Аналіз задачі

Площа круга обчислюється за формулою $S = \pi * R^2$, а сторона правильного трикутника, вписаного в коло, дорівнює $a = R * \sqrt{3}$. Отже, з першої формули треба обчислити $R = \sqrt{S/\pi}$ і підставити результат у другу формулу.

Текстова форма

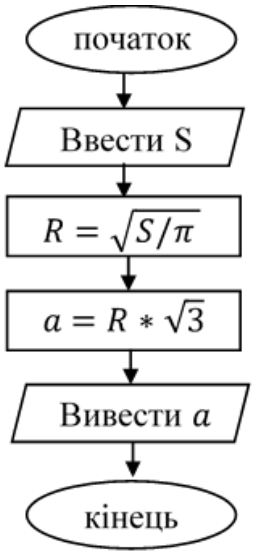
К1. Ввести площу S.

К2. Обчислити $R = \sqrt{S/\pi}$.

К3. Обчислити $a = R * \sqrt{3}$.

К4. Вивести a.

Блок-схема алгоритму



Змінні величини для алгоритму:
S – дійсна змінна (дійсне число);
R – дійсна змінна (дійсне число);
a – дійсна змінна (дійсне число).

Задачі

Для вказаних задач розробити алгоритм їхнього розв'язання. Подати алгоритм в текстовій формі та у вигляді блок-схеми.

1. Дано додатні числа a, b. Обчислити
$$y = \sqrt[5]{log_a b} \cdot \sin a^b.$$
2. Дано координати вершин трикутника: A(x_1, y_1), B(x_2, y_2), C(x_3, y_3). Обчислити периметр і площу трикутника.
3. Дано x. Обчислити $2x^4-3x^3+4x^2-5x+6$ за найменшу кількість операцій.
4. Обчислити дробову частину середнього геометричного трьох даних додатних чисел.

☺ Індивідуальні завдання

Виконати аналіз вказаних задач та розробити алгоритм їхнього розв'язання. Подати алгоритм в текстовій формі та у вигляді блок-схеми.

А

1. Дано сторони трикутника. Обчислити його площу.
2. Дано висоту та радіус основи конуса. Обчислити його об'єм.
3. Дано дві сторони трикутника та кут між ними. Обчислити площу трикутника.

4. Дано сторони трикутника. Знайти його висоти.
5. Дано сторону ромба та гострий кут між сторонами. Знайти площу ромба.
6. Дано основи та висоту трапеції. Знайти її площу.
7. Дано висоту та сторону основи правильної чотирикутної піраміди. Обчислити її об'єм.
8. Дано висоту та сторону основи правильної трикутної призми. Обчислити її об'єм.
9. Дано сторони трикутника. Знайти його медіани.
10. Дано сторони трикутника. Знайти його бісектриси.

Б

1. Визначити **h** – повну кількість годин та **m** – повну кількість хвилин від початку доби до того моменту (у першій половині доби), коли стрілка, що показує години, повернулася на **f** градусів (0<=**f**<360, **f** – ціле число).
2. Нехай **k** – ціле число днів від 1 до 365. Присвоїти цілій змінній **n** значення 1,2,…,6 або 7 залежно від того, на який день тижня (понеділок, вівторок, …, суботу або неділю) припадає **k**-й день не високосного року, у якому 1 січня – це понеділок.
3. Цілій змінній **s** присвоїти суму цифр тризначного цілого числа **k**.
4. Дано ціле п'ятизначне число. Підрахувати суму цифр, які стоять на парних позиціях в числі, та добуток цифр, які стоять на непарних позиціях.
5. Триває **k**-та секунда доби. Визначити, скільки повних годин (**h**) та хвилин (**m**) пройшло до цього моменту (наприклад, якщо **k**=13257=3*3600+40*60+57, то **h**=3, **m**=40).
6. Паралелограм ABCD заданий координатами трьох його вершин: $A(x_1,y_1)$, $B(x_2,y_2)$, $C(x_3,y_3)$. Знайти довжини сторін та площу паралелограма.
7. Вартість товару **n** гривень. Яку найменшу кількість банкнот по 20, 10 та 1 гривні треба витратити на купівлю товару?
8. Визначити **f** – кут (у градусах) між положенням годинної стрілки на початку доби та її положенням у **h** годин, **m** хвилин та **s** секунд (0<=**h**<=11, 0<=**m**, **s**<=59).
9. Визначити номери під'їзду та поверху за номером квартири дев'ятиповерхового будинку, якщо на кожному поверсі рівно 4 квартири, а нумерація квартир починається з першого під'їзду.
10. Визначити нормальну вагу людини та індекс маси її тіла за формулами: **h·t/240** та $\mathbf{m/h^2}$, де **h** – зріст людини (у сантиметрах для першої формули та у метрах – для другої); **t** – довжина обхвату грудей (в см); **m** – вага (у кг). Індекс маси тіла прийнятий Всесвітньою організацією охорони здоров'я. Він не повинен перевищувати 25 пунктів.

Тема 2: Розгалуження

Теоретичні відомості

Алгоритмічна структура розгалуження розглядалася в темі 1.

Приклад 1

Дано число x. Обчислити y(x):

$$y = \begin{cases} 3^{x+1}, \text{якщо } x \le -1; \\ |x|, \text{якщо } -1 < x \le 0; \\ 0, \text{якщо } 0 < x \le 1; \\ log_{1/3}x, \text{в іншому випадку.} \end{cases}$$

Розробити алгоритм розв'язання задачі та подати його в текстовій формі та у вигляді блок-схеми.

Аналіз задачі

Функція y(x) задана на числовій прямій кусками. Потрібно задати число x та обчислити значення функції залежно від того, якій умові задовольняє x.

Текстова форма

К1. Ввести x.

К2. Якщо $x \le -1$, то обчислити $y = 3^{x+1}$ і перейти на К5.

К3. Якщо $x \le 0$, то обчислити $y = |x|$ і перейти на К5.

К4. Якщо $x \le 1$, то обчислити $y = 0$. Інакше обчислити $y = log_{1/3}x$.

К5. Вивести y.

Зауваження. У дії К3 замість умови $-1 < x \le 0$ написана умова $x \le 0$ тому, що при переході на цю дію вже $x > -1$. Аналогічно, в дії К4 замість умови $0 < x \le 1$ перевіряється умова $x \le 1$.

Змінні величини для алгоритму:

x – дійсна змінна (дійсне число);

y – дійсна змінна (дійсне число).

Зауваження. На блок-схемі знаком + (плюс) зображено виконання умови, записаної в середині ромба, а знаком – (мінус) – хибність цієї умови.

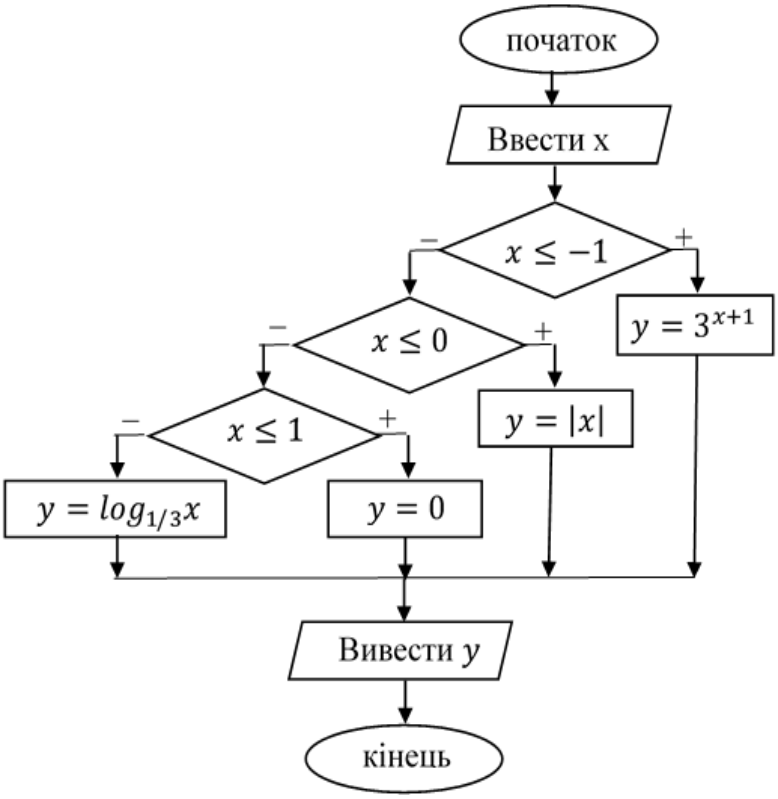


Рис. 2.1. Блок-схема алгоритму

Приклад 2

Дано ціле число x. Обчислити y(x):

$$y = \begin{cases} tg\ x + 3, x = -3; -2; -1 \\ 2\cos(2x)\ , x = 2 \\ x^x, x = 3\ ; 4 \\ 1001, \text{в інших випадках} \end{cases}$$

Розробити алгоритм розв'язання задачі та подати його в текстовій формі та у вигляді блок-схеми.

Аналіз задачі

На відміну від прикладу 1 у даній задачі функція y(x) задана в цілочислових точка прямої, а не на всій числовій прямій. Потрібно задати ціле число x та обчислити значення функції залежно від значення цього числа.

Текстова форма

К1. Ввести x.

К2. Якщо $x == -3; -2;\ -1$, то обчислити $y = tg\ x + 3$ і перейти на К5.

К3. Якщо $x == 2$, то обчислити $y = 2cos(2x)$ і перейти на К5.

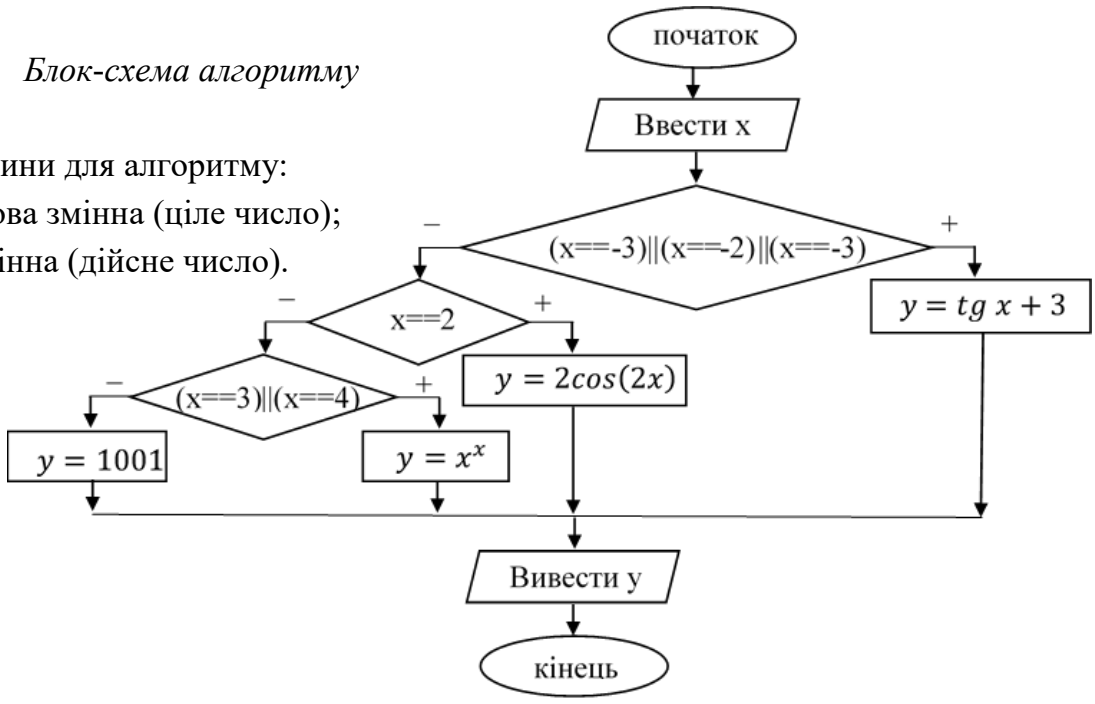
К4. Якщо $x == 3; 4$, то обчислити $y = x^x$. Інакше обчислити $y = 1001$.

К5. Вивести y.

Зауваження. У мовах програмування символ = може трактуватися як оператор присвоєння або як відношення рівності. **Ми позначаємо символом = оператор присвоєння, а відношення рівності позначаємо ==.**

Умова в алгоритмічних конструкціях розгалуження і циклах може бути складним виразом. Зазвичай вона утворюється за допомогою операторів відношень (== – дорівнює, < – менше, <= – менше або рівно, > – більше, >= – більше або рівно) та логічних операторів (! – не (заперечення), && – логічне «і», || – логічне «або»). Ми використали позначення операторів, прийнятих в мові C++.

Блок-схема алгоритму



Змінні величини для алгоритму:
x – цілочислова змінна (ціле число);
y – дійсна змінна (дійсне число).

Приклад 3

Дані числа $a_1, b_1, c_1, a_2, b_2, c_2$. Знайти координати точки перетину прямих, що задаються рівняннями $a_1x + b_1y = c_1$ і $a_2x + b_2y = c_2$ або повідомити, що прямі співпадають, паралельні, не існують.

Розробити алгоритм розв'язання задачі та подати його в текстовій формі та у вигляді блок-схеми.

Аналіз задачі

Спочатку вияснимо, чи існують прямі. Пряма $a_1x + b_1y = c_1$ існує, якщо хоч одне з чисел a_1, b_1 не рівне нулю. Значить, прямої не існує, якщо одночасно $a_1 = 0, b_1 = 0$. Аналогічно, прямої $a_2x + b_2y = c_2$ не існує, якщо одночасно $a_2 = 0, b_2 = 0$. Отже, обидві прямі існують, якщо виконується умова

$$((a_1 \neq 0)||(b_1 \neq 0))\ \&\&((a_2 \neq 0)||(b_2 \neq 0))\ .$$

При невиконанні цієї умови хоч одної з прямих не існує. У цьому випадку потрібно вивести повідомлення «прямої не існує».

Далі будемо вважати, що обидві прямі існують. На площині ці прямі можуть перетинатися, чи бути паралельними, чи співпадати. Якщо прямі перетинаються, то точка перетину єдина і потрібно знайти її координати. Якщо прямі паралельні, то точок перетину не існує. Нарешті, якщо прямі співпадають, то вони мають безліч спільних точок.

Розглянемо випадок перетину двох прямих. Прямі перетинаються в одній точці, якщо $\frac{a_1}{a_2} \neq \frac{b_1}{b_2}$, що рівносильно умові $a_1 \cdot b_2 - a_2 \cdot b_1 \neq 0$. Тоді координати точки перетину обчислюються за формулами

$$x_1 = \frac{c_1 \cdot b_2 - c_2 \cdot b_1}{a_1 \cdot b_2 - a_2 \cdot b_1}\ , \qquad x_2 = \frac{a_1 \cdot c_2 - a_2 \cdot c_1}{a_1 \cdot b_2 - a_2 \cdot b_1}.$$

Прямі паралельні, якщо $\frac{a_1}{a_2} = \frac{b_1}{b_2} \neq \frac{c_1}{c_2}$. Ця умова рівносильна одночасному виконанню двох вимог $a_1 \cdot b_2 - a_2 \cdot b_1 = 0$ і $a_1 \cdot c_2 - a_2 \cdot c_1 \neq 0$.

Прямі співпадають, якщо $\frac{a_1}{a_2} = \frac{b_1}{b_2} = \frac{c_1}{c_2}$, що рівносильно одночасному виконанню двох рівностей $a_1 \cdot b_2 - a_2 \cdot b_1 = 0$ і $a_1 \cdot c_2 - a_2 \cdot c_1 = 0$.

Текстова форма

К1. Ввести числа $a_1, b_1, c_1, a_2, b_2, c_2$.

К2. Якщо умова $((a_1 \neq 0)||(b_1 \neq 0))\ \&\&((a_2 \neq 0)||(b_2 \neq 0))$ не виконується, то вивести повідомлення «прямої не існує» і перейти на К5.

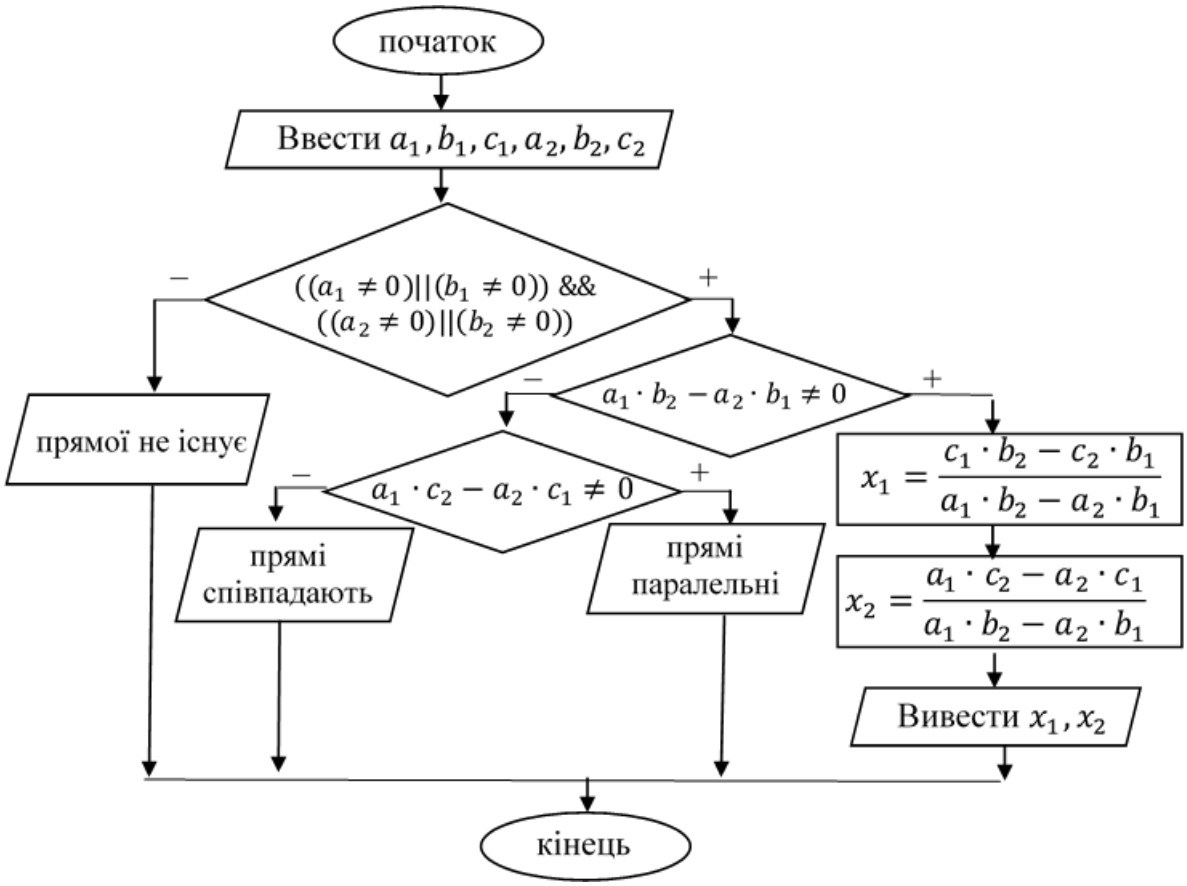
К3. Якщо $a_1 \cdot b_2 - a_2 \cdot b_1 \neq 0$, то обчислити координати точки перетину прямих за формулами $x_1 = \frac{c_1 \cdot b_2 - c_2 \cdot b_1}{a_1 \cdot b_2 - a_2 \cdot b_1}\ , \qquad x_2 = \frac{a_1 \cdot c_2 - a_2 \cdot c_1}{a_1 \cdot b_2 - a_2 \cdot b_1}$, вивести ці координати і перейти на К5.

К4. Якщо $a_1 \cdot c_2 - a_2 \cdot c_1 \neq 0$, то вивести повідомлення «прямі паралельні». Інакше вивести повідомлення «прямі співпадають».

К5. Кінець

Зауваження. У команді К4 не потрібно вимагати виконанні умови $a_1 \cdot b_2 - a_2 \cdot b_1 = 0$, бо вона уже виконується. Ця команда виконається лише тоді, коли не спрацюють команди К2 та К3. Команда К3 не виконається тоді, коли умова $a_1 \cdot b_2 - a_2 \cdot b_1 \neq 0$ хибна, тобто має місце рівність $a_1 \cdot b_2 - a_2 \cdot b_1 = 0$.

Блок-схема алгоритму



Змінні величини для алгоритму:

$a_1, b_1, c_1, a_2, b_2, c_2$ – дійсні змінні (дійсні числа);

x_1, y_1 – дійсні змінні (дійсні числа).

Приклад 4

Дано розміри a, b, c цеглини і розміри x, y прямокутного отвору. Вияснити, чи пройде цеглина в отвір. Просувати цеглину в отвір дозволяється тільки так, щоб кожне її ребро було паралельне або перпендикулярне кожній із сторін отвору.

Розробити алгоритм розв'язання задачі та подати його в текстовій формі та у вигляді блок-схеми.

Аналіз задачі

Можна розв'язувати задачу, перебираючи всі можливі варіанти розмірів цеглини та отвору. Однак, при такому підході можна припуститися помилки.

Припустимо, що розміри задовольняють умовам $a \leq b \leq c$, $x \leq y$. Тоді цеглина пройде в отвір, якщо $a < x$ і $b < y$. Таким чином, можна запропонувати наступний

алгоритм розв’язання задачі: переставити числа a, b, c так, щоб виконувалась вимога $a \le b \le c$; переставити числа x, y, щоб $x \le y$.

Текстова форма

К1. Ввести числа a, b, c, x, y.

Упорядкування a, b, c за зростанням, тобто виконання умови $a \le b \le c$.

К2. Якщо $a > b$, то поміняти їхні значення: $z = a, a = b, b = z$.

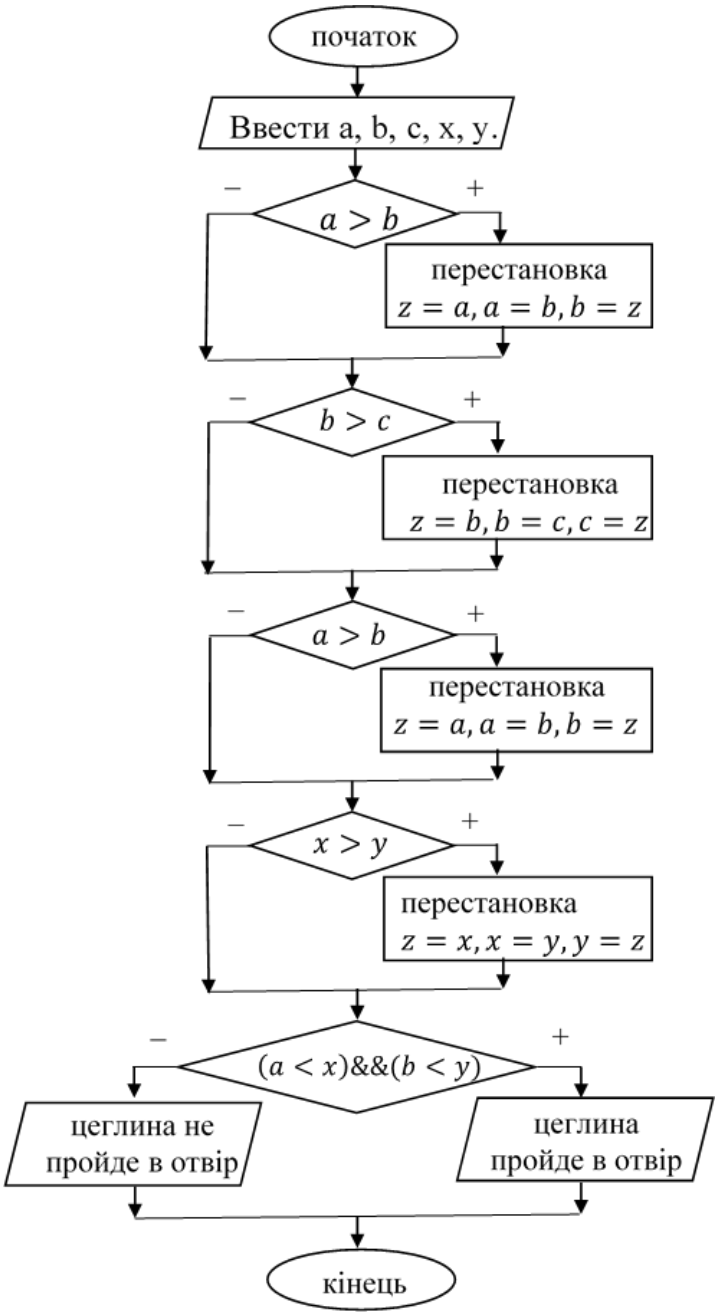
К3. Якщо $b > c$, то поміняти їхні значення: $z = b, b = c, c = z$.

К4. Якщо $a > b$, то поміняти їхні значення: $z = a, a = b, b = z$.

Упорядкування x, y за зростанням, тобто виконання умови $x \le y$.

К5. Якщо $x > y$, то поміняти їхні значення: $z = x, x = y, y = z$.

К6. Якщо $a < x$ і $b < y$, то вивести повідомлення «цеглина пройде в отвір». Інакше вивести повідомлення «цеглина не пройде в отвір».



Блок-схема алгоритму

Зауваження. Команди К2 і К4 ідентичні. Однак вони потрібні для упорядкування чисел a, b, c. Процес упорядкування здійснювався методом сортування, який називають методом бульбашок. Розглянемо його на прикладі. Нехай задані значення чисел a, b, c. Наведена нижче таблиця демонструє роботу команд К2 – К4, які упорядковують послідовність за зростанням.

a	b	c	
10	6	2	Початкові значення чисел
6	10	2	Виконання команди К2
6	2	10	Виконання команди К3
2	6	10	Виконання команди К4

Змінні величини для алгоритму:

a, b, c, x, y – дійсні змінні (дійсні числа);

z – дійсна допоміжна змінна (дійсне число).

Задачі

Для вказаних задач розробити алгоритм їхнього розв'язання. Подати алгоритм в текстовій формі та у вигляді блок-схеми.

1. Дано число x. Обчислити y(x):

$$y(x) = \begin{cases} 0, \text{якщо } x \le 0, \\ x^2 - x, \text{якщо } 0 < x \le 1, \\ x^2 - \sin \pi x^2 - 1, \text{ в інших випадках.} \end{cases}$$

2. Написати програму обчислення коренів квадратного рівняння $ax^2+bx+c=0$.
3. Дані числа x, y, z. Обчислити $V=\min(x^2+y^2+z^2, (x+y+z)^2) -3$.
4. Надрукувати фразу "мені **n** рік (роки, років)" так, щоб відмінок слова "рік" узгоджувався з числом **n**, яке вводиться з клавіатури.
5. Знайти найбільше із трьох різних даних дійсних чисел.
6. Визначити, чи дорівнює сума двох перших цифр заданого натурального чотиризначного числа сумі двох його останніх цифр.

☺ Індивідуальні завдання

Виконати аналіз вказаних задач та розробити алгоритм їхнього розв'язання. Подати алгоритм в текстовій формі та у вигляді блок-схеми.

A

1. Визначити, чи є серед трьох цілих чисел **a**, **в**, **c** хоча б одна пара рівних між собою.
2. Дано прямокутник зі сторонами **a**, **b** і квадрат із стороною **c**. З'ясувати, чи поміститься прямокутник у квадрат.
3. Визначити, чи є серед цифр заданого натурального тризначного числа однакові.
4. Визначити, чи дорівнює квадрат заданого двозначного числа кубу суми цифр цього числа.
5. Дано тризначне натуральне число. Скільки в ньому різних цифр?
6. Знайти найменше із трьох різних даних дійсних чисел.
7. Дано один прямокутник зі сторонами **a**, **b** і другий прямокутник зі сторонами **c**, **d**. З'ясувати, чи можна перший прямокутник розмістити всередині другого так, щоб сторони прямокутників були паралельні.
8. Дано два циліндри з радіусами основ R1, R2 і висотами H1, H2 відповідно. З'ясувати, чи можна розмістити один циліндр всередині другого так, щоб їхні твірні були паралельними.

9. Дано три різних дійсних числа. Знайти те з них, яке за величиною міститься між двома іншими.
10. Поміняти місцями три різних дійсних числа **x, y, z** так, щоб вони утворили спадну послідовність.

Б

Дано число x. Обчислити y(x):

1. $y = \begin{cases} \frac{1}{x^2}, \text{якщо } x \le -1; \\ -x, \text{якщо } -1 < x \le 0; \\ x^2, \text{якщо } 0 < x < 1; \\ 1, \text{в іншому випадку.} \end{cases}$

2. $y = \begin{cases} x + 2, \text{якщо } x \le -2; \\ 0, \text{якщо } -2 < x \le 0; \\ \frac{x}{2}, \text{якщо } 0 < x \le 2; \\ \sqrt{x-1}, \text{в іншому випадку.} \end{cases}$

3. $y = \begin{cases} (x+1)^2, \text{якщо } x \le -1; \\ x + 1, \text{якщо } -1 < x \le 0; \\ 2^x, \text{якщо } 0 < x \le 2; \\ 4, \text{в іншому випадку.} \end{cases}$

4. $y = \begin{cases} \cos 3x, \text{якщо } x \le 0; \\ -x + 1, \text{якщо } 0 < x \le 1; \\ (x-1)^2, \text{якщо } 1 < x \le 3; \\ 4, \text{в іншому випадку.} \end{cases}$

5. $y = \begin{cases} 3x + 15, \text{якщо } x \le -4; \\ 3, \text{якщо } -4 < x \le -2; \\ x^2 - 1, \text{якщо } -2 < x \le 1; \\ log_{1/2} x, \text{в іншому випадку.} \end{cases}$

6. $y = \begin{cases} \sin \frac{\pi x}{4}, \text{якщо } x \le 0; \\ x^2, \text{якщо } 0 < x \le 2; \\ 3x - 2, \text{якщо } 2 < x \le 3; \\ \sqrt{5x^2 + 4}, \text{в іншому випадку.} \end{cases}$

7. $y = \begin{cases} 4^{x+2}, \text{якщо } x \le -2; \\ -2x - 3, \text{якщо } -2 < x \le 0; \\ -3, \text{якщо } 0 < x \le 8; \\ log_{1/2} x, \text{в іншому випадку.} \end{cases}$

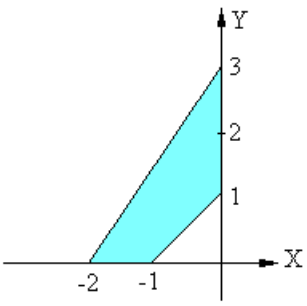
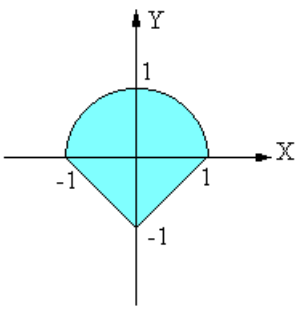
8. $y = \begin{cases} \sqrt[3]{x+1}, \text{якщо } x \le -2; \\ -|x| + 1, \text{якщо } -2 < x \le 0; \\ arcsin\, x + 1, \text{якщо } 0 < x \le 1; \\ \frac{\pi}{2} + 1, \text{в іншому випадку.} \end{cases}$

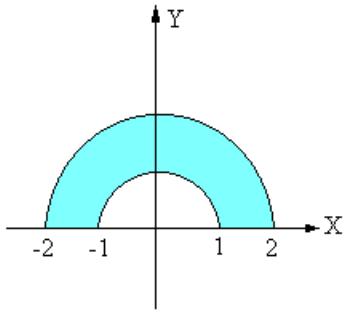
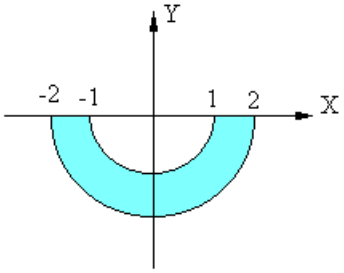
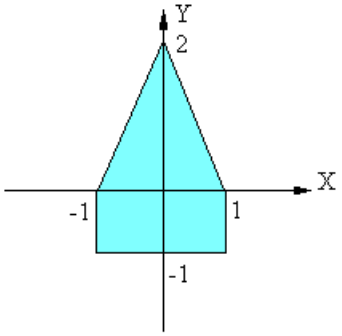
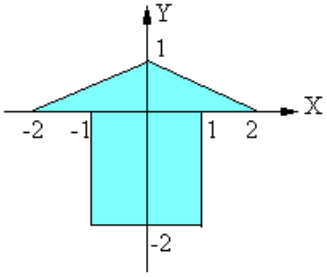
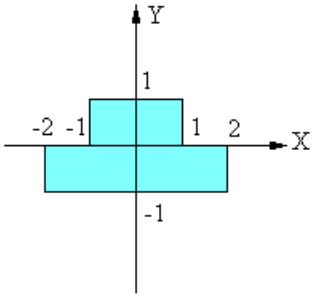
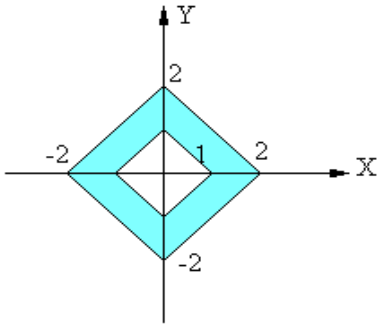
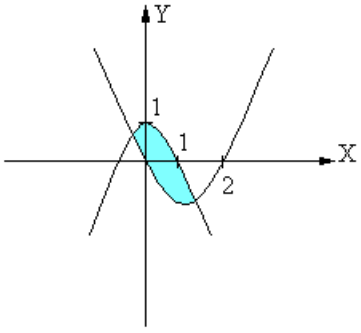
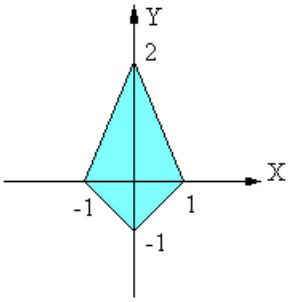
9. $y = \begin{cases} x^{2/3} + x^{1/3}, \text{якщо } x \le 0; \\ \sqrt{x}, \text{якщо } 0 < x \le 4; \\ 2, \text{якщо } 4 < x \le 9; \\ log_3 x, \text{в іншому випадку.} \end{cases}$

10. $y = \begin{cases} \pi, \text{якщо } x \le -1; \\ \arccos x, \text{якщо } -1 < x \le 1; \\ x^2 - 1, \text{якщо } 1 < x \le 3; \\ 4\sqrt{x+1}, \text{в іншому випадку.} \end{cases}$

В

Дані числа x, y. З'ясувати, чи належить точка (x, y) області, зображеній на малюнку.

1. 	2. 

3.  Y X -2 -1 1 2	4.  Y X -2 -1 1 2
5.  Y 2 X -1 1 -1	6.  Y 1 X -2 -1 1 2 -2
7.  Y 1 -2 -1 1 2 X -1	8.  Y 2 -2 1 2 X -2
9.  Y 1 1 X 2	10.  Y 2 X -1 1 -1

Тема 3: Цикли

Теоретичні відомості

Алгоритмічна конструкція повторення (цикли) розглянута в темі 1. Там наведені два види циклів: ПОКИ і ДО. У мовах програмування зазвичай використовується ще цикл з лічильником, що починається зі службового слова **for**. Цей цикл є різновидом циклу ПОКИ.

Приклад 1

Дано натуральне число **n** та дійсне число **x**. Обчислити:

$$\frac{x}{2!}+\frac{x^2}{3!}+\cdots+\frac{x^n}{(n+1)!}.$$

Розробити алгоритм розв'язання задачі та подати його в текстовій формі та у вигляді блок-схеми.

Аналіз задачі

Вираз $\frac{x}{2!}+\frac{x^2}{3!}+\cdots+\frac{x^n}{(n+1)!}$ є сумою n доданків. Якщо задати число x, то кожний доданок є числом. Таким чином, якщо задати x та n, то потрібно обчислити суму n чисел.

Розглянемо обчислення суми багатьох доданків у загальному вигляді, яке використовується у програмуванні. Нехай задана сума n чисел

$a_1+a_2+\cdots+a_n.$ (3.1)

Введемо позначення для так званих часткових сум:

$S_1=a_1$ – сума з одного доданку,

$S_2=a_1+a_2$ – сума з двох доданків,

$S_3=a_1+a_2+a_3$ – сума з трьох доданків,

$S_k=a_1+a_2+a_3+\cdots+a_k$ – сума з k доданків,

$S_n=a_1+a_2+a_3+\cdots+a_n$ – сума з n доданків.

Остання сума S_n – це вираз (3.1), який потрібно обчислити. На основі часткових сум, отримуємо дуже важливу рівність

$S_{k+1}=S_k+a_{k+1}.$ (3.2)

Ця рівність стверджує, що суму із k+1 доданків можна обчислити, якщо до суми із перших k доданків додати (k+1)-й доданок. Рівність (3.2) є прикладом лінійного рекурентного співвідношення першого порядку – обчислення суми через суму.

Якщо у мові програмування для зберігання суми створити змінну **suma**, а для доданку – змінну **a**, то рівність (3.2) запишеться у вигляді присвоєння

suma = suma + a. (3.3)

Це є основне присвоєння для обчислення суми.

Перейдемо тепер до аналізу виразу $\frac{x}{2!}+\frac{x^2}{3!}+\cdots+\frac{x^n}{(n+1)!}$. Доданки суми мають вигляд $a_k=\frac{x^k}{(k+1)!}$, k = 1,2,…,n. Вони мають степені та факторіали, які дуже швидко

зростають при збільшенні k. Доцільно обчислювати кожний наступний доданок через попередній. Оскільки $a_{k+1} = \frac{x^{k+1}}{(k+2)!}$, то $a_{k+1} = \frac{x^k}{(k+1)!} \cdot \frac{x}{k+2} = a_k \cdot \frac{x}{k+2}$. Ця рівність показує, що (k+1)-й доданок можна обчислити через k-й доданок, помноживши останній на $\frac{x}{k+2}$. Важливо те, що в такому способі обчислення доданків відсутні степені та факторіали. Мовою програмування обчислення доданків записується через присвоєння

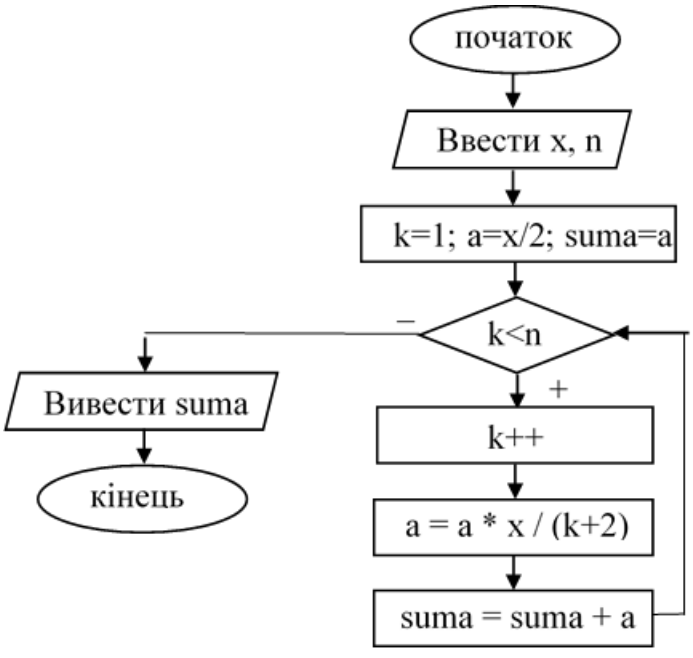
a = a * x / (k+2). (3.4)

Присвоювання (3.3) та (3.4) вимагають задання початкових значень змінних **suma** і **a**. Крім того, треба також задати початкове значення лічильника **k**, який задає номер доданку. Цей лічильник буде інформувати про те, який доданок додається до суми в присвоєнні (3.3). Доцільно покласти **k**=1. Тоді початкове значення змінної **a** треба покласти рівним першому доданку суми: **a**=x/2. Відповідно, таке ж значення отримає змінна **suma**=**a**. Далі в циклі потрібно повторювати присвоєння:

1) збільшуємо **k** на 1;
2) обчислюємо k-й доданок **a = a * x / (k+2)**;
3) додаємо цей доданок до суми **suma = suma + a**.

Ці дії будемо повторювати в циклі ПОКИ до тих пір, поки k<n. Цикл припинить свою роботу коли k стане рівним n.

У даному прикладі кількість доданків суми відома наперед і задається числом n. Зауважимо також, що можна було б задати інші початкові значення змінних **k**, **a**, **suma** і тоді порядок виконання дій 1) – 3) в циклі міг би бути іншим.



Блок-схема алгоритму

Текстова форма

К1. Ввести x та n.
К2. Присвоїти k=1, a=x/2, suma=a.
К3. Якщо k<n, то збільшити k на 1.Інакше перейти на К6.
К4. Обчислити a = a * x / (k+2), suma = suma + a.
К5. Перейти на К3.
К6. Вивести значення suma.

Змінні величини для алгоритму:
x, a, suma – дійсні змінні (дійсні числа);
n, k – цілочислові змінні (цілі числа).

Приклад 2

Функція *cos x* представлена рядом:

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \cdots + \frac{(-1)^n x^{2n}}{(2n)!} + \cdots \qquad \left(|x| < \frac{\pi}{4}\right).$$

Задавши *x*, обчислити значення функції з точністю до ε>0. Необхідна точність вважається досягнутою, якщо черговий доданок за модулем менший ε. Усі наступні доданки можна не враховувати.

Розробити алгоритм розв'язання задачі та подати його в текстовій формі та у вигляді блок-схеми.

Аналіз задачі

Вираз $1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \cdots + \frac{(-1)^n x^{2n}}{(2n)!} + \cdots$ є степеневим рядом. Такі ряди часто використовуються для обчислення наближеного значення багатьох відомих математичних функцій. Наприклад, якщо б ми захотіли обчислити значення функції cos x при x=0.1, то не отримали б точного значення цієї функції, а лише наближене значення. Це наближене значення отримується через вказаний степеневий ряд, у якому замість x підставлене число 0.1.

Таким чином, якщо в степеневому ряду x замінити на якесь число, то отримаємо нескінченну суму числових доданків, яку формально можна записати так

$$a_0 + a_1 + a_2 + \cdots a_n + \cdots \qquad (3.5)$$

Вираз (3.5) називається числовим рядом. При прямуванні n до нескінченності числовий ряд може бути наближатися до якогось числа або ні.

У нашому прикладі потрібно виконати обчислення з точністю до ε>0. Якщо задати ε=0.01, то кажуть, що обчислення виконуються до сотих, а при ε=0.001 – до тисячних і т. д. Точність визначає похибку між істинним результатом і наближеним результатом. Наприклад, якщо істинний результат дорівнює 34.455672, то з точністю до ε=0.001 наближений результат буде рівний 34.455, тобто з точністю до трьох цифр дробової частини числа.

У прикладі 1 кількість доданків суми задавалася числом n. У даному прикладі кількість доданків невідома і залежить від точності ε.

При обчисленні суми ми будемо використовувати ті ж самі міркування, що й в прикладі 1. Суму будемо обчислювати, використовуючи присвоєння (3.3). Тепер розглянемо, як обчислювати доданки. Формула k-го доданку має вигляд $a_k = \frac{(-1)^k x^{2k}}{(2k)!}$. Він містить степінь і факторіал. Доцільно його обчислювати рекурентно. (k+1)-й доданок має вигляд $a_{k+1} = \frac{(-1)^{k+1} x^{2(k+1)}}{(2(k+1))!}$. Поділимо a_{k+1} на a_k. Отримуємо

$$\frac{a_{k+1}}{a_k} = \frac{(-1)^{k+1} x^{2(k+1)} (2k)!}{(-1)^k x^{2k} \big(2(k+1)\big)!} = -\frac{x^2}{(2k+1)(2k+2)}.$$

Отже $a_{k+1} = -\frac{x^2}{(2k+1)(2k+2)} a_k$. Це і є шукане рекурентне співвідношення, яке дозволяє знаходити наступний доданок через попередній. Оскільки $a_0 = 1$, то рекурентне співвідношення дозволяє обчислювати доданки для k = 0,1,2,… Воно реалізується через присвоєння

a = –a · x · x / ((2k+1)(2k+2)).

Тепер алгоритм розв'язання задачі може бути таким. Початкові значення суми і доданку обираємо такими: **a=1**, **suma**=**a**. Також задаємо початкове значення лічильника **k**, яке покладаємо **k =-1**. Таке початкове значення лічильника обране тому, що рекурентне співвідношення для доданків виконується при k = 0, 1, 2, … Далі в циклі потрібно повторювати присвоєння:

1) збільшуємо **k** на 1;
2) обчислюємо k-й доданок a = -a · x · x / ((2k+1)(2k+2));
3) додаємо цей доданок до суми **suma = suma + a**.

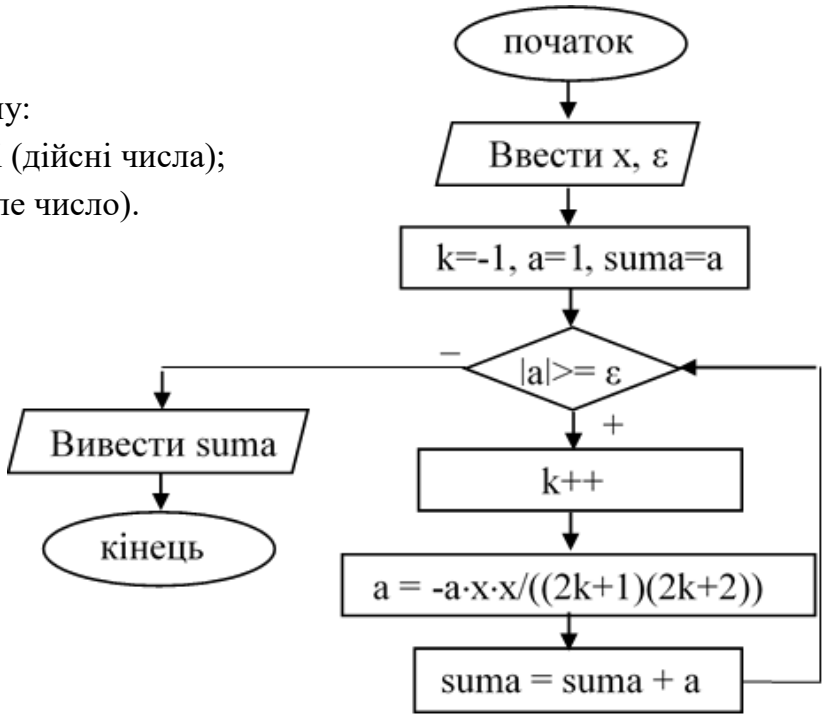
Ці дії будемо повторювати в циклі ПОКИ до тих пір, поки **|a|>= ε**. Цикл припинить свою роботу коли **|a|** стане меншим за **ε**. Цей доданок ще буде добавлений до суми, а всі наступні доданки вже не будуть додаватися до суми.

Текстова форма

К1. Ввести x та ε.
К2. Присвоїти k=-1, a=1, suma=a.
К3. Якщо |a|>= ε, то збільшити k на 1.Інакше перейти на К6.
К4. Обчислити a = -a * x*x / ((2k+1)(2k+2)), suma = suma + a.
К5. Перейти на К3.
К6. Вивести значення suma.

Змінні величини для алгоритму:

x, a, suma, ε – дійсні змінні (дійсні числа);
k – цілочислова змінна (ціле число).



Блок-схема алгоритму

Приклад 3

Дано натуральне число **n** та дійне число **x**. Обчислити:

$$\left(\frac{1}{2} - \cos|x|\right) \cdot \left(\frac{2}{3} - cos^2|x|\right) \cdot \ldots \cdot \left(\frac{n}{n+1} - cos^n|x|\right).$$

Аналіз задачі

У попередніх прикладах ми обчислювали суми доданків. У даному прикладі потрібно обчислити добуток скінченого числа множників. Будемо проводити це обчислення так само як для суми, використовуючи рекурентні співвідношення.

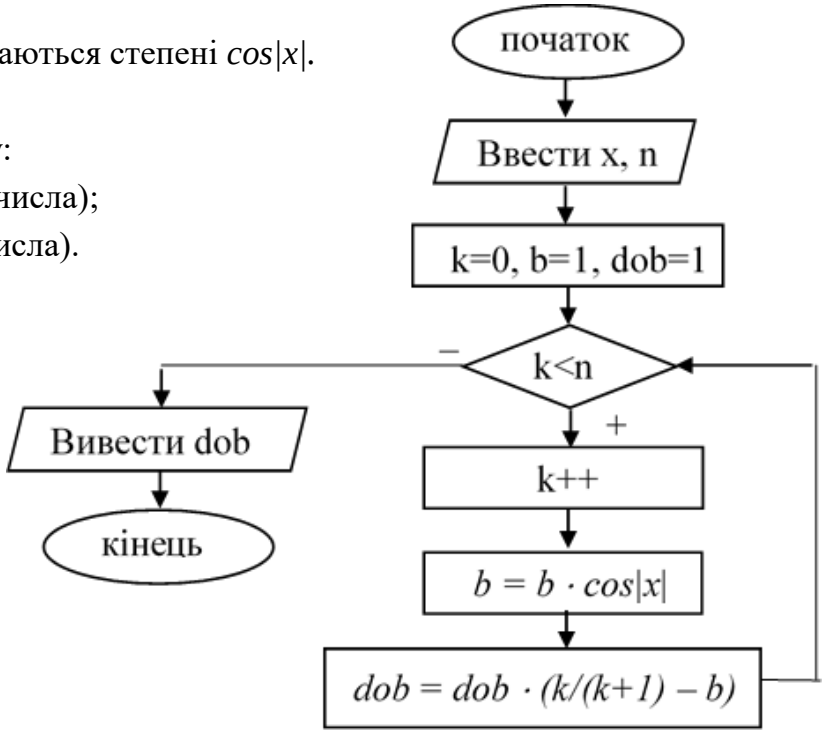
Якщо позначити через D_k добуток k множників, то отримуємо рекурентне співвідношення $D_{k+1} = D_k \cdot a_{k+1}$ для обчислення добутку (k+1) множників, де a_{k+1} – (k+1)-й множник. Позначимо змінну для добутку як **dob**. Тоді рекурентне співвідношення можна реалізувати у вигляді присвоєння **dob = dob*a**. Зазвичай початкове значення добутку покладають рівним 1. При створенні алгоритму потрібно використати лічильник k, який буде визначати номери множників. Початковий множник має номер 1.

Текстова форма

К1. Ввести x та n.
К2. Присвоїти k=0, b=1, dob=1.
К3. Якщо k<n, то збільшити k на 1.Інакше перейти на К6.
К4. Обчислити b = b * cos|x|, dob = dob * (k/(k+1) – b).
К5. Перейти на К3.
К6. Вивести значення dob.

Зауваження. У змінній *b* зберігаються степені *cos|x|*.

Змінні величини для алгоритму:
x, b, dob – дійсні змінні (дійсні числа);
k, n – цілочислові змінні (цілі числа).



Блок-схема алгоритму

Приклад 4

З точністю до ε>0 знайти корінь рівняння *arctg x – ln x = 0* методом поділу відрізка навпіл.

Розробити алгоритм розв'язання задачі та подати його в текстовій формі та у вигляді блок-схеми.

Аналіз задачі

Задане рівняння напишемо у вигляді функціональної рівності $f(x) = 0$, де $f(x) = \text{arctg}\, x - \ln x$ – функція. Знайти корінь рівняння рівносильно знаходженню такого значення x, при якому функція приймає значення 0.

Якщо графічно зобразити функції arctg x та ln x, то вони перетинаються в точці, яка належить відрізку [1; 5]. Дійсно $f(1) = arctg\ 1 - ln\ 1 = 0.785398$, а $f(5) = arctg\ 5 - ln\ 5 = -0.23604$. Тобто на кінцях відрізка [1; 5] функція f(x) приймає значення протилежні за знаками ($f(1) * f(5) < 0$), а значить у якійсь точці цього відрізка функція приймає значення 0.

Розглянемо суть методу поділу відрізка навпіл (методу бісекції). Нехай задане рівняння $f(x) = 0$ і відомо, що на відрізку $[a_0, b_0]$ рівняння має тільки один простий корінь і на кінцях відрізка функція приймає значення різних знаків, тобто $f(a_0) * f(b_0) < 0$. Місцезнаходження кореня на відрізку невідоме і його відшукують з певною точністю. Для цього уточнюють положення кореня, будуючи послідовність вкладених інтервалів, кожний із яких містить корінь. Алгоритм знаходження кореня рівняння методом поділу відрізка навпіл складається з послідовності аналогічних кроків, один із яких розглянемо.

Припустимо, що на k-му кроці ми отримали відрізок $[a_k, b_k]$, який містить шуканий корінь і на кінцях відрізка функція приймає значення різних знаків, тобто $f(a_k) * f(b_k) < 0$. Знаходимо середину відрізка $c_k=(a_k+b_k)/2$. Якщо $f(a_k) * f(c_k) < 0$, то покладаємо $a_{k+1}=a_k$, $b_{k+1}=c_k$, а якщо $f(c_k) * f(b_k) < 0$, то покладаємо $a_{k+1}=c_k$, $b_{k+1}=b_k$. В результаті знайдений новий (k+1)-й відрізок $[a_{k+1}, b_{k+1}]$, що містить корінь. Для цього відрізка знову застосовуємо описані дії, тобто ділимо його навпіл і визначаємо новий відрізок.

Алгоритм припиняє роботу, якщо виявиться, що $f(a_k) * f(c_k) = 0$. Тоді $f(c_k) = 0$ і коренем є c_k, який знайдений точно. В іншому випадку, алгоритм припиняє роботу коли $b_{k+1} - a_{k+1} \le \varepsilon$. Наближене значення кореня знаходиться за формулою $(b_{k+1} + a_{k+1})/2$.

Текстова форма

К1. Ввести точність ε>0.

К2. Присвоїти a=1, b=5 (a, b – кінці відрізка).

К3. Якщо b-a>=ε перейти на К4.Інакше перейти на К6.

К4. Обчислити c = (b-a)/2 – середина відрізка.

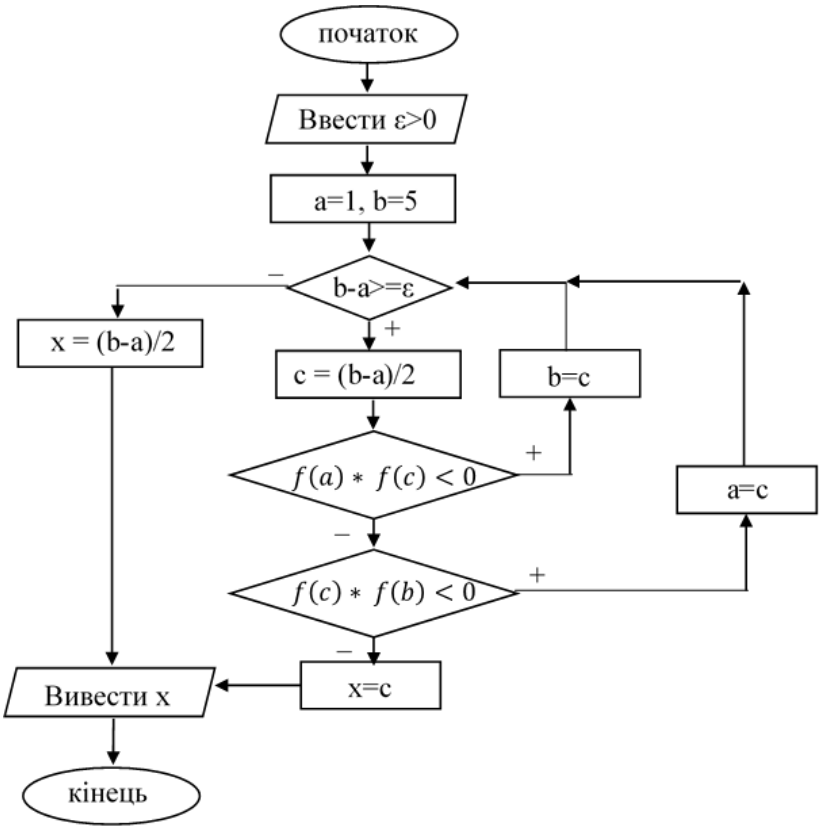
К5. Якщо $f(a) * f(c) < 0$, то покласти b=c і перейти на К3. Інакше, якщо $f(c) * f(b) < 0$, то покласти a=c і перейти на К3. Інакше x=c – корінь і перейти на К7.

К6. Обчислити x = (b-a)/2 – наближене значення кореня.
К7 Вивести значення x.

Блок-схема алгоритму

Змінні величини для алгоритму:

ε, a, b, c, x – дійсні числа.



Приклад 5

Вивести всі натуральні числа менші за n, сума цифр яких рівна m.

Розробити алгоритм розв'язання задачі та подати його в текстовій формі та у вигляді блок-схеми.

Аналіз задачі

Алгоритм розв'язання задачі простий. Потрібно в цикл послідовно переглядати натуральні числа і для кожного числа знаходити суму цифр.

Розглянемо знаходження цифр натурального числа. У мовах програмування є операції над цілими числами, які часто позначають div – ціла частина від ділення та mod – остача від ділення. У мові C++ ці операції позначаються / та % відповідно. Будемо використовувати нотацію мови C++.

Візьмемо число k=730350 і зайдемо його цифри, які зберігатимемо в змінній c.

1) c = k % 10 – остання цифра c=0, остача від ділення на 10,
2) k = k / 10 – ціла частина від ділення на 10, нове k = 73035,
3) c = k % 10 – цифра десятків c=5,
4) k = k / 10 – ціла частина від ділення на 10, нове k = 7303,
5) c = k % 10 – цифра сотень c=3,
6) k = k / 10 – ціла частина від ділення на 10, нове k = 730,
7) c = k % 10 – цифра тисяч c=0,

8) k = k / 10 – ціла частина від ділення на 10, нове k = 73,

9) c = k % 10 – цифра десятків тисяч c=3,

10) k = k / 10 – ціла частина від ділення на 10, нове k = 7,

11) c = k % 10 – цифра сотень тисяч c=7,

12) k = k / 10 – ціла частина від ділення на 10, нове k = 0.

Як бачимо в алгоритмі знаходження цифр числа повторюються дії: знаходження остачі від ділення на 10, знаходження цілої частини від ділення на 10. Ці дії легко реалізувати в циклі, який завершується при k=0.

Текстова форма

К1. Ввести n, m.

К2. Присвоїти p =1 (p – змінна для натурального числа).

К3. Якщо p<n перейти на К4.Інакше перейти на К9.

Обчислення суми цифр числа p

К4. Присвоїти suma = 0, k=p (змінна suma накопичує суму цифр).

К5. Якщо k≠0, то перейти на К6. Інакше перейти на К8.

К6. Обчисли suma = suma + k % 10 (до суми додається цифра).

К7. Присвоїти k = k / 10. Перейти на К5.

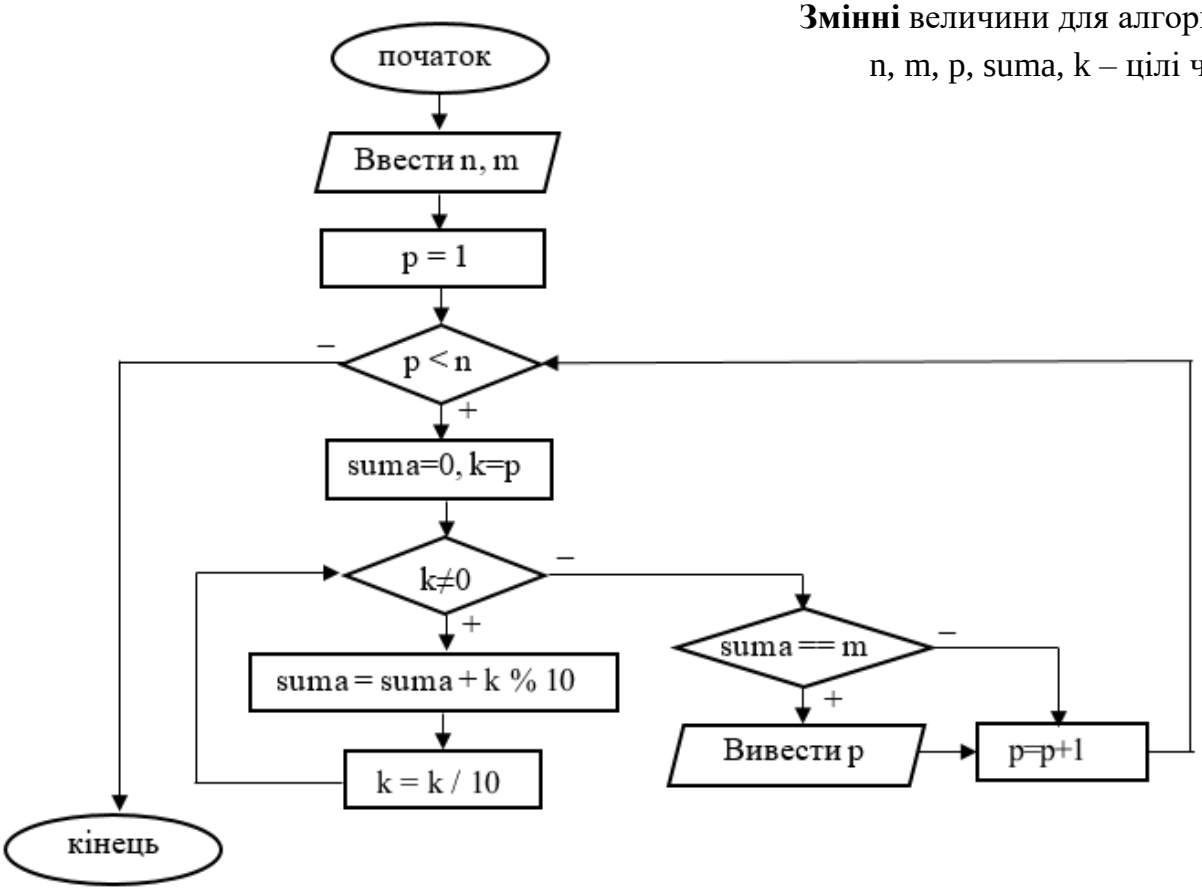
Порівняння суми цифр з числом m

К8. Якщо suma == m, то вивести p. Збільшити p на 1: p = p + 1. Перейти на К3.

К9. Кінець.

Блок-схема алгоритму

Змінні величини для алгоритму:
n, m, p, suma, k – цілі числа.



Приклад 6

Дана послідовність із n дійсних чисел. Знайти кількість членів у найдовшій зростаючій підпослідовності даної послідовності.

Розробити алгоритм розв'язання задачі та подати його в текстовій формі та у вигляді блок-схеми.

Аналіз задачі

Розглянемо алгоритм розв'язання задачі на прикладі. Нехай задана послідовність із 15 чисел: 6, 9, 2, -4, 0, 0, 3, 6, 5, 7, 10, -2, -8, 0, 1.

Для роботи з послідовністю використаємо дві змінні a та b для зберігання пар сусідніх чисел, лічильник k членів зростаючої підпослідовності та max – кількість членів у найдовшій зростаючій підпослідовності.

1. На початку покладаємо a = 6, k = 1 (зростаюча підпослідовність складається з одного числа і її довжина 1), max = 1.
2. Читаємо наступний член послідовності b = 9. Порівнюємо a з b. Оскільки a<b, то підпослідовність зростає. Збільшуємо k на 1: k = 2.
3. Читаємо наступний член послідовності, виконуючи присвоювання: a = 9, b = 2. Порівнюємо a з b. Оскільки a>b, то підпослідовність припинила зростання. Оскільки k>max, то покладаємо max = 2.
4. Читаємо наступний член послідовності, виконуючи присвоювання: k = 1, a = 2, b = -4. Порівнюємо a з b. Оскільки a>b, то підпослідовність припинила зростання. Оскільки k<max, то max не змінюється.
5. Читаємо наступний член послідовності, виконуючи присвоювання: k = 1, a = -4, b = 0. Порівнюємо a з b. Оскільки a<b, то підпослідовність зростає. Збільшуємо k на 1: k = 2.
6. Читаємо наступний член послідовності, виконуючи присвоювання: a = 0, b = 0. Порівнюємо a з b. Оскільки a=b, то підпослідовність припинила зростання. Оскільки k=max, то max не змінюється.
7. Читаємо наступний член послідовності, виконуючи присвоювання: k = 1, a = 0, b = 3. Порівнюємо a з b. Оскільки a<b, то підпослідовність зростає. Збільшуємо k на 1: k = 2.
8. Читаємо наступний член послідовності, виконуючи присвоювання: a = 3, b = 6. Порівнюємо a з b. Оскільки a<b, то підпослідовність зростає. Збільшуємо k на 1: k = 3.
9. Читаємо наступний член послідовності, виконуючи присвоювання: a = 6, b = 5. Порівнюємо a з b. Оскільки a>b, то підпослідовність припинила зростання. Оскільки k>max, то покладаємо max = 3.
10. Читаємо наступний член послідовності, виконуючи присвоювання: k = 1, a = 5, b = 7. Порівнюємо a з b. Оскільки a<b, то підпослідовність зростає. Збільшуємо k на 1: k = 2.

11. Читаємо наступний член послідовності, виконуючи присвоювання: a = 7, b = 10. Порівнюємо a з b. Оскільки a<b, то підпослідовність зростає. Збільшуємо k на 1: k = 3.
12. Читаємо наступний член послідовності, виконуючи присвоювання: a = 10, b = -2. Порівнюємо a з b. Оскільки a>b, то підпослідовність припинила зростання. Оскільки k=max, то max не змінюється.
13. Читаємо наступний член послідовності, виконуючи присвоювання: k = 1, a = -2, b = -8. Порівнюємо a з b. Оскільки a>b, то підпослідовність припинила зростання. Оскільки k<max, то max не змінюється.
14. Читаємо наступний член послідовності, виконуючи присвоювання: k = 1, a = -8, b = 0. Порівнюємо a з b. Оскільки a<b, то підпослідовність зростає. Збільшуємо k на 1: k = 2.
15. Читаємо наступний член послідовності, виконуючи присвоювання: a = 0, b = 1. Порівнюємо a з b. Оскільки a<b, то підпослідовність зростає. Збільшуємо k на 1: k = 3.
16. Перегляд послідовності завершений. Оскільки k=max, то max не змінюється.

Таким чином довжина найдовшої зростаючої підпослідовності рівна 3. Розглянутий приклад показує, що не потрібно зберігати в пам'яті комп'ютера всю послідовність. Достатньо знати її два сусідні елементи – a та b. На основі порівняння a з b вирішується питання про зростання підпослідовності. Як тільки припиняється зростання при a>b або завершений перегляд послідовності, ми порівнюємо k з max і починаємо розглядати нову підпослідовність або завершуємо алгоритм.

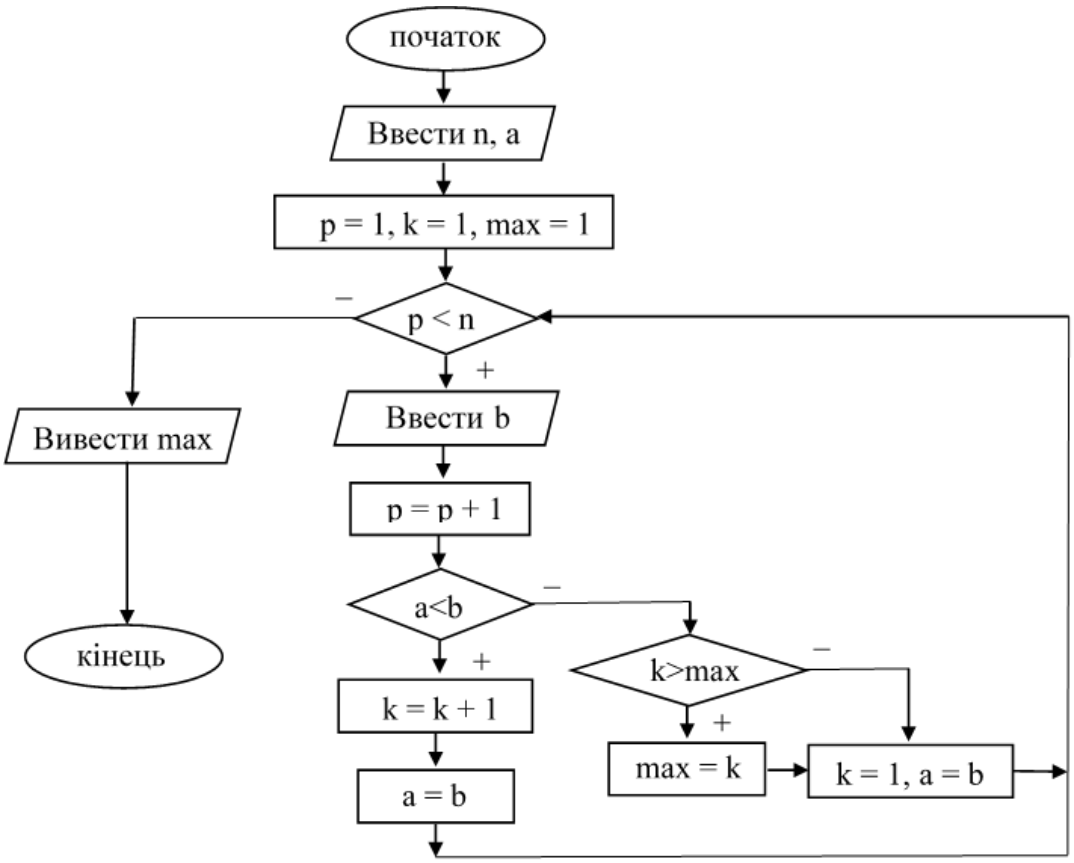
Текстова форма

К1. Ввести n.
К2. Ввести a. Присвоїти p =1 (p – лічильник членів послідовності).
К3. Присвоїти k = 1, max = 1.
К4. Якщо p<n перейти на К5.Інакше перейти на К9.
К5. Ввести b, присвоїти p = p + 1.
К6. Якщо a<b, то збільшуємо k на 1:k = k + 1. Присвоїти a = b. Перейти на К4. Інакше перейти на К7.
К7. Якщо k>max, то присвоїти max = k.
К8. Присвоїти k = 1, a = b. Перейти на К4.
К9. Ввести max.

Блок-схема алгоритму

Змінні величини для алгоритму:

n, p, k, max – цілочислові змінні (цілі числа);
a, b – дійсні змінні (дійсні числа)



Задачі

Для вказаних задач розробити алгоритм їхнього розв'язання. Подати алгоритм в текстовій формі та у вигляді блок-схеми.

1. Для даних **x** і **n** (**n** – натуральне) обчислити:
   a) $y = sinx \cdot sinx^2 \cdot sinx^3 \cdot \ldots \cdot sinx^n$;
   b) $y = sinx + sinx^2 + sinx^3 + \ldots + sinx^n$.
2. Стандартні функції представлені рядами:
   a) $y = e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^n}{n!} + \cdots \quad (0 \le x < 1)$,
   b) $y = \ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \cdots + \frac{(-1)^{n-1}x^n}{n} + \cdots \quad (|x| < 1)$,
   c) $y = \sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \cdots + \frac{(-1)^{n-1}x^{2n-1}}{(2n-1)!} + \cdots \quad \left(|x| < \frac{\pi}{4}\right)$,
   d) $y = arctg\, x = x - \frac{x^3}{3} + \frac{x^5}{5} - \cdots + \frac{(-1)^n x^{2n+1}}{2n+1} + \cdots \quad (|x| < 1)$.

   Задавши x, обчислити значення функції з точністю до ε>0. Необхідна точність вважається досягнутою, якщо черговий доданок за модулем менший ε. Усі наступні доданки можна не враховувати.
3. З точністю до ε>0 знайти корінь рівняння методом поділу відрізка навпіл:
   a) $x^3+x^2+x+1 = 0$ на відрізку [-2; 1];
   b) $x^2 \cos 2x + 1 = 0$ на відрізку [0; π/2];
   c) $2 \operatorname{arctg} x - e^{2x} = 0$.
4. Знайти канонічний розклад даного натурального числа.

5. Дано натуральне число k. Знайти k-ту цифру послідовності:
   a) 149162536…, у якій виписані підряд квадрати всіх натуральних чисел;
   b) 1123581321…, у якій виписані підряд всі числа Фібоначі.

☺ Індивідуальні завдання

Виконати аналіз вказаних задач та розробити алгоритм їхнього розв'язання. Подати алгоритм в текстовій формі та у вигляді блок-схеми.

А

Дано натуральне число **n**. Обчислити:

1. $\frac{1}{2^2}+\frac{1}{3^2}+\cdots+\frac{1}{(n+1)^2}$;
2. $\frac{1}{1^3}+\frac{1}{2^3}+\cdots+\frac{1}{n^3}$;
3. $\frac{1}{2!}-\frac{2}{3!}+\frac{3}{4!}-\cdots+\frac{(-1)^{n+1}\cdot n}{(n+1)!}$;
4. $\frac{1}{2^2}+\frac{1}{4^2}+\frac{1}{8^2}+\cdots+\frac{1}{2^{2n}}$;
5. $\frac{1^2}{1^2+5}\cdot\frac{2^2}{2^2+5}\cdot\ldots\cdot\frac{n^2}{n^2+5}$;
6. $\frac{1}{1^4}+\frac{1}{2^4}+\cdots+\frac{1}{n^4}$;
7. $\frac{1}{3^2}+\frac{1}{5^2}+\cdots+\frac{1}{(2n+1)^2}$;
8. $-\frac{1}{2}+\frac{1}{4}-\cdots+\frac{(-1)^n}{2n}$;
9. $\frac{1}{1\cdot2}-\frac{1}{2\cdot3}+\cdots+\frac{(-1)^{n+1}}{n\cdot(n+1)}$;
10. $-\frac{2}{1!}+\frac{3}{2!}-\cdots+\frac{(-1)^n(n+1)}{n!}$;

Б

Обчислити наближене значення нескінченної суми з точністю до заданого ε>0. Наближене значення вважати отриманим, якщо вже обчислена сума декількох перших доданків, а черговий доданок виявився за модулем меншим ε. Цей доданок теж треба потім додати до суми.

1. $1-\frac{1}{2}+\frac{1}{3}-\frac{1}{4}+\cdots+\frac{(-1)^{n+1}}{n}+\cdots$;
2. $1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\cdots+\frac{(-1)^n}{2n+1}+\cdots$;
3. $\frac{1}{1\cdot2!}+\frac{1}{2\cdot3!}+\frac{1}{3\cdot4!}+\cdots+\frac{1}{n\cdot(n+1)!}+\cdots$;
4. $1+\frac{2}{2}+\frac{3}{2^2}+\frac{4}{2^3}+\cdots+\frac{n}{2^{n-1}}+\cdots$;
5. $1+\frac{2}{1!}+\frac{2^2}{2!}+\frac{2^3}{3!}+\cdots+\frac{2^n}{n!}+\cdots$;
6. $1-\frac{1}{1\cdot2}+\frac{1}{2\cdot3}-\frac{1}{3\cdot4}+\cdots+\frac{(-1)^n}{n\cdot(n+1)}+\cdots$;

Функції подано у вигляді рядів. Задавши x, обчислити значення функції з точністю до заданого ε>0. Необхідна точність вважається досягнутою, якщо черговий доданок за модулем менший ε. Усі наступні доданки можна не враховувати:

7. $y=sh\,x=x+\frac{x^3}{3!}+\frac{x^5}{5!}+\cdots+\frac{x^{2n-1}}{(2n-1)!}+\cdots,\quad\left(|x|<\frac{\pi}{4}\right)$;
8. $y=ch\,x=1+\frac{x^2}{2!}+\frac{x^4}{4!}+\cdots+\frac{x^{2n}}{(2n)!}+\cdots,\quad\left(|x|<\frac{\pi}{4}\right)$;
9. $y=\ln\frac{1+x}{1-x}=2x\left(1+\frac{x^2}{3}+\frac{x^4}{5}+\cdots+\frac{x^{2n}}{2n+1}+\cdots\right),\ (|x|<1)$;
10. $y=\frac{1}{(1-x)^2}=1+2x+3x^2+\cdots+nx^{n-1}+\cdots,\quad(|x|<1)$;

В

1. Обчислити y=cos(1+cos(2+…+cos(39+cos40) …)).
2. Дано непарне натуральне n. Обчислити n!!=1*3*5*…*n.

3. Дано парне натуральне n. Обчислити n!!=2*4*…n.
4. Дано натуральне число. Знайти добуток його цифр.
5. Дано натуральне число. Знайти число, яке отримується записуванням цифр заданого числа у зворотному порядку.
6. Дано деяке натуральне число. Знайти в ньому цифру, що стоїть на к-й позиції.
7. Дано натуральне число. Знайти суму його цифр.
8. Дана послідовність із n цілих чисел, серед яких є хоч одне від'ємне. Знайти величину найбільшого від'ємного числа цієї послідовності.
9. Дано натуральне число. Визначити, скільки разів у його записі зустрічається цифра 0.
10. Дано n дійсних чисел. Знайти порядковий номер того з них, яке найближче до цілого.

Тема 4: Обчислювальна складність алгоритмів

Теоретичні відомості

У програмуванні обчислювальну складність алгоритмів зазвичай оцінюють за **кількістю дій**, які виконує алгоритм, та за **обсягом задіяної пам'яті**. У першому випадку кажуть про часову складність (час виконання алгоритму), а в другому – про ємнісну складність (кількість пам'яті, що необхідна для роботи алгоритму). Одним зі спрощених видів аналізу складності алгоритмів, що використовують при комп'ютерній їх реалізації, є асимптотичний аналіз алгоритмів. Асимптотичні оцінки показують як швидко зростають час роботи алгоритму чи обсяги пам'яті при збільшенні розміру вхідних даних.

Кількість вхідних даних алгоритму прийнято позначати натуральним числом n. Тоді час виконання алгоритму є додатною функцією від n.

При обчисленні складності алгоритму найчастіше використовується оцінка O-велике. Вона визначає функцію, яка показує, як буде змінюватися обчислювальна складність алгоритму зі зміною кількості вхідних даних у **найгіршому** для алгоритму випадку. Оцінка O-велике застосовуються, коли необхідно вказати верхню межу функції з точністю до сталого множника.

Означення 1. Функція складності алгоритму f(n) має оцінку O(«O-велике») й записується як f(n) = O(g(n)), якщо існує невід'ємна функція g(n) та додатні n_0, c такі, що

$$0 \le f(n) \le cg(n), \qquad \text{при всіх } n > n_0.$$

Для позначення оцінки найкращого випадку використовується Ω-нотація (омега велике), а для оцінки середнього випадку – Θ-нотація (тета велике).

Означення 2. Функція складності алгоритму f(n) має оцінку Ω(«омега-велике») й записується як f(n) = Ω(g(n)), якщо існує невід'ємна функція g(n) та додатні n_0, c такі, що

$$0 \le cg(n) \le f(n) \qquad \text{при всіх } n > n_0.$$

Означення 3. Функція складності алгоритму f(n) має оцінку Θ(тета) й записується як f(n) = Θ(g(n)), якщо існує невід'ємна функція g(n) та додатні n_0, c_1, c_2 такі, що

$$c_1 g(n) \le f(n) \le c_2 g(n), \qquad \text{при всіх } n > n_0.$$

У такому разі говорять, 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 сталого множника. Важливо розуміти, що Θ (g(n)) є не однією функцією, а множиною функцій для опису зростання f(n) з точністю до сталого множника. Іншими словами, функція f(n) належить множині Θ (g(n)), якщо існують додатні c_1 та c_2, що дозволяють обмежити цю функцію у рамки між функціями $c_1 g(n)$ та $c_2 g(n)$ для достатньо великих значень n.

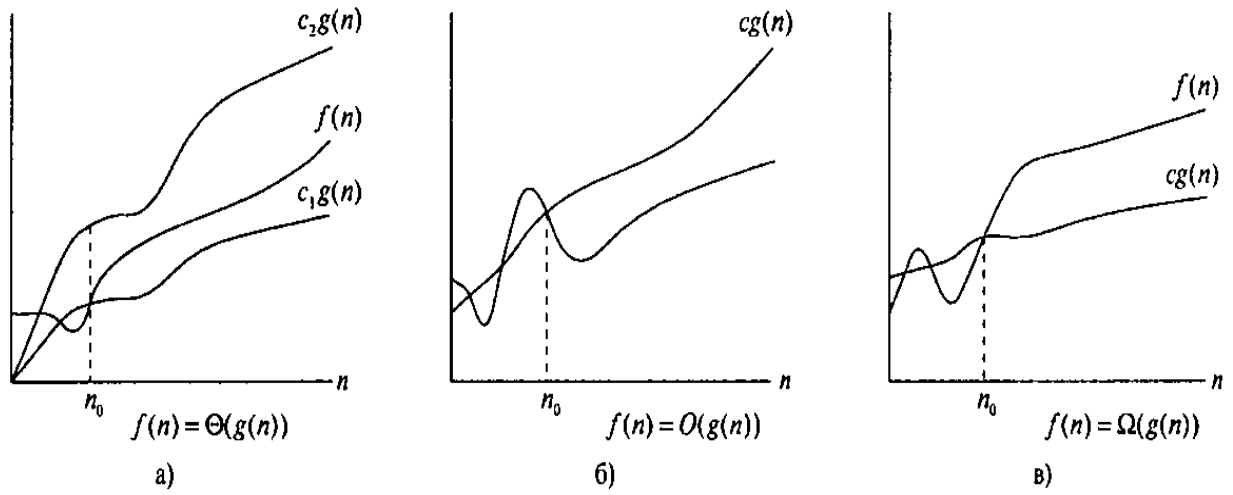


Рисунок 4.1 – Графічні приклади асимптотичних позначень

Теорема. Для будь-яких двох функцій f(n) і g(n) співвідношення f(n) = Θ(g(n)) виконується тоді й тільки тоді, коли f(n)=O(g(n)) і f(n)=Ω(g(n)).

На практиці ця теорема застосовується для визначення асимптотично точної оцінки алгоритму за допомогою асимптотично верхньої та нижньої меж.

У більшості випадків для алгоритму вдається знайти оцінку О-велике (оцінку роботи алгоритму в найгіршому випадку). Значно складніше знайти оцінку роботи в середньому.

При обчисленні оцінок О-велике використовують наступні властивості при умові, що всі функції є додатнозначними функціями натурального аргументу:

1. Нехай $f_1(n)=O(g_1(n))$ і $f_2(n)=O(g_2(n))$. Тоді $f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n))$ – правило суми.
2. Нехай $f(n) = f_1(n) + f_2(n)$, причому $\lim\limits_{n\to\infty} \frac{f_2(n)}{f_1(n)} = L$, де L деяка стала. Тоді $f(n)=O(f_1(n))$.
3. Нехай $f_1(n)=O(g_1(n))$ і $f_2(n)=O(g_2(n))$. Тоді $f_1(n)\times f_2(n) = O((g_1(n)\times g_2(n))$ – правило добутку.
4. Нехай $f_1(n)=O(g_1(n))$. Тоді для будь-якої функції $g_2(n)$ виконується $f_1(n)\times g_2(n) = O((g_1(n)\times g_2(n))$.
5. Нехай f(n)=O(g(n)) і g(n)=O(h(n)). Тоді f(n) = O(h(n)) – транзитивність.

У таблиці нижче наведені найпоширеніші класи асимптотичної складності алгоритмів:

Складність	***Коментар***	***Приклади***
$O(1)$	Сталий час роботи не залежно від розміру задачі	Пошук у хеш-таблиці
$O(log_2 log_2 n)$	Дуже повільне зростання необхідного часу	Очікуваний час роботи інтерполюючого пошуку n елементів

$O(log_2 n)$	Логарифмічне зростання – подвоєння розміру задачі збільшує час роботи на сталу величину	Швидке обчислення xn; бінарний пошук у відсортованому масиві з n елементів
$O(n)$	Лінійне зростання – подвоєння розміру задачі подвоїть і необхідний час	Додавання/віднімання чисел з n цифр; лінійний пошук в масиві з n елементів
$O(nlog_2 n)$	Лінеаритмічне зростання – подвоєння розміру задачі збільшить необхідний час трохи більше ніж вдвічі	Сортування злиттям або купою масиву з n елементів.
$O(n^2)$	Квадратичне зростання – подвоєння розміру задачі вчетверо збільшує необхідний час	Елементарні алгоритми сортування масивів з n елементів; Лінійний пошук у квадратній матриці розмірності n.
$O(n^3)$	Кубічне зростання – подвоєння розміру задачі збільшує необхідний час у вісім разів.	Звичайне множення матриць
$O(a^n)$	Експоненціальне зростання – збільшення розміру задачі на 1 призводить до a-кратного збільшення необхідного часу; подвоєння	Деякі задачі комівояжера; Алгоритми пошуку повним перебором

Асимптотичний аналіз алгоритмів

Зазвичай, найбільший інтерес щодо дослідження алгоритму з практичної точки зору складає оцінка часу його роботи у найгіршому випадку. Це аргументовано тим що:

1) це дає оцінку швидкодії алгоритму для довільних вхідних даних визначеного розміру;

2) з практичної точки зору, «погані» вхідні дані трапляються досить часто;

3) час роботи в середньому є досить близьким до часу роботи у найгіршому випадку.

У попередніх темах ми подавали алгоритм у текстовій формі та у вигляді блок-схеми. Зручно також записувати алгоритм псевдомовою. Це подання алгоритму в текстовій формі близькій до мови програмування. У ній можна використовувати конструкції розгалужень та циклів.

Здебільшого визначення асимптотичної складності алгоритму переважно зводиться до аналізу циклів і рекурсивних викликів підпрограм.

При визначенні асимптотичної оцінки алгоритму будемо користуватися наступними правилами:

1. У будь-якій програмі є елементарні операції, що виконуються комп'ютером за фіксований час, який є константою. Асимптотична оцінка часу виконання таких операцій є O(1).
2. Нехай T(n) – це час виконання алгоритму або його частини. Якщо T(n) є многочленом степеня k, тобто $T(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$, то $T(n) = O(n^k)$.
3. Нехай алгоритм є послідовністю дій $S_1, S_2, \ldots, S_m$ і для кожної дії знайдена асимптотична оцінка $T_1(n) = O\big(g_1(n)\big), T_2(n) = O\big(g_2(n)\big), \ldots, T_m(n) = O\big(g_m(n)\big)$. Тоді асимтотична оцінка всієї послідовності дій за правилом суми дорівнює $T_1(n) + T_2(n) + \cdots + T_m(n) = O\big(\max(g_1(n), g_2(n), \ldots, g_m(n))\big)$.
4. (Цикл while). Найгірший час виконання алгоритму, що містить цикл while

   while S1:

   S2

   має асимптотичну складність $O(\max\{T1(n)\times(I(n)+1), T2(n)\times I(n)\})$, де $T1(n), T2(n)$ – час виконання відповідно інструкцій S1, S2, а $I(n)$ – кількість ітерацій циклу, що виконується у найгіршому випадку.
5. (Цикл for). Найгірший час виконання алгоритму, що містить цикл for

   for i in range(S1): S2

   має асимптотичну складність $O(\max\{T1(n)\times(I(n)+1), T2(n)\times I(n)\})$, де $T1(n), T2(n)$ – час виконання відповідно інструкцій S1, S2, а $I(n)$ – кількість ітерацій циклу, що виконується у найгіршому випадку.
6. (Умовний оператор). Найгірший час виконання алгоритму

   if S1: S2 else: S3

   має асимптотичну складність $O(\max\{T1(n), T2(n), T3(n)\})$, де $T1(n), T2(n), T3(n)$ – час виконання відповідно інструкцій S1, S2, S3.

Приклад 1

Нехай $f(n) = 8n + 128$. Довести, що $f(n) = O(n^2)$.

Доведення

Очевидно, що $f(n) \geq 0$ для всіх натуральних n. Згідно з означенням, нам треба знайти таке натуральне n_0 і таку сталу C>0, що для всіх $n \geq n_0$ виконується нерівність

$8n + 128 \leq Cn^2$.

Не має значення величина сталої – головне чи вона існує. Припустимо, що C=1. Тоді $8n + 128 \leq n^2 \Longrightarrow n^2 - 8n - 128 \geq 0 \Longrightarrow (n - 16)(n + 8) \geq 0$.

Очевидно, що остання нерівність виконується при n≥16. Таким чином, існує C=1 та існує n_0=16, що для всіх $n \geq n_0$ $8n+128 \leq n^2$. Отже, $f(n)=O(n^2)$.

Очевидно, що існує безліч значень таких пар C та n_0.

Приклад 2

Довести, що $7n^2 - 2n + 8 = O(n^2)$.

Доведення

Очевидно, що $f(n) \geq 0$ для всіх натуральних n. Згідно з означенням, нам треба знайти таке натуральне n_0 і таку сталу C>0, що для всіх $n \geq n_0$ виконується нерівність

$$7n^2 - 2n + 8 \leq Cn^2.$$

Покладемо C=8. Тоді

$$7n^2 - 2n + 8 \leq 8n^2 \Longrightarrow n^2 + 2n - 8 \geq 0 \Longrightarrow (n-2)(n+4) \geq 0.$$

Очевидно, що остання нерівність виконується при n≥2. Таким чином, існує C=8 та існує n_0=2, що для всіх $n \geq n_0$ $7n^2 - 2n + 8 \leq 8n^2$. Отже, $7n^2 - 2n + 8 = O(n^2)$.

Приклад 3

Дано вектор розмірності **n** дійсних чисел, представлений одновимірним масивом. Знайти суму компонент вектора.

Розробити алгоритм розв'язання задачі та записати його псевдомовою. Знайти асимптотичну оцінку цього алгоритму в найгіршому випадку.

Аналіз задачі

Потрібно визначити змінну suma до якої послідовно додавати компоненти вектора.

Алгоритм

Масив a[n] використовується для зберігання компонент вектора

```
1. for i = 1 to n
2.       ввести a[i]
3. suma = 0
4. for i = 1 to n
5.       suma += a[i]
6. вивести suma.
```

Зауваження. Алгоритм написаний псевдомовою, яка нагадує мову програмування. Команди алгоритму пронумеровані з метою знаходження його асимптотичної оцінки.

Створимо таблицю для знаходження оцінки O-велике для кожного рядка алгоритму.

Рядок	Оцінка	Коментар
1	O(n)	n разів параметру i присвоюється значення та виконується порівняння i з n. Операції присвоювання та порівняння мають складність O(1). За правилом добутку отримуємо оцінку O(n)
2	O(n)	n разів вводиться значення a[i]. Оцінка введення одного значення a[i] рівна O(1).
3	O(1)	Присвоювання виконується за фіксований час – оцінка O(1).
4	O(n)	Аналогічно рядку 1.
5	O(n)	Аналогічно рядку 2.
6	O(1)	Виведення виконується за фіксований час – оцінка O(1).

Загальна оцінка всього алгоритму обчислюється за правилом суми:

T(n) = O(max(n, n, 1, n, n, 1)) = O(n).

Задачі

1. Довести, що $nlog_2 n = O(n^2)$.
2. Довести, що $5n^2 - n + 12 = O(n^2)$.
3. Довести, що $345n^3 + 123n^2 + 98n + 15 = O(n^3)$.
4. Дані дві квадратні матриці порядку **n** цілих чисел, представлені двовимірним масивом. Знайти добуток матриць. Розробити алгоритм розв'язання задачі та записати його псевдомовою або мовою програмування. Знайти асимптотичну оцінку цього алгоритму в найгіршому випадку.

☺ Індивідуальні завдання

А

1. Довести, що $3n^2 + 2n - 1 = O(n^2)$.
2. Довести, що $2n^2 + n + 6 = O(n^2)$.
3. Довести, що $3n^2 + 3n + 4 = O(n^2)$.
4. Довести, що $3n^2 + 4 = O(n^2)$.
5. Довести, що $2n^2 + 2n + 3 = O(n^2)$.
6. Довести, що $4n^2 + n + 2 = O(n^2)$.
7. Довести, що $10n^2 + 1 = O(n^2)$.
8. Довести, що $6n^2 + 2n + 8 = O(n^2)$.
9. Довести, що $4n^2 + 3n + 4 = O(n^2)$.
10. Довести, що $5n^2 + n + 2 = O(n^2)$.

Б

Для поданих нижче задач розробити алгоритм їх розв'язання. Записати алгоритм псевдомовою або мовою програмування. Знайти асимптотичну оцінку цього алгоритму в найгіршому випадку.

1. Дано два вектори розмірності **n** дійсних чисел, представлені одновимірним масивом. Знайти різницю векторів.
2. Дано два вектори розмірності **n** дійсних чисел, представлені одновимірним масивом. Знайти скалярний добуток векторів.
3. Дано вектор розмірності **n** дійсних чисел, представлений одновимірним масивом. Знайти кількість найменших компонент вектора.
4. Дано вектор розмірності **n** дійсних чисел, представлений одновимірним масивом. Знайти найменший компонент вектора.
5. Дано вектор розмірності **n** дійсних чисел, представлений одновимірним масивом. Знайти кількість найбільших компонент вектора.

6. Дано вектор розмірності **n** дійсних чисел, представлений одновимірним масивом. Знайти найбільший компонент вектора.
7. Дано вектор розмірності **n** дійсних чисел, представлений одновимірним масивом. Вияснити, чи міститься серед компонент вектора число 0.
8. Дано два вектори розмірності **n** дійсних чисел, представлені одновимірним масивом. Знайти суму векторів.
9. Дано вектор розмірності **n** дійсних чисел, представлений одновимірним масивом. Підрахувати скільки компонент вектора рівні 0.
10. Дано вектор розмірності **n** дійсних чисел, представлений одновимірним масивом, та число a. Вияснити, чи співпадає якась компонента вектора з числом a.

B

Для поданих нижче задач розробити алгоритм їх розв'язання. Записати алгоритм псевдомовою або мовою програмування. Знайти асимптотичну оцінку цього алгоритму в найгіршому випадку.

1. Дана квадратна матриця порядку **n** цілих чисел, представлена двовимірним масивом. Знайти суми елементів кожного рядка матриці.
2. Дана квадратна матриця порядку **n** цілих чисел, представлена двовимірним масивом. Знайти суми елементів кожного стовпця матриці.
3. Дана квадратна матриця порядку **n** цілих чисел, представлена двовимірним масивом. Знайти кількість найменших елементів матриці.
4. Дана квадратна матриця порядку **n** цілих чисел, представлена двовимірним масивом. Знайти транспоновану матрицю.
5. Дана квадратна матриця порядку **n** цілих чисел, представлена двовимірним масивом. Знайти кількість найбільших елементів матриці.
6. Дана квадратна матриця порядку **n** цілих чисел, представлена двовимірним масивом. Знайти суму діагональних елементів матриці.
7. Дана квадратна матриця порядку **n** цілих чисел, представлена двовимірним масивом. Знайти суму елементів матриці.
8. Дані дві квадратні матриці порядку **n** цілих чисел, представлені двовимірним масивом. Знайти суму матриць.
9. Дані дві квадратні матриці порядку **n** цілих чисел, представлені двовимірним масивом. Знайти різницю матриць.
10. Дана квадратна матриця порядку n цілих чисел, представлена двовимірним масивом. Знайти найменший елемент кожного рядка матриці.

Тема 5: Машини з натуральнозначними регістрами

Теоретичні відомості

Машини з натуральнозначними регістрами (МНР) є ідеалізованою моделлю комп'ютера та прикладом формальної моделі алгоритму.

МНР містить нескінченну кількість регістрів, які є аналогом моделі пам'яті комп'ютера. У регістри записуються натуральні числа. Початкові значення регістрів вважаються рівними 0. Регістри нумеруються (іменуються) натуральними числами, починаючи з 0, і позначаються $R_0, R_1, \dots, R_n, \dots$ Вміст регістру R_i позначимо r_i.

Послідовність $(r_0, r_1, \dots, r_n, \dots)$ вмістів регістрів МНР назвемо *конфігурацією* МНР.

МНР може змінити вміст регістрів згідно виконуваної нею команди. Скінченний список команд утворює програму МНР. Команди програми послідовно нумеруємо натуральними числами, починаючи з 1. Номер команди в програмі називатимемо адресом команди.

Команди МНР бувають наступних чотирьох типів.

Тип 1. Обнулення n-го регістру Z(n): $r_n = 0$.

Тип 2. Збільшення вмісту n-го регістру на 1 S(n): $r_n = r_n + 1$.

Тип 3. Копіювання вмісту m-го регістру в n-й T(m,n): $r_n = r_m$ (при цьому r_m не змінюється).

Тип 4. Умовний перехід $J(m, n, q)$: якщо $r_n == r_m$, то перейти до виконання q-ї команди, інакше виконувати наступну за списком команду програми. Число q в команді $J(m, n, q)$ назвемо адресою переходу.

Команди типів 1-3 називають арифметичними. Після виконання арифметичної команди МНР повинна виконувати наступну за списком команду програми. Виконання однієї команди МНР назвемо кроком МНР.

Виконання програми МНР починає, перебуваючи в деякій початковій конфігурації, з виконання 1-ї за списком команди. Наступна для виконання команда програми визначається так, як описано вище. Виконання програми завершується (програма зупиняється), якщо наступна для виконання команда відсутня (тобто номер наступної команди перевищує номер останньої команди програми). Конфігурація МНР в момент завершення виконання програми називається фінальною, вона визначає результат роботи МНР-програми над даною початковою конфігурацією.

МНР-програми, як моделі алгоритмів, працюють лише зі скінченим набором даних, тому обмежимося розглядом скінченних конфігурацій. Конфігурацію вигляду $(r_0, r_1, \dots, r_n, 0, 0, \dots)$, в якій $r_m == 0$ для всіх m>n, назвемо скінченною. Таку конфігурацію позначимо $(r_0, r_1, \dots, r_n)$.

Кожну програму можна розглядати як алгоритм з множиною можливих початкових даних. Застосування такого алгоритму до початкових даних ($x_1,\dots,x_k$) полягає в наступному. У початковий момент натуральні числа $x_1,\dots, x_k$ поміщаються відповідно в регістри $R_0, R_1, \dots, R_{k-1}$ при цьому в решті інших регістрів міститься 0.

Потім виконуються команди даної програми, починаючи з першої. Один крок роботи алгоритму полягає у виконанні однієї команди. Послідовність кроків роботи алгоритму називається обчисленням. Обчислення завершується, коли в програмі немає команди, яку можна було б виконати. Це може відбутися, якщо

1) виконана остання команда програми, і ця команда була арифметичною;
2) при виконанні команди $J(m,n,q)$ виявилось, що $r_n == r_m$, але q більше за число команд в програмі;
3) при виконанні команди $J(m,n,q)$ виявилось, що $r_n! = r_m$, , але це була остання команда програми.

Завершення роботи алгоритму завжди вважається результативним. Результатом роботи алгоритму є натуральне число, **записане в регістрі R_0** у момент завершення обчислення. Таким чином, кожна програма обчислює часткову k-місну числову функцію.

Означення. Часткова функція $f: N^k \to N$, де N – множина натуральних чисел, називається МНР-обчислювальною, якщо існує програма для МНР, яка обчислює цю функцію.

Кожна МНР-програма обчислює безліч функцій, заданих на N, але, зафіксовуючи наперед арність функцій (тобто кількість компонент початкових конфігурацій), отримуємо, що кожна МНР-програма обчислює єдину функцію заданої арності.

Приклад 1

МНР-програма для функції x +3:

1. S(0)
2. S(0)
3. S(0)

Пояснення

Функція від однієї змінної x. Значення x заноситься в регістр R_0. МНР-програма тричі добавляє в R_0 по 1. Промоделюємо це на прикладі при x=5. Результат функції рівний 8.

R_0	Коментар
5	Початкове значення x
6	Виконання команди 1
7	Виконання команди 2
8	Виконання команди 3

Приклад 2

МНР-програма для функції x + y:

1. J(1,2,5)
2. S(0)
3. S(2)
4. J(0,0,1)

Пояснення

Функція від двох змінних x та y. Значення x та y заносяться в регістри R_0 та R_1 відповідно. Додатково використовується регістр R_2. У ньому за допомогою додавання 1 накопичується число y. Паралельно ці 1 додаються в R_0. Як тільки в R_2 отримаємо y, то в R_0 отримаємо x+y. Промоделюємо роботу програми на прикладі при x=5, y=3. Результат функції рівний 8.

R_0	R_1	R_2	**Коментар**
5	3	0	Початкові значення регістрів
5	3	0	Команда 1 порівнює значення регістрів R_1 та R_2. Вони не співпадають. Перехід на команду 2
6	3	0	Виконання команди 2
6	3	1	Виконання команди 3
6	3	1	Виконання команди 4 передає управління на команду 1. Команда 4 дозволяє організувати цикл у програмі
6	3	1	Команда 1 порівнює значення регістрів R_1 та R_2. Вони не співпадають. Перехід на команду 2
7	3	1	Виконання команди 2
7	3	2	Виконання команди 3
7	3	2	Виконання команди 4 передає управління на команду 1.
7	3	2	Команда 1 порівнює значення регістрів R_1 та R_2. Вони не співпадають. Перехід на команду 2
8	3	2	Виконання команди 2
8	3	3	Виконання команди 3
8	3	3	Виконання команди 4 передає управління на команду 1.
8	3	3	Команда 1 порівнює значення регістрів R_1 та R_2. Вони співпадають. Перехід на команду 5, якої немає у програмі. Програма припиняє роботу. У регістрі R_0 міститься результат – значення функції.

Приклад 3

МНР-програма для функції x – y:

1. J(0,1,5)
2. S(1)
3. S(2)
4. J(0,0,1)
5. T(2,0)

Пояснення

Функція від двох змінних x та y. Значення x та y заносяться в регістри R_0 та R_1 відповідно. Додатково використовується регістр R_2. У ньому за допомогою додавання

1 накопичується шукана різниця чисел. Паралельно ці 1 додаються в R_1. Як тільки в R_1 отримаємо x, то в R_2 отримаємо x-y.

Функція x-y не всюди визначена. Якщо x≥y, то функція приймає значення x-y. При x<y функція не визначена. У цьому випадку МНР-програма не повинна видати результат, програма виконує нескінченний цикл. Промоделюємо роботу програми на прикладах. Нехай x=5, y=5. Результат функції рівний 0.

R_0	**R_1**	**R_2**	**Коментар**
5	5	0	Початкові значення регістрів
5	5	0	Команда 1 порівнює значення регістрів R_0 та R_1. Вони співпадають. Перехід на команду 5
0	5	0	Виконання команди 5. Копіюємо вміст регістру R_2 в R_0.

Нехай x=7, y=4. Результат функції рівний 3.

R_0	**R_1**	**R_2**	**Коментар**
7	4	0	Початкові значення регістрів
7	4	0	Команда 1 порівнює значення регістрів R_0 та R_1. Вони не співпадають. Перехід на команду 2
7	5	0	Виконання команди 2
7	5	1	Виконання команди 3
7	5	1	Виконання команди 4 передає управління на команду 1. Команда 4 дозволяє організувати цикл у програмі
7	5	1	Команда 1 порівнює значення регістрів R_0 та R_1. Вони не співпадають. Перехід на команду 2
7	6	1	Виконання команди 2
7	6	2	Виконання команди 3
7	6	2	Виконання команди 4 передає управління на команду 1.
7	6	2	Команда 1 порівнює значення регістрів R_0 та R_1. Вони не співпадають. Перехід на команду 2
7	7	2	Виконання команди 2
7	7	3	Виконання команди 3
7	7	3	Виконання команди 4 передає управління на команду 1.
7	7	3	Команда 1 порівнює значення регістрів R_0 та R_1. Вони співпадають. Перехід на команду 5.
3	7	3	Виконання команди 5. Копіюємо вміст регістру R_2 в R_0. Програма припиняє роботу. У регістрі R_0 міститься результат – значення функції.

Нехай x=4, y=7. Результат функції невизначений.

R_0	R_1	R_2	**Коментар**
4	7	0	Початкові значення регістрів
4	7	0	Команда 1 порівнює значення регістрів R_0 та R_1. Вони не співпадають. Перехід на команду 2
4	8	0	Виконання команди 2
4	8	1	Виконання команди 3
4	8	1	Виконання команди 4 передає управління на команду 1. Команда 4 дозволяє організувати цикл у програмі
4	8	1	Команда 1 порівнює значення регістрів R_0 та R_1. Вони не співпадають. Перехід на команду 2. Очевидно, що значення регістрів R_0 та R_1 ніколи не будуть співпадати. Програма зациклюється.

Приклад 4

МНР-програма для функції [x / 3]:

1. J(0,2,9)
2. S(2)
3. J(0,2,9)
4. S(2)
5. J(0,2,9)
6. S(2)
7. S(1)
8. J(0,0,1)
9. T(1,0)

Пояснення

Функція f(x)=[x/3] обчислює цілу частину від ділення 3. Наприклад, f(7)=[7/3]=2, f(2)=[2/3]=0.

Значення x заносяться в регістр R_0. Додатково використовуються регістри R_1 та R_2. У регістрі R_2 за допомогою додавання 1 накопичується значення x, а в регістрі R_1 накопичується кількість трійок, тобто ціла частина від ділення на 3.

Промоделюємо роботу програми на прикладі. Нехай x=7.

Результат функції рівний 2.

R_0	R_1	R_2	**Коментар**
7	0	0	Початкові значення регістрів
7	0	0	Команда 1 порівнює значення регістрів R_0 та R_2. Вони не співпадають. Перехід на команду 2
7	0	1	Виконання команди 2.
7	0	1	Команда 3 порівнює значення регістрів R_0 та R_2. Вони не співпадають. Перехід на команду 4
7	0	2	Виконання команди 4.

R_0	R_1	R_2	**Коментар**
7	0	2	Команда 5 порівнює значення регістрів R_0 та R_2. Вони не співпадають. Перехід на команду 6
7	0	3	Виконання команди 6.
7	1	3	Виконання команди 7.
7	1	3	Команда 8 здійснює перехід на команду 1. Створення циклу.
7	1	3	Команда 1 порівнює значення регістрів R_0 та R_2. Вони не співпадають. Перехід на команду 2
7	1	4	Виконання команди 2.
7	1	4	Команда 3 порівнює значення регістрів R_0 та R_2. Вони не співпадають. Перехід на команду 4
7	1	5	Виконання команди 4.
7	1	5	Команда 5 порівнює значення регістрів R_0 та R_2. Вони не співпадають. Перехід на команду 6
7	1	6	Виконання команди 6.
7	2	6	Виконання команди 7.
7	2	6	Команда 8 здійснює перехід на команду 1.
7	2	6	Команда 1 порівнює значення регістрів R_0 та R_2. Вони не співпадають. Перехід на команду 2
7	2	7	Виконання команди 2.
7	2	7	Команда 3 порівнює значення регістрів R_0 та R_2. Вони співпадають. Перехід на команду 9
2	2	7	Виконання команди 9. Вміст регістру R_1 копіюється в R_0.

Приклад 5

МНР-програма для функції (x+2)/3:

1. S(0)
2. S(0)
3. J(0,2,14)
4. S(2)
5. S(3)
6. J(0,2,14)
7. S(2)
8. S(3)
9. J(0,2,14)
10. S(2)
11. S(1)
12. Z(3)
13. J(0,0,3)
14. J(3,4,16)
15. J(3,3,15)
16. T(1,0)

Пояснення

Функція f(x)=(x+2)/3 не всюди визначена. Наприклад, f(7)=(7+2)/3=3, а f(2)=(2+2)/3 не є натуральним числом і функція невизначена.

Значення x заносяться в регістр R_0. Спочатку в регістрі R_0 обчислимо x+2. Додатково використовуються регістри R_1, R_2 та R_3. У регістрі R_2 за допомогою

додавання 1 накопичується значення x, а в регістрі R_1 накопичується кількість трійок, тобто ціла частина від ділення на 3. У регістрі R_3 зберігатиметься остача від ділення на 3. Якщо ця остача буде рівна 0, то програма обчислить значення функції, в іншому випадку програма працюватиме нескінченно. Для перевірки остачі на рівність 0 використовується регістр R_4.

Промоделюємо роботу програми на прикладі. Нехай x=7. Результат функції рівний 3.

R_0	**R_1**	**R_2**	**R_3**	**R_4**	**Коментар**
7	0	0	0	0	Початкові значення регістрів
8	0	0	0	0	Виконання команди 1.
9	0	0	0	0	Виконання команди 2.
9	0	0	0	0	Команда 3 порівнює значення регістрів R_0 та R_2. Вони не співпадають. Перехід на команду 4
9	0	1	0	0	Виконання команди 4.
9	0	1	1	0	Виконання команди 5.
9	0	1	1	0	Команда 6 порівнює значення регістрів R_0 та R_2. Вони не співпадають. Перехід на команду 7
9	0	2	1	0	Виконання команди 7.
9	0	2	2	0	Виконання команди 8.
9	0	2	2	0	Команда 9 порівнює значення регістрів R_0 та R_2. Вони не співпадають. Перехід на команду 10
9	0	3	2	0	Виконання команди 10.
9	1	3	2	0	Виконання команди 11.
9	1	3	0	0	Виконання команди 12.
9	1	3	0	0	Команда 13 здійснює перехід на команду 3. Створення циклу.
9	1	3	0	0	Команда 3 порівнює значення регістрів R_0 та R_2. Вони не співпадають. Перехід на команду 4
9	1	4	0	0	Виконання команди 4.
9	1	4	1	0	Виконання команди 5.
9	1	4	1	0	Команда 6 порівнює значення регістрів R_0 та R_2. Вони не співпадають. Перехід на команду 7
9	1	5	1	0	Виконання команди 7.
9	1	5	2	0	Виконання команди 8.
9	1	5	2	0	Команда 9 порівнює значення регістрів R_0 та R_2. Вони не співпадають. Перехід на команду 10
9	1	6	2	0	Виконання команди 10.
9	2	6	2	0	Виконання команди 11.
9	2	6	0	0	Виконання команди 12.

R_0	R_1	R_2	R_3	R_4	**Коментар**
9	2	6	0	0	Команда 13 здійснює перехід на команду 3.
9	2	6	0	0	Команда 3 порівнює значення регістрів R_0 та R_2. Вони не співпадають. Перехід на команду 4
9	2	7	0	0	Виконання команди 4.
9	2	7	1	0	Виконання команди 5.
9	2	7	1	0	Команда 6 порівнює значення регістрів R_0 та R_2. Вони не співпадають. Перехід на команду 7
9	2	8	1	0	Виконання команди 7.
9	2	8	2	0	Виконання команди 8.
9	2	8	2	0	Команда 9 порівнює значення регістрів R_0 та R_2. Вони не співпадають. Перехід на команду 10
9	2	9	2	0	Виконання команди 10.
9	3	9	2	0	Виконання команди 11.
9	3	9	0	0	Виконання команди 12.
9	3	9	0	0	Команда 13 здійснює перехід на команду 3.
9	3	9	0	0	Команда 3 порівнює значення регістрів R_0 та R_2. Вони співпадають. Перехід на команду 14
9	3	9	0	0	Команда 4 порівнює значення регістрів R_3 та R_4. Вони співпадають. Перехід на команду 16
3	3	9	0	0	Копіюємо вміст регістру R_1 в R_0. Програма завершила роботу

Якщо значення регістрів R_3 та R_4 не будуть співпадати, то команда 14 не спрацює і почне виконуватися команда 15, яка реалізує нескінченний цикл.

Приклад 6

МНР-програма для всюди невизначеної функції:

1. J(0,0,1)

Пояснення

МНР-програма реалізовує нескінченний цикл. При будь-якому значенні x, яке помістимо в регістр R_0 результату не отримаємо.

Приклад 7

МНР-програма для функції $f(x, y) = x \div y$:

1. J(0,1,7)
2. J(0,2,6)
3. S(1)
4. S(2)
5. J(0,0,1)
6. Z(2)
7. T(2,0)

Пояснення

У прикладі 3 розглядалась часткова функція f(x,y)=x-y. Функція $f(x,y) = x \dot{-} y$ теж реалізує операцію віднімання, але вона всюди визначена. При x<y вона приймає значення 0.

Значення x та y заносяться в регістри R_0 та R_1. Додатково використовується регістр R_2. У регістрі R_2 за допомогою додавання 1 накопичується шукана різниця. Паралельно 1 додається в регістр R_1. Якщо x≥y, то в регістрі R_2 отримаємо шуканий результат. При x<y значення регістрів R_0 та R_1 ніколи не будуть рівними. Однак на якомусь кроці програми стануть рівними значення регістрів R_0 та R_2. Тоді виконається команда 6, яка обнулить регістр R_2. Командою 7 це значення буде скопійовано в R_0.

Приклад 8

МНР-програма для функції $f(x,y) = x \cdot y$:

1. J(3,1,9)
2. J(0,2,6)
3. S(2)
4. S(4)
5. J(0,0,2)
6. Z(2)
7. S(3)
8. J(0,0,1)
9. T(4,0)

Пояснення

Значення x та y заносяться в регістри R_0 та R_1. Додатково використовується регістри R_2, R_3, R_4. У програмі множення реалізовується через додавання.

У регістрі R_2 за допомогою додавання 1 накопичується значення x і паралельно 1 заноситься в R_4. Як тільки в R_2 отримаємо значення x, R_2 обнуляється, а в R_3 додається 1. У регістрі R_4 накопичується шуканий добуток, а в регістрі R_3 кількість разів додавання x в R_4. Як тільки ця кількість буде рівна y, програма завершиться.

Задачі

Скласти МНР-програму для функцій.

1) $f(x) = 3x - 2$
2) $f(x,y) = max(x,y)$
3) $f(x) = sg(x) + 2$
4) $f(x) = nsg(3x)$
5) $f(x,y,z) = (2x - 3y) + 2z$
6) $f(x) = [\frac{2x}{3}]$
7) $f(x) = x!$
8) $f(x,y) = x^y$

☺ Індивідуальні завдання

Скласти МНР-програму для функцій.

1. $f(x,y) = x - 2y + 1$
2. $f(x,y,z) = (x - y) + z$
3. $f(x,y) = nsg(x + y)$
4. $f(x,y) = sg(x \cdot y) + 2$
5. $f(x,y,z) = max(x,y) + z$
6. $f(x,y,z) = x - min(y,z)$
7. $f(x,y) = max(x,2y)$
8. $f(x) = (x+3)/4$
9. $f(x) = [\frac{x+2}{4}]$
10. $f(x,y) = 2x \dot{-} y$

Тема 6: Машини Тьюрінга

Теоретичні відомості

Машина Тьюрінга (МТ) є ідеалізованою моделлю комп'ютера та прикладом формальної моделі алгоритму. МТ – це така алгоритмічна система, в якій правила, що визначають дію алгоритму, побудовані за командно-адресним принципом, аналогічно сучасним комп'ютерам.

МТ має два скінченних алфавіти: *зовнішній* $A=\{a_0, a_1,..., a_n\}$ і *внутрішній* $Q=\{q_0, q_1,..., q_k\}$. У зовнішньому алфавіті записується вхідна, проміжна та вихідна інформація. Внутрішній алфавіт призначений для позначення (кодування) так званих внутрішніх станів машини.

Вхідна інформація записується на нескінченній в обидві сторони стрічці пам'яті (аналог пам'яті комп'ютера), яка розбита на окремі комірки, в кожній з яких записується один символ зовнішнього алфавіту. Вважається, що серед символів зовнішнього алфавіту є порожній символ, який будемо позначати λ.

Серед символів внутрішнього алфавіту (станів) $Q=\{q_0, q_1,..., q_k\}$ виділяється початковий стан q_0. У цьому стані МТ починає роботу. МТ може мати декілька кінцевих (фінальних) станів, у яких машина припиняє роботу. У нашому випадку машина має однин кінцевий стан, який позначимо $q*$.

Неформально МТ складається зі скінченної пам'яті, розділеної на клітинки нескінченної з обох боків стрічки та головки зчитування-запису (ЧЗ).

У *кожній* клітинці стрічки міститься тільки один символ алфавіту A, причому в кожний момент роботи МТ стрічка містить скінченну кількість символів, відмінних від символу λ. Головка ЧЗ в кожний момент оглядає єдину клітинку стрічки.

МТ виконує роботу відповідно до створеної програми. Програма складається зі скінченої множини команд одного із трьох видів:

$qa \to pbR$ $\qquad$ $qa \to pbL$ $\qquad$ $qa \to pb$

де $q, p \in Q,\ \ a, b \in A,$ інші символи є допоміжними.

Якщо МТ перебуває у стані q і головка ЧЗ зчитує зі стрічка 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 в комірку стрічки символ b і зміщує головку ЧЗ на одну клітинку направо (відповідно на одну клітинку наліво, залишає головку на місці).

Моделювати роботу МТ зручно за допомогою поняття конфігурації. *Конфігурація,* або *повний стан* МТ, – це слово виду xqy, де $x, y \in A^*$ (A^* – це множина всіх можливих слів над алфавітом A), q – стан МТ.

Неформально це означає, що на стрічці записано слово xy, тобто зліва і справа від xy можуть стояти лише символи λ, МТ знаходиться в стані q, головка читає перший символ підслова y.

Конфігурацію вигляду q_0x, де перший і останній символи слова x відмінні від λ, називають *початковою*.

Конфігурацію вигляду *xq*y* називають *фінальною*.

Після переходу до фінального стану, а отже, до фінальної конфігурації, МТ зупиняється.

Кожна МТ задає деяке вербальне відображення множини слів у множину слів над алфавітом *A*.

Машина Тьюрінга *M* переводить слово $u \in A^*$ у слово $v \in A^*$, якщо вона з початкової конфігурації q_0u переходить до фінальної конфігурації *xq*y*, де *xy=αvβ*, *α*, *β* – слова із символу λ. При цьому перший і останній символи слова *v* відмінні від λ або *v*=ε – порожнє слово.

Те, що МТ *M* переводить слово *u* у слово *v*, записують так: *v=M(u).*

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

Натуральні числа можна записувати на стрічці у вигляді слова із одиниць. Замість одиниць будемо використовувати символ |, який називатимемо «паличкою». Число 5 на стрічці можна записами зі слова з п'яти паличок – |||||, Це слово буде займати 5 комірок на стрічці, кожна паличка записується в окрему комірку. 5 паличок можна коротко записати $|^5$.

Загалом запис $|^x$ означає слово із x паличок.

Машина Тьюрінга *M обчислює* часткову функцію $f: N^k \to N$, якщо вона кожне слово вигляду $|^{x_1}\#|^{x_2}\# \dots \#|^{x_k}$ переводить у слово $|^{f(x_1,x_2,\dots,x_k)}$ у випадку, коли $(x_1, x_2, \dots, x_k)$ належить області визначення функції, і $M(|^{x_1}\#|^{x_2}\# \dots \#|^{x_k})$ не визначене, якщо $(x_1, x_2, \dots, x_k)$ не належить області визначення функції. Символ # використовується як роздільний символ між словами із паличок.

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

Як уже відмічалось, програма МТ складається зі скінченої множини команд одного із трьох видів:

$qa \to pbR$ $\qquad$ $qa \to pbL$ $\qquad$ $qa \to pb.$

Команди не нумеруються і їхній порядок у програмі не важливий. Кожна команда має ліву частину (до стрілки) і праву (після стрілки). Ми розглядаємо детерміновані (однозначні) МТ. Тому у програмі всі команди повинні мати різні ліві частини. Яка з команд буде виконуватися на кожному кроці роботи програми визначається *станом машини* і *символом, який зчитує головка зі стрічки*, тобто лівою частиною команди.

Приклад 1

Створити програму МТ для функції слідування S(n)=n+1, де n – натуральне число.

Програма

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

Пояснення

Для запису натурального числа n використаємо зовнішній алфавіт A={|} із одного символу – паличка. Кількість внутрішніх станів МТ залежить від розміру програми і визначається програмістом. У нашому випадку *Q={q_0, q*}*. Початкова конфігурація така: $q_0|^n$.

Змістовно можна запропонувати один з таких алгоритмів: головка ЧЗ зсувається вправо до тих пір, поки не зустріне символ λ (порожня комірка). Записуємо в комірку паличку і припиняємо роботу програми.

Промоделюємо роботу програми на прикладі n=4.

Стрічка	**Команда**	**Коментар**
λ | | | | λ λ		Початковий стан МТ. На стрічці записане слово із 4 паличок. Головка зчитує символ |.
q_0	$q_0\| \to q_0\|R$	Виконується команда з лівою частиною $q_0\|$, яка зсуває головку на одну клітинку вправо.
λ | | | | λ λ		
q_0	$q_0\| \to q_0\|R$	Виконується та сама команда.
λ | | | | λ λ		
q_0	$q_0\| \to q_0\|R$	Виконується та сама команда.
λ | | | | λ λ		
q_0	$q_0\| \to q_0\|R$	Виконується та сама команда.
λ | | | | λ λ		
q_0	$q_0\lambda \to q^*\|$	Виконується команда з лівою частиною $q_0\lambda$, яка змінює вміст комірки на |.
λ | | | | | λ		Машина переходить зі стану q_0 в q_* і припиняє роботу
q_*		

Приклад 2

Створити програму МТ для функції f(x,y)=x+y.

Програма

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

Пояснення

Функція залежить від двох змінних. Початкове слово на стрічці має вигляд $|^x\#|^y$ і для його запису використаний зовнішній алфавіт A={|, #}. У програмі використані три внутрішні стани. Початкова конфігурація така: $q_0|^x\#|^y$.

Пропонується наступний алгоритм. Якщо x не рівний нулю, замінимо першу паличку на λ і будемо рухати головку вправо, поки не зустрінеться символ #. Замінимо # на | та припинимо роботу програми.

Розглянемо роботу програми на прикладі. Замість зображення стрічки використаємо конфігурації. Нехай x=4, y=5.

Конфігурація	**Команда**	**Коментар**
q_0|||#||||		Початкова конфігурація.
λq_1|||#||||	$q_0\| \to q_1\lambda R$	Замінюємо | на λ (витираємо |), стан q_0 змінюємо на q_1, зсуваємо головку вправо.
λ|q_1||#||||	$q_1\| \to q_1\|R$	Зсуваємо головку вправо.
λ||q_1|#||||	$q_1\| \to q_1\|R$	Зсуваємо головку вправо.
λ|||q_1#||||	$q_1\# \to q^*\|$	Замінюємо # на |, стан q_1 змінюємо на q^*, припиняємо роботу.
λ|||q^*|||||		На стрічці слово із 9 паличок

У прикладі не використана остання команда $q_0\# \to q^*\lambda$ програми. Вона потрібна для випадку, коли x=0.

Приклад 3

Створити програму МТ для функції f(x,y)=x-y.

Програма

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

Пояснення

Функція залежить від двох змінних. Початкове слово на стрічці має вигляд $|^x\#|^y$ і для його запису використаний зовнішній алфавіт A={|, #}. Функція часткова і визначена при x≥y. У програмі використані п’ять внутрішні стани. Зазвичай внутрішні стани машини потрібно змінювати, якщо виникає конфліктна ситуація в програмі, зокрема, зациклювання. Початкова конфігурація така: $q_0|^x\#|^y$.

Пропонується наступний алгоритм. Витираємо зі стрічки спочатку крайню ліву паличку, а потім крайню праву. Повторюємо це до тих пір, поки справа від символу # не залишиться паличок. Це можливо при x≥y, тобто коли функція визначена. Фактично витремо у паличок. Замінимо # на | та припинимо роботу програми.

Розглянемо роботу програми на прикладі. Нехай x=8, y=3. На стрічці повинно залишитися 8-3=5 паличок.

Конфігурація	**Команда**	**Коментар**
q_0| | | | | | |#| | |λ		Початкова конфігурація.
λq_1| | | | | |#| | |λ	$q_0\| \to q_1\lambda R$	Замінюємо | на λ (витираємо |), стан q_0 змінюємо на q_1, зсуваємо головку вправо.
λ| | | | | | q_1#| | |λ	$q_1\| \to q_1\|R$	Зсуваємо головку вправо. Команду виконали 7 разів.
λ| | | | | |# q_1| | |λ	$q_1\# \to q_1\#R$	Зсуваємо головку вправо.
λ| | | | | |# | | | q_1λ	$q_1\| \to q_1\|R$	Зсуваємо головку вправо. Команду виконали 3 рази.
λ| | | | | |# | | | q_1λ	$q_1\lambda \to q_2\lambda L$	Міняємо стан. Зсуваємо головку вліво.
λ| | | | | |# | | q_2| λ	$q_2\| \to q_3\lambda L$	Витираємо |, міняємо стан, зсуваємо головку вліво.
λ| | | | | |# | q_3| λλ	$q_3\| \to q_3\|L$	Зсуваємо головку вліво. Команду виконали 2 рази.
λ| | | | | | q_3#| | λλ	$q_3\# \to q_3\#L$	Зсуваємо головку вліво.
λ| | | | |q_3 | #| | λλ	$q_3\| \to q_3\|L$	Зсуваємо головку вліво. Команду виконали 7 разів.
q_3λ| | | | | |#| | λλ	$q_3\lambda \to q_0\lambda R$	Міняємо стан. Зсуваємо головку вправо.
λq_0| | | | | |#| | λλ		

Ми витерли ліву паличку, а потім праву. Отримали конфігурацію аналогічну початковій. Повторивши ці дії ще двічі, отримаємо конфігурацію, у якій справа від # будуть порожні комірки.

λq_0 | | | | | # λ		Отримана конфігурація. Продовжуємо роботу програми.
λq_0 | | | | | # λ	$q_0\| \to q_1\lambda R$	Замінюємо | на λ (витираємо |), стан q_0 змінюємо на q_1, зсуваємо головку вправо.
λλ q_1 | | | | # λ	$q_1\| \to q_1\|R$	Зсуваємо головку вправо. Команду виконали 4 рази.
λλ | | | | q_1 # λ	$q_1\# \to q_1\#R$	Зсуваємо головку вправо.
λλ | | | | # q_1 λ	$q_1\lambda \to q_2\lambda L$	Міняємо стан. Зсуваємо головку вліво.
λλ | | | | q_2 # λ	$q_2\# \to q^*\|$	Символ # міняємо паличку. Припиняємо роботу. На стрічці 5 паличок.
λλ | | | | q^* | λ		

Друга частина дій програми демонструє її роботу при y=0.

Розглянемо роботу програми при x=3, y=5. У цьому випадку функція не визначена.

Конфігурація	**Команда**	**Коментар**
λq_0 | | |#| | | |λ		Початкова конфігурація. Витерши ліві три палички і три праві, отримаємо конфігурацію.
λq_0 #| |λ	$q_0\# \to q_4\lambda R$	Витираємо #, стан q_0 змінюємо на q_4, зсуваємо головку вправо.
λ λ q_4 | |λ		Програма не може продовжити роботу, бо відсутня необхідна команда. Результат програми невизначений – програма «зависла».

Команди

$q_0\# \to q_4\lambda R$

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

спрацюють у випадку, коли x=y.

Приклад 4

Створити програму МТ для функції f(x)=sg(x).

Програма

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

Пояснення

Функція f(x)=sg(x) визначена для всіх натуральних. Вона приймає значення 0 при x=0 і значення 1 при x>0.

Якщо x=0, то стрічка порожня. Виконується перша команда, яка припиняє роботу програми. Якщо x>0, виконуються решта команд. Вони залишають на стрічці одну паличку, а решту витирають.

Приклад 5

Створити програму МТ для функції $f(x, y) = x \cdot y$.

Програма

1. $q_0\# \to q_1\lambda R$
2. $q_1| \to q_1\lambda R$
3. $q_1\lambda \to q^*\lambda$
4. $q_1a \to q_1|R$
5. $q_0| \to q_2\lambda R$
6. $q_2| \to q_2|R$
7. $q_2\# \to q_3\#R$
8. $q_3| \to q_4\lambda R$
9. $q_4| \to q_4|R$
10. $q_4a \to q_4aR$
11. $q_4\lambda \to q_5aL$
12. $q_5| \to q_5|L$
13. $q_5a \to q_5aL$
14. $q_5\lambda \to q_3|R$
15. $q_3a \to q_6aL$
16. $q_6| \to q_6|L$
17. $q_6\# \to q_6\#L$
18. $q_6\lambda \to q_0\lambda R$
19. $q_3\lambda \to q_7\lambda L$
20. $q_7\# \to q_7\lambda L$
21. $q_7| \to q_7\lambda L$
22. $q_7\lambda \to q^*\lambda$

Пояснення

Функція залежить від двох змінних. Початкове слово на стрічці має вигляд $|^x\#|^y$ і для його запису використаний зовнішній алфавіт A={|, #}. Функція добуток всюди визначена. Початкова конфігурація така: $q_0|^x\#|^y$ при x і y відмінних від 0.

Програма працює наступним чином. Витираємо першу паличку слова $|^x$. Справа стрічки після слова $|^y$ у вільні комірки записуємо *y* разів символ *a*. Далі витираємо другу паличку слова $|^x$ і справа стрічки у вільні комірки записуємо *y* разів символ *a*. Продовжуємо цей процес до тих пір, поки не витремо всі палички слова $|^x$. Справа на стрічці кількість символів *a* буде рівна добутку *x* на *y*. Далі витираємо символ #, всі палички слова $|^y$, а символи *a* замінюємо на палички.

У даній програмі використаний допоміжний символ *a* зовнішнього алфавіту. Допоміжні символи потрібно використовувати тоді, коли потрібно розрізняти символи на стрічці. Пропонуємо перевірити роботу програми на конкретному прикладі.

Задачі

Створити програму МТ для функцій.

1) $f(x) = \left[\frac{x+2}{5}\right] + 3$
2) $f(x, y, z) = (2x - 3y) + 2z$
3) $f(x) = 3x - 2$
4) $f(x, y) = max(x, 2y)$
5) $f(x, y) = nsg(x \cdot y)$
6) $f(x) = (x + 2)/3$
7) $f(x) = x!$
8) $f(x, y) = x^y$

☺ Індивідуальні завдання

Створити програму МТ для функцій.

1. $f(x, y) = 2x + 3y$
2. $f(x, y) = |x - 2y|$
3. $f(x, y) = 3x + 2y + 1$
4. $f(x) = \left[\frac{x+1}{3}\right] + 2$
5. $f(x, y) = sg(x \cdot y) + 2$
6. $f(x, y) = max(x, y)$
7. $f(x, y) = min(x, y)$
8. $f(x) = \left[\frac{4x}{7}\right] + 1$
9. $f(x) = (x + 3)/4$
10. $f(x, y) = 2x \dot{-} y$

Тема 7: Нормальні алгоритми Маркова

Теоретичні відомості

Нормальні алгоритми – це алфавітні алгоритми, які перетворюють вхідне слово в деякому алфавіті у вихідне слово в тому ж самому чи іншому алфавіті.

Під *нормальним алгоритмом* (НА) *в алфавіті* T розуміють впорядковану послідовність правил підстановки вигляду $\alpha \to \beta$ або $\alpha \to\cdot \beta$, де $\alpha, \beta \in T^*$, $\to \notin T$, $\cdot \notin T$.

Правила вигляду $\alpha \to\cdot \beta$ називаються фінальними.

В нормальних алгоритмах використовується тільки один тип операторів дії – правила підстановки, і один тип операторів розпізнавання – оператори розпізнавання входження одного слова в інше слово. Для детального опису цих операторів розглянемо ряд понять.

Нехай α і β два слова в деякому алфавіті. Кажуть, що слово β входить в слово α, якщо слово α можна представити у вигляді $\alpha=\alpha_1\beta\alpha_2$, де α_1 і α_2 – деякі слова (можливо і порожні). Знайдене входження слова β в слово α називається першим зліва (або просто першим) входженням, якщо в представленні слова α у вигляді $\alpha=\alpha_1\beta\alpha_2$, слово α_1 не містить входження β.

Розпізнавач входження задається фіксацією деякого слова α, а зміст його застосування полягає в тому, що для будь-якого заданого слова α_1 перевіряється умова – входить чи ні слово α_1 в слово α.

Виконання правила підстановки $\alpha_1 \to \beta_1$ полягає в тому, що здійснюється підстановка слова β_1 замість першого зліва входження α_1 в задане слово α. Наприклад, якщо α_1 є першим зліва входженням в слово $\alpha=p\alpha_1 q$, то після застосування правила підстановки $\alpha_1 \to \beta_1$ слово α перетворюється в слово $\beta=p\beta_1 q$.

Якщо ліва частина підстановки $\alpha_1 \to \beta_1$, тобто слово α_1 входить у задане слово α, то кажуть, що підстановка $\alpha_1 \to \beta_1$ застосовна до слова α. Якщо ж ліва частина підстановки $\alpha_1 \to \beta_1$ не входить у задане слово α, то кажуть, що підстановка $\alpha_1 \to \beta_1$ незастосовна до слова α.

Зауважимо, що в нормальних алгоритмах можуть використовуватися фінальні підстановки, які записуються у вигляді $\alpha_1 \to\cdot \beta_1$ (замість стрілки пишемо стрілку з крапкою). Після виконання будь-якої фінальної підстановки дія алгоритму припиняється.

Нормальний алгоритм – це впорядкована скінченна сукупність підстановок, які застосовуються до вхідного слова у відповідності з такими правилами:

1. Перевірку застосовності підстановок до перетворюваного слова на будь-якому етапі його переробки треба починати з *першої підстановки*.
2. Якщо підстановка застосовна до слова, то після її виконання знову переходимо до п.1).
3. Якщо жодна з перших k підстановок незастосовна до слова, а (k+1)-ша застосовна, то після її виконання знову переходимо до п.1).

4. Процес перетворення вхідного слова продовжується доти, поки не дістанемо слово, до якого жодна з підстановок незастосовна або поки до слова не буде застосовна одна з фінальних підстановок.

Якщо в процесі обробки слова α НА не зупиняється, то вважається, що для слова α результат алгоритму невизначений.

Якщо НА U перетворює слово α в β, то це будемо записувати $U(\alpha)=\beta$.

Нормальний алгоритм U *обчислює* часткову функцію $f: N^k \to N$, якщо він кожне слово вигляду $|^{x_1}\#|^{x_2}\# \dots \#|^{x_k}$ переводить у слово $|^{f(x_1,x_2,\dots,x_k)}$ у випадку, коли $(x_1, x_2, \dots, x_k)$ належить області визначення функції, і $U(|^{x_1}\#|^{x_2}\# \dots \#|^{x_k})$ не визначене, якщо $(x_1, x_2, \dots, x_k)$ не належить області визначення функції.

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

Приклад 1

Нормальний алгоритм U_1 в алфавіті $A = \{a,b\}$ заданий упорядкованою послідовністю підстановок (програмою):

1. $bab \to aa$,
2. $aa \to b$,
3. $bb \to a$.

Знайти результат дії цього алгоритму на слово $\alpha = abaaabb$.

Пояснення

Перша підстановка до α незастосовна, а друга – застосовна. Виконавши її, одержимо слово $\alpha_1 = abbabb$. До α_1 застосовною є перша підстановка. Виконавши її, дістанемо слово $\alpha_2 = abaab$. До α_2 перша підстановка незастосовна. Застосувавши до α_2 другу підстановку, матимемо слово $\alpha_3 = abbb$. Перші дві підстановки до слова α_3 незастосовні. Виконавши третю підстановку, прийдемо до слова $\alpha_4 = aab$. До одержаного слова α_4 перша підстановка незастосовна. Виконавши далі послідовно другу, а потім третю підстановки, одержимо результат: $U\ (abaaabb) = a$, бо жодна підстановка до слова a уже незастосовна. Зауважимо ще раз, що перевірку застосовності підставок до слова кожного разу починаємо з першої підстановки у встановленому порядку.

Приклад 2

В алфавіті $A = \{a,b,\ c\}$ нормальний алгоритм U_2 заданий такою системою підстановок:

1. $ba \to ab$,
2. $ca \to ac$,
3. $cb \to bc$.

Знайти U_2(acbcab).

Пояснення

Коротко роботу алгоритму опишемо наступним чином, взявши в круглі дужки підслова, до який застосовується підстановка:

α = acb(ca)b – підстановка 2 $\Rightarrow$ α_1 = acbacb

α_1 = ac(ba)cb – підстановка 1 $\Rightarrow$ α_2 = acabcb

α_2 = a(ca)bcb – підстановка 2 $\Rightarrow$ α_3 = aacbcb

α_3 = aa(cb)cb – підстановка 3 $\Rightarrow$ α_4 = aabccb

α_4 = aabc(cb) – підстановка 3 $\Rightarrow$ α_5 = aabcbc

α_5 = aab(cb)c – підстановка 3 $\Rightarrow$ α_6 = aabbcc

До слова $\alpha_6 = aabbcc$ жодна підстановка не застосовна, отже $U_2(acbcab) = aabbcc$.

Детальний аналіз змісту підстановок цього алгоритму приводить до висновку, що він в будь-якому слові, заданому в алфавіті $\{a,b,c\}$, впорядковує його букви в алфавітному порядку.

Приклад 3

НА для функції f(x) = x+3.

1. #| → |#,
2. # →· |||,
3. ε → #.

Пояснення

Натуральне число x записуватимемо словом із паличок. До цього слова потрібно добавити символ #, щоб мати можливість в кінці початкового слова дописати три палички. Розглянемо приклад для x=5.

Слово	**Підстановка**	**Коментар**
|||||		Початкове слово
	ε → #	Символ ε позначає порожнє слово. Ця підстановка дозволяє добавити зліва у початкове слово символ #. Цю підстановку треба добавити в алгоритм останньою, щоб вона виконалася тільки один раз.
|#|||	#| → |#	
||#||	#| → |#	
|||#|	#| → |#	
||||#|	#| → |#	
|||||#	#| → |#	5 разів виконали першу підстановку
||||||||	# →· |||	Фінальна підстановка. Алгоритм припинив роботу.

Приклад 4

НА для функції f(x,y) = x+y.

1. # → ε.

Пояснення

Початкове слово для функції має вигляд $|^x\#|^y$. Підстановка замінює символ # на порожнє підслово і отримується шукане слово $|^x|^y$. Алгоритм припиняє роботу.

Приклад 5

НА для функції f(x,y) = x-y.

1. $|\#| \to \#$,
2. $\#| \to \#|$,
3. $\# \to \varepsilon$.

Пояснення

Початкове слово для функції має вигляд $|^x\#|^y$. Спочатку буде використовуватися перша підстановка стільки разів, скільки можливо. Якщо x≥y, то функція визначена і її значення рівне x-y. У цьому випадку після завершення застосувань першої підстановки отримаємо слово $|^{x-y}\#$. Далі застосовується третя підстановка, результатом якої є слово $|^{x-y}$. Алгоритм припиняє роботу.

Якщо x<y, то функція невизначена. У цьому випадку після завершення застосувань першої підстановки отримаємо слово $\#|^{y-x}$. Почне виконуватись друга підстановка, яка застосовуватиметься нескінченно. Алгоритм зациклиться.

Приклад 6

НА для функції f(x) = [x/3].

1. $\#||| \to |\#$,
2. $\#| \to \#$,
3. $\# \to\cdot \varepsilon$,
4. $\varepsilon \to \#$.

Пояснення

Дана функція обчислює цілу частину від ділення на 3. Початкове слово має вигляд $|^x$. Воно складається з x паличок.

Спочатку застосовується четверта підстановка і отримуємо слово $\#|^x$. Далі застосовується перша підстановка певну кількість разів, яка замінює 3 палички на одну. Реально з її допомогою знаходимо цілу частину. Потім застосовується друга підстановка, яка витирає палички, що утворюють остачу від ділення. Нарешті застосовується третя фінальна підстановка, яка знищує символ #. Алгоритм завершив роботу.

Приклад 7

НА для функції f(x) = x/3.

1. $\#||| \to |\#$,
2. $\#| \to \#|$,
3. $\# \to\cdot \varepsilon$,
4. $\varepsilon \to \#$.

Пояснення

Дана функція визначена тільки для чисел, що діляться на 3. Початкове слово має вигляд $|^x$. Воно складається з x паличок.

Спочатку застосовується четверта підстановка і отримуємо слово $\#|^x$. Далі застосовується перша підстановка певну кількість разів, яка замінює 3 палички на одну. Реально з її допомогою знаходимо цілу частину.

Якщо x не ділиться на 3, то справа від символу # буде одна або дві палички. У цьому випадку застосовується друга підстановка, яка зациклює роботу алгоритму.

Якщо x ділиться на 3, то справа від символу # не буде паличок. У цьому випадку застосовується третя фінальна підстановка, яка знищує символ #. Алгоритм завершив роботу.

Приклад 8

НА для функції f(x,y) = x·y.

1. $\# \to **$,
2. $| * \to * a$,
3. $* | \to b *$,
4. $* \to \varepsilon$,
5. $ab \to ba|$,
6. $|b \to b|$,
7. $a \to \varepsilon$.
8. $b \to \varepsilon$.

Пояснення

Функція обчислює добуток двох натуральних чисел. Початкове слово для функції має вигляд $|^x\#|^y$. Потрібно кожну паличку слова $|^x$ замінити на у паличок. Це можна зробити, добавивши додаткові символи в початкове слово. Розглянемо роботу алгоритму при x=3, y=4.

Слово	**Підстановка**	**Коментар**
|||#||||		Початкове слово
	$\# \to **$	Застосована перша підстановка. Вводяться додаткові символи *. Лівий *буде використовуватися для слова$\|^x$, а правий – для $\|^y$.
|||**||||		
||* a *||||	$\| * \to * a$	Застосована друга підстановка. Символ a використовується для перейменування паличок слова $\|^x$.
*aaa *||||		Застосували ще двічі другу підстановку
*$aaa\ b$ *|||	$* \| \to b *$	Застосували третю підстановку. Символ b використовується для перейменування паличок слова $\|^y$.
**aaa bbbb *		Застосували третю підстановку ще тричі.
aaabbbb	$* \to \varepsilon$	Четверта підстановка витирає символ *.

Слово	Підстановка	Коментар
aaba | bbb	$ab \to ba\|$	Застосована п’ята підстановка.
ba|a|a| bbb		Застосована п’ята підстановка ще двічі. Почало виконуватися множення.
ba|a|a|bbb	$\|b \to b\|$	Застосована шоста підстановка.
ba|a|ab|bb	$ab \to ba\|$	
ba|a|ba||bb	$\|b \to b\|$	
ba|ab|a||bb	$ab \to ba\|$	
ba|ba||a||bb	$\|b \to b\|$	
bab|a||a||bb	$ab \to ba\|$	
bba||a||a||bb	$\|b \to b\|$	Підстановку застосовуємо двічі
bba||a||ab||b	$ab \to ba\|$	
bba||a||ba|||b	$\|b \to b\|$	Підстановку застосовуємо двічі
bba||ab||a|||b	$ab \to ba\|$	
bba||ba|||a|||b	$\|b \to b\|$	Підстановку застосовуємо двічі
bbab||a|||a|||b	$ab \to ba\|$	
bbba|||a|||a|||b	$\|b \to b\|$	Підстановку застосовуємо тричі
bbba|||a|||ab|||	$ab \to ba\|$	
bbba|||a|||ba||||	$\|b \to b\|$	Підстановку застосовуємо тричі
bbba|||ab|||a||||	$ab \to ba\|$	
bbba|||ba||||a||||	$\|b \to b\|$	Підстановку застосовуємо тричі
bbbab|||a||||a||||	$ab \to ba\|$	
bbbba||||a||||a||||	$a \to \varepsilon$	Застосовуємо 3 рази
bbbb|||||||||||	$b \to \varepsilon$	Застосовуємо 4 рази
||||||||||		Шукане слово

Приклад 9

НА для функції f(x,y) = 2^x.

1. $* \mid \to \mid **$,
2. $\mid *\to *$,
3. $\# *\to \mid \#$,
4. $\# \to \cdot \varepsilon$,
5. $*\to \# *$,
6. $\varepsilon \to *$.

Пояснення

Початкове слово для функції має вигляд $|^x$. Потрібно добавити додаткові символи, бо слово складається тільки з паличок

Розглянемо роботу алгоритму при x=3.

Слово	Підстановка	Коментар
|||		Початкове слово
*|||	$\varepsilon \rightarrow *$	Застосована шоста підстановка.
|**||	$* \mid \rightarrow \mid **$	Застосована перша підстановка тричі.
|*|**|	$* \mid \rightarrow \mid **$	
||****|	$* \mid \rightarrow \mid **$	
||***|**	$* \mid \rightarrow \mid **$	
||**|****	$* \mid \rightarrow \mid **$	
||*|******	$* \mid \rightarrow \mid **$	
|||********	$* \mid \rightarrow \mid **$	Результат застосування першої підстановки.
********	$\mid *\rightarrow *$	Другу підстановку застосували тричі.
#********	$*\rightarrow \# *$	П’ята підстановка.
|********	$\# *\rightarrow \mid \#$	Третя підстановка.
|||||||#	$\# *\rightarrow \mid \#$	Сім разів застосована третя підстановка.
|||||||	$\# \rightarrow \cdot \varepsilon$	Четверта підстановка фінальна

Задачі

Створити НА для функцій.

1) $f(x) = \left[\frac{x+2}{5}\right] + 3$
2) $f(x, y, z) = (2x - 3y) + 2z$
3) $f(x) = 3x - 2$
4) $f(x, y) = min(x, 2y)$
5) $f(x, y) = nsg(x \cdot y)$
6) $f(x) = (x + 2)/3$
7) $f(x) = x!$
8) $f(x, y) = x^y$
9) $f(x, y) = 3^x - 2y$

☺ Індивідуальні завдання

Створити НА для функцій.

1. $f(x) = \left[\frac{2x}{7}\right] + 1$
2. $f(x) = \left[\frac{3x}{4}\right] + 3$
3. $f(x) = 3x + 2$
4. $f(x) = 5x - 1$
5. $f(x, y) = |x - y|$
6. $f(x, y) = 2x + y$
7. $f(x, y) = x + 3y$
8. $f(x) = (x + 3)/4$
9. $f(x) = \left[\frac{x+1}{3}\right] + 2$
10. $f(x, y) = max(x, y)$

Тема 8: Рекурсивні функції

Теоретичні відомості

Історично першою алгоритмічною системою є система, запропонована в 1936 році американським математиком А. Чорчом, яка базується на використанні конструктивно означених арифметичних (цілозначних) функцій, які одержали назву рекурсивних функцій. Застосування подібних функцій в теорії алгоритмів ґрунтується на ідеї нумерації слів в довільному скінченному алфавіті послідовними натуральними числами.

Функція, для якої існує алгоритм обчислення її значень, називають *обчислюваною* функцією. Оскільки загальне поняття алгоритму не є математично точним, то таким же є і поняття обчислювальної функції, яке потрібно уточнити.

Розглянемо спосіб задання обчислюваних функцій, який ґрунтується на породженні таких функцій за допомогою певних обчислюваних операцій (композицій) із певних базових функцій.

Основними обчислюваними операціями для n-арних функцій, заданих на множині N, є такі операції:

- операція суперпозиції $\mathbf{S}^{n+1}$,
- операція примітивної рекурсії **R**,
- операція мінімізації **M**.

Серед обчислюваних функцій виділимо найбільш прості функції, які називають *базовими обчислюваними n-арними функціями*.

1. **o**(x)=0 – константа нуль або нуль-функція,
2. **s**(x)=x+1 – функція слідування,
3. функції селектори $I_m^n(x_1, \dots, x_n) = x_m$, де n≥m≥1.

Із базових функцій за допомогою трьох операцій – суперпозиції (підстановки), примітивної рекурсії та операції найменшого кореня (операція мінімізації) будуються більш складні обчислювані функції. При цьому в обчислюваних функціях можна використовувати додаткові, так звані фіктивні аргументи, від яких функція фактично не залежить.

Операція суперпозиції $\boldsymbol{S}^{n+1}$ за відомою n-арною функцією $g(x_1, \dots, x_n)$ та відомими n функціями $g_1(x_1, \dots, x_m), \dots, g_n(x_1, \dots, x_m)$ однієї і тієї ж арності утворює нову функцію

$f(x_1, \dots, x_m) = g(g_1(x_1, \dots, x_m), \dots, g_n(x_1, \dots, x_m))$.

Таку функцію позначають $S^{n+1}(g, g_1, \dots, g_n)$, її арність збігається з арністю функцій $g_1, \dots, g_n$. Позначення S^{n+1} означає суперпозицію $n+1$ функції.

Твердження 1. Якщо функції $g, g_1, \dots, g_n$ всюди визначені та алгоритмічно обчислювані, то функція $S^{n+1}(g, g_1, \dots, g_n)$ також всюди визначена та алгоритмічно обчислювана.

Приклад 1

Функції константи.

- Візьмемо нуль-функцію $o(x_1) = 0$ та функцію селектор $I_1^n(x_1, \dots, x_n) = x_1$. Виконаємо підстановку $o\left(I_1^n(x_1, \dots, x_n)\right) = 0$. Отримали нову функцію $o^n(x_1, \dots, x_n) = 0$, яка є нуль-функцією від n аргументів. Ця функція є суперпозицією $S^2(o, I_1^n)$.
- Візьмемо функцію слідування $s(x) = x + 1$ та нуль-функцію $o^n(x_1, \dots, x_n) = 0$. Виконаємо підстановку $s(o^n(x_1, \dots, x_n)) = o^n(x_1, \dots, x_n) + 1 = 1$. Отримали нову функцію $1^n(x_1, \dots, x_n) = 1$ – константна функція одиниця. Вона є суперпозицією $S^2(s, o^n) = S^2(s, S^2(o, I_1^n))$. Із функцій $s(x)$ та 1^n аналогічною суперпозицією можна отримати константну функцію двійку $2^n(x_1, \dots, x_n) = 2$. Загалом, для довільного натурального числа k можна отримати відповідну константну функцію $k^n(x_1, \dots, x_n) = k$. Відповідна суперпозиція має вигляд $S^2\left(s, S^2(s, \dots, S^2\left(s, S^2(o, I_1^n)\right)\right)\right)$.

Приклад 2

Окремими випадками операції суперпозиції є операції перестановки аргументів та ідентифікації аргументів.

Так, функція $f(x, y) = g(y, x)$ утворюється з функції $g(y, x)$ підстановкою: $f(x, y) = g(I_2^2(x, y), I_1^2(x, y))$, тобто $f(x, y) = S^3(g, I_2^2, I_1^2)$.

Функція $f(x) = g(x, x)$ утворюється з функції $g(y, x)$ підстановками: $f(x) = g(I_1^2(x, y), I_1^2(x, y))$.

Операція примітивної рекурсії R дає можливість будувати (n+1)-арну функцію f (функцію від n+1 аргументів) за двома заданими функціями, одна з яких n-арна g, а друга – (n+2)-арна h. Ця операція визначається такими двома співвідношеннями:

$$f(x_1, \dots, x_n, 0) = g(x_1, \dots, x_n);$$
$$f(x_1, \dots, x_n, y + 1) = h\left(x_1, \dots, x_n, y, f(x_1, \dots, x_n, y)\right),$$

які називають *схемою примітивної рекурсії* і позначають $f = R(g, h)$. Друге співвідношення реалізує рекурсивне обчислення функції f.

За схемою примітивної рекурсії для всіх $a_1, \dots, a_n, b$ значення $f(a_1, \dots, a_n, b)$ обчислюється так:

$$f(a_1, \dots, a_n, 0) = g(a_1, \dots, a_n);$$
$$f(a_1, \dots, a_n, 1) = h\left(a_1, \dots, a_n, 0, f(a_1, \dots, a_n, 0)\right);$$
$$f(a_1, \dots, a_n, 2) = h\left(a_1, \dots, a_n, 1, f(a_1, \dots, a_n, 1)\right);$$
$$\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots$$
$$f(a_1, \dots, a_n, b) = h\left(a_1, \dots, a_n, b - 1, f(a_1, \dots, a_n, b - 1)\right).$$

Наведені співвідношення показують, що функція f однозначно визначається за функціями g і h.

Якщо для деяких $a_1, \dots, a_n, b$ значення $f(a_1, \dots, a_n, b)$ невизначене, то значення $f(a_1, \dots, a_n, t)$ невизначені для всіх $t \geq b$.

При n=0 за визначення вважаємо, що функція g – це 1-арна константа.

Твердження 2. Якщо функції g і h всюди визначені та алгоритмічно обчислювані, то функція $R(g, h)$ також всюди визначена та алгоритмічно обчислювана.

Означення 1. Функції, які будуються з базових обчислюваних функцій за допомогою операцій суперпозиції і примітивної рекурсії, застосованих скінченне число разів у довільній послідовності, називаються ***примітивно-рекурсивними функціями*** **(ПРФ)**.

Оскільки базові функції всюди визначені, а операції суперпозиції і примітивної рекурсії зберігають всюди визначеність побудованих функцій, то всі примітивно-рекурсивні функції всюди визначені.

Більшість відомих обчислюваних функцій належать до класу примітивно-рекурсивних функцій.

Приклад 3

Функція $f(x_1, x_2) = x_1 + x_2$ – ПРФ.

Доведення

Справді, маємо

$$f(x_1, 0) = x_1 = I_1^1(x_1);$$

$$f(x_1, x_2 + 1) = x_1 + (x_2 + 1) = (x_1 + x_2) + 1 = f(x_1, x_2) + 1 = h(x_1, x_2, f(x_1, x_2)).$$

Це схема примітивної рекурсії для функції $f(x_1, x_2)$. Потрібно визначити функції g і h. Очевидно $g(x_1) = I_1^1(x_1)$. Функцію h потрібно обрати таку $h(x_1, x_2, x_3) = x_3 + 1 = s(x_3) = S^2(s, I_3^3)(x_1, x_2, x_3)$. Функції g і h отримані з базових функцій, а $f(x_1, x_2) = x_1 + x_2 = R(g, h)$. Доведення завершене.

Приклад 4

Функція $f(x_1, x_2) = x_1 \cdot x_2$ – ПРФ.

Доведення

Розглянемо схему примітивної рекурсії

$$f(x_1, 0) = x_1 \cdot 0 = 0 = o(x_1);$$

$$f(x_1, x_2 + 1) = x_1 \cdot (x_2 + 1) = x_1 + x_1 \cdot x_2 = x_1 + f(x_1, x_2) = h(x_1, x_2, f(x_1, x_2)).$$

Отже функція $f(x_1, x_2)$ отримується примітивною рекурсією з функцій $g(x_1) = o(x_1)$ та $h(x_1, x_2, x_3) = x_1 + x_3$. У прикладі 3 доведено, що функція h є ПРФ. Тому Функція $f(x_1, x_2) = x_1 \cdot x_2$ – ПРФ.

Приклад 5

Функція $sg(x_1) = \begin{cases} 0, & \text{якщо } x_1 = 0, \\ 1, & \text{якщо } x_1 \geq 1. \end{cases}$ – ПРФ.

Доведення

Справді, маємо

$$sg(0) = 0 = o(x_1);$$
$$sg(x_1 + 1) = 1 = h(x_1, x_2).$$

Отже, $sg(x_1)$ одержується схемою примітивної рекурсії з функцій констант $o(x_1) = 0$ та $h(x_1, x_2) = 1$, які є примітивно рекурсивними функціями.

Приклад 6

Функція $nsg(x_1) = \begin{cases} 1, & \text{якщо } x_1 = 0, \\ 0, & \text{якщо } x_1 \geq 1. \end{cases}$ – ПРФ.

Доведення

Справді, маємо

$nsg(0) = 1 = s\big(o(x_1)\big) = S^2(s, o);$

$nsg(x_1 + 1) = 0 = h(x_1, x_2) = S^2(o, I_1^2).$

Отже, $nsg(x_1) = R(S^2(s, o), S^2(o, I_1^2)).$

Приклад 7

Функція $f(x_1, x_2) = x_1 \dot{-} x_2 = \begin{cases} x_1 - x_2, & \text{якщо } x_1 \geq x_2 \\ 0, & \text{якщо } x_1 < x_2 \end{cases}$ – ПРФ.

Доведення

Спочатку покажемо, що функція $f(x_1) = x_1 \dot{-} 1$ є ПРФ.

Справді, маємо

$f(0) = 0 \dot{-} 1 = 0 = o(x_1);$

$f(x_1 + 1) = (x_1 + 1) \dot{-} 1 = x_1 = I_1^2(x_1, x_2).$

Таким чином, функція $f(x_1) = x_1 \dot{-} 1$ отримується примітивною рекурсією з базових функцій $o(x_1)$ та $I_1^2(x_1, x_2)$, а тому є примітивно рекурсивною.

Тепер покажемо, що функція $f(x_1, x_2) = x_1 \dot{-} x_2$ є ПРФ.

Маємо

$f(x_1, 0) = x_1 \dot{-} 0 = x_1 = I_1^1(x_1);$

$f(x_1, x_2 + 1) = x_1 \dot{-} (x_2 + 1) = (x_1 \dot{-} x_2) \dot{-} 1 = f(x_1, x_2) \dot{-} 1 = h(x_1, x_2, f(x_1, x_2)).$

Отже функція $f(x_1, x_2)$ отримується примітивною рекурсією з функцій $g(x_1) = I_1^1(x_1)$ та $h(x_1, x_2, x_3) = x_3 \dot{-} 1$, які є примітивно рекурсивними.

Приклад 8

Функція $f(x_1, x_2) = |x_1 - x_2|$ – ПРФ.

Доведення

Для функції виконується рівність

$$f(x_1, x_2) = |x_1 - x_2| = (x_1 \dot{-} x_2) + (x_2 \dot{-} x_1).$$

Згідно з прикладом 7 функції $x_1 \dot{-} x_2$ та $x_2 \dot{-} x_1$ є ПРФ. Сума цих функцій теж ПРФ, відповідно до прикладу 3.

Приклад 9

Функція $f(x_1) = x_1!$ – ПРФ.

Доведення

За визначенням $f(0) = 0! = 1$. Маємо схему примітивної рекурсії

$$f(0) = 0! = 1;$$

$$f(x_1 + 1) = (x_1 + 1)! = x_1! \cdot (x_1 + 1) = (x_1 + 1) \cdot f(x_1).$$

Функція $g(x_1) = 0! = 1$ константа і є ПРФ.

Функція $h(x_1, x_2) = (x_1 + 1) \cdot x_2 = s(x_1) \cdot x_2$ є добутком, а значить ПРФ.

Отже $f(x_1) = R(g, h)$ є ПРФ.

Приклад 10

Функція $f(x_1, x_2) = \max(x_1, x_2)$ – ПРФ.

Доведення

Значення функції дорівнює більшому з пари чисел $\{x_1, x_2\}$. Доведення, що функція є примітивно рекурсивною слідує з того, що вона задається наступною формулою:

$\max(x_1, x_2) = x_1 \cdot sg(x_1 \dot{-} x_2) + x_2 \cdot sg(x_2 \dot{-} x_1)$.

Множина примітивно рекурсивних функцій зліченна. Існують такі обчислювані функції, які не є примітивно рекурсивними. Для розширення класу примітивно рекурсивних функцій вводиться нова операцію, так звана операція найменшого кореня або операція мінімізації.

Операція мінімізації M (*n*+1)-арній функції g ставить у відповідність *n*-арну функцію *f*, яку позначають *M(g)*, що задається співвідношенням

$f(x_1, \dots, x_n) = \mu_y[g(x_1, \dots, x_n, y) = 0]$.

Це означає, що для всіх значень $x_1, \dots, x_n$ значення функції $f(x_1, \dots, x_n)$ обчислюється так.

Послідовно обчислюємо значення функції $g(x_1, \dots, x_n, y)$ для всіх *y*=0, 1, 2,… .

Перше таке значення *y*, для якого $g(x_1, \dots, x_n, y) = 0$, буде *шуканим значенням функції* $f(x_1, \dots, x_n)$. При цьому для всіх $t<y$ значення $g(x_1, \dots, x_n, t)$ мусять бути визначені та не дорівнювати нулю.

Процес знаходження значення $\mu_y[g(x_1, \dots, x_n, y) = 0]$ ніколи не закінчиться в таких випадках:

- значення $g(x_1, \dots, x_n, 0)$ не визначене;
- для всіх значень *y* значення $g(x_1, \dots, x_n, y)$ визначене та не дорівнює нулю;
- для всіх $t<y$ значення $g(x_1, \dots, x_n, t)$ визначене та не дорівнює нулю, а значення $g(x_1, \dots, x_n, y)$ не визначене.

Твердження 3. Якщо функція g алгоритмічно обчислювана, то функція $M(g)$ також алгоритмічно обчислювана.

Означення 1. Функції, які будуються з базових обчислюваних функцій за допомогою скінченного числа операцій суперпозиції, примітивної рекурсії та мінімізації, називаються ***частково рекурсивними функціями*** (ЧРФ).

Всюди визначені частково рекурсивні функції називаються ***рекурсивними функціями*** (РФ).

Між класами функцій ПРФ, РФ, ЧРФ мають місце такі співвідношення

$$\text{ПРФ} \subseteq \text{РФ} \subseteq \text{ЧРФ}.$$

Приклад 11

Функція $f(x_1, x_2) = x_1 - x_2$ – ЧРФ.

Доведення

Функція $f(x_1, x_2) = x_1 - x_2$ є частковою. Вона визначена при $x_1 \geq x_2$. Нехай $x_1 \geq x_2$. Тоді за визначенням різниці існує таке x_3, що $x_1 = x_2 + x_3$. Отже $f(x_1, x_2) = x_1 - x_2 = \mu_{x_3}[x_1 = x_2 + x_3] = \mu_{x_3}[|x_1 - (x_2 + x_3)| = 0]$. Таким чином, функція $g(x_1, x_2, x_3) = |x_1 - (x_2 + x_3)|$ є ПРФ і f=M(g).

Приклад 12

Функція $f(x_1, x_2) = [x_1/x_2]$ – ЧРФ.

Доведення

Функція $f(x_1, x_2) = [x_1/x_2]$ – це ціла частина від ділення. Вона є частковою, бо невизначена при x_2=0.

Нехай $[x_1/x_2] = x_3$. Тоді $\frac{x_1}{x_2} < x_3 + 1$ або $x_2(x_3 + 1) > x_1$. Отже

$[x_1/x_2] = \mu_{x_3}[x_2(x_3 + 1) > x_1] = \mu_{x_3}[nsg(x_2(x_3 + 1) \dot{-} x_1) = 0]$.

Приклад 13

Функція $f(x_1) = [\sqrt{x_1}]$ – РФ.

Доведення

Функція $f(x_1) = [\sqrt{x_1}]$ це ціла частина від кореня і вона всюди визначена, бо x_1 натуральне число.

Нехай $[\sqrt{x_1}] = x_2$. Тоді $(x_2 + 1)\ (x_2 + 1) > x_1$. Отже

$[\sqrt{x_1}] = \mu_{x_2}[(x_2 + 1)\ (x_2 + 1) > x_1] = \mu_{x_3}[nsg((x_2 + 1)\ (x_2 + 1) \dot{-} x_1) = 0]$.

Приклад 14

Всюди невизначена функція $f_\emptyset$ є ЧРФ.

Доведення

Справді, $f_\emptyset(x_1) = \mu_{x_1}[x_1 + 1 = 0]$ і f=M(s).

Задачі

Довести, що наступні функції є ПРФ.

1) $f(x_1, x_2) = [x_1/x_2]$ (при x_2=0 вважаємо $[x_1/0] = x_1$);
2) $f(x_1, x_2) = mod(\frac{x_1}{x_2})$ – остача від ділення (при x_2=0 вважаємо $mod(\frac{x_1}{0}) = 0$);
3) $f(x_1, x_2) = div\left(\frac{x_1}{x_2}\right) =$ – остача від ділення (при x_2=0 вважаємо $mod(\frac{x_1}{0}) = 0$);
4) $f(x_1, x_2) = \min(x_1, x_2)$;
5) $f(x) = a^x$;
6) $f(x, y) = x^y$.
7) Показати, що якщо функція $g(x_1, \dots, x_{n-1}, x_n)$ є примітивно рекурсивною, то наступні функції також примітивно рекурсивні:
   a. $f(x_1, \dots, x_{n-1}, x_n) = \sum_{i=0}^{x_n} g(x_1, \dots, x_{n-1}, i)$;
   b. $f(x_1, \dots, x_{n-1}, x_n) = \prod_{i=0}^{x_n} g(x_1, \dots, x_{n-1}, i)$.
8) Довести, що характеристична функція $\chi(x)$ довільної скінченної множини натуральних чисел є примітивно рекурсивною функцією.

☺ Індивідуальні завдання

Довести, що подані функції є примітивно рекурсивними:

1. $f(x, y) = 3x + 2y$
2. $f(x, y) = 3xy + 2$
3. $f(x, y) = x + 5y + 1$
4. $f(x, y) = x(3y + 1)$
5. $f(x, y) = 3x(2y + 1)$
6. $f(x, y) = 3(2x + 1)^y$
7. $f(x, y) = x^y + 2y$
8. $f(x, y) = 2x^y + y$
9. $f(x, y, z) = 3xy + 2z$
10. $f(x, y, z) = xy + 2z + 3$

Тема 9: Рекурентні співвідношення

Теоретичні відомості

Рекурентні співвідношення мають надзвичайно важливе значення для програмування. Вони часто використовуються у аналізі алгоритмів.

Рекурентне співвідношення першого порядку

Нехай $\{a_k: k \geq 0\}$ деяка послідовність дійсних чисел.

Означення 1. Послідовність $\{a_k: k \geq 0\}$ називається заданою рекурентним співвідношенням першого порядку, якщо явно задано її перший член a_0, а кожен наступний член a_k цієї послідовності визначається деякою залежністю через її попередній член a_{k-1}, тобто

$$\begin{cases} a_0 = u \\ a_k = f(a_{k-1}), k \geq 1 \end{cases},$$

де *u* задане (початкове) числове значення, *f* – деяка функція.

Приклад 1

a) Рекурентне співвідношення арифметичної прогресії: $a_k = a_{k-1} + d$. Якщо задати перший член прогресії та d, то послідовність визначається однозначно.

b) Рекурентне співвідношення геометричної: $a_k = a_{k-1} \cdot q$.

c) Послідовність $a_k = k!$ задається рекурентним співвідношенням

$$\begin{cases} a_0 = 0! = 1 \\ a_k = k \cdot (k-1)! = k \cdot a_{k-1}, k \geq 1 \end{cases}.$$

d) Обчислення елементів послідовності $a_k = \frac{x^k}{k!}, k \geq 0$, на комп'ютері доцільно проводити через рекурентне співвідношення, оскільки безпосереднє обчислення степенів і факторіалів може приводити до помилкових результатів. З рівності

$$a_k = \frac{x^k}{k!} = \frac{x \cdot x^{k-1}}{k \cdot (k-1)!} = \frac{x}{k} \cdot a_{k-1}$$

отримуємо шукане рекурентне співвідношення $a_k = \frac{x}{k} \cdot a_{k-1}$.

e) У темі 3 ми використовували рекурентне співвідношення $S_k = S_{k-1} + a_k$ для обчислення суми елементів послідовності.

Лінійні однорідні рекурентні співвідношення зі сталими коефіцієнтами

Означення 2. Послідовність $\{a_k: k \geq 0\}$ називається заданою рекурентним співвідношенням *m*-го порядку, якщо

$$\begin{cases} a_0 = u, \ a_1 = v, \ldots, \ a_{m-1} = w, \\ a_k = f(a_{k-1}, a_{k-2}, \ldots, a_{k-m}), k \geq m \end{cases},$$

де *u*, *v*, *w* – задані числові сталі, *f* – деяка функція.

Якщо числова послідовність задана рекурентним співвідношенням, то не завжди відома аналітична формула для її елементів. Для класу лінійних однорідних рекурентних співвідношень зі сталими коефіцієнтами формулу для елементів можна знайти.

Означення 3. Послідовність $\{a_k: k \geq 0\}$ називається заданою лінійним однорідним рекурентним співвідношенням m-го порядку зі сталими коефіцієнтами, якщо

$$\begin{cases} a_0 = u, \ a_1 = v, \ldots, \ a_{m-1} = w, \\ a_k = c_1 \cdot a_{k-1} + c_2 \cdot a_{k-2} + \cdots + c_m \cdot a_{k-m}, k \geq m, \end{cases}$$

де u, v, w – задані числові сталі, $c_1, c_2, \ldots, c_m$ – коефіцієнти рекурентного співвідношення, які є сталими.

Існує загальний метод розв'язання лінійних однорідних рекурентних співвідношень зі сталими коефіцієнтами.

Розв'язком рекурентного співвідношення вважається формула, яка дозволяє отримати елемент послідовності за його номером, без обчислення попередніх елементів.

Розглянемо метод знаходження розв'язку рекурентного співвідношення. За співвідношенням $a_k = c_1 \cdot a_{k-1} + c_2 \cdot a_{k-2} + \cdots + c_m \cdot a_{k-m}$ складають характеристичний многочлен m-го степеня

$$x^m = c_1 \cdot x^{m-1} + c_2 \cdot x^{m-2} + \cdots + c_m$$

або

$$x^m - c_1 \cdot x^{m-1} - c_2 \cdot x^{m-2} - \cdots - c_m = 0$$

У загальному цей многочлен має m коренів.

Нехай $\lambda_1, \lambda_2, \ldots, \lambda_m$ – корені многочлена і всі корені різні. Тоді

$$a_k = b_1 \cdot \lambda_1^k + b_2 \cdot \lambda_2^k + \cdots + b_m \cdot \lambda_m^k, \ k \geq 0$$

це розв'язок рекурентного співвідношення і формула елемента послідовності. Коефіцієнти $b_1, b_2, \ldots, b_m$ визначаються із рівностей $a_0 = u, \ a_1 = v, \ldots, \ a_{m-1} = w$, які приводять до розв'язання системи із m рівнянь.

Нехай многочлен має кратні корені. Точніше, нехай $\lambda_1, \lambda_2, \ldots, \lambda_s$ корені кратностей $k_1, k_2, \ldots, k_s$ відповідно, причому $k_1 + k_2 + \cdots + k_s = m$. Тоді

$$a_k = p_1(k) \cdot \lambda_1^k + p_2(k) \cdot \lambda_2^k + \cdots + p_s(k) \cdot \lambda_s^k, \ k \geq 0$$

це розв'язок рекурентного співвідношення і формула елемента послідовності. $p_1(k), p_2(k), \ldots, p_s(k)$ – це многочлени від змінної k степенів $k_1 - 1, k_2 - 1, \ldots, k_s - 1$ відповідно. Точніше $p_i(k) = d_{i0} + d_{i1} \cdot k + d_{i2} \cdot k^2 + \cdots + d_{i\,k_i-1} \cdot k^{k_i-1}, i = 1,2, \ldots, s$. Коефіцієнти $d_{i0}, d_{i1}, \ldots, d_{i\,k_i-1}$ таких многочленів визначаються із рівностей $a_0 = u, \ a_1 = v, \ldots, \ a_{m-1} = w$.

Приклад 2

Послідовність чисел Фібоначчі задається рекурентним співвідношенням: кожне наступне число є сумою двох попередніх, тобто

$$\begin{cases} a_0 = 0, \ a_1 = 1, \\ a_k = a_{k-1} + a_{k-2}, k \geq 2' \end{cases}$$

Рівність $a_k = a_{k-1} + a_{k-2}$ – це лінійне однорідне рекурентне співвідношення другого порядку зі сталими коефіцієнтами.

Розв'яжемо співвідношення $a_k = a_{k-1} + a_{k-2}$. Складаємо характеристичний многочлен цього співвідношення. Оскільки співвідношення другого порядку, то многочлен матиме другий степінь – $x^2 = x + 1$. Цей многочлен має два різні корені

$$\lambda_1 = \frac{1+\sqrt{5}}{2}, \qquad \lambda_2 = \frac{1-\sqrt{5}}{2}.$$

На основі коренів записуємо формулу елемента послідовності

$$a_k = A \cdot \left(\frac{1+\sqrt{5}}{2}\right)^k + B \cdot \left(\frac{1-\sqrt{5}}{2}\right)^k.$$

У цій формулі невідомими є коефіцієнти *A*, *B*. Їх знайдемо, скориставшись рівностями

$$a_0 = 0, \ a_1 = 1.$$

Отримуємо систему рівнянь

$$\begin{cases} a_0 = A + B = 0 \\ a_1 = A \cdot \left(\frac{1+\sqrt{5}}{2}\right) + B \cdot \left(\frac{1-\sqrt{5}}{2}\right) = 1 \end{cases}.$$

Її розв'язки – $A = \frac{1}{\sqrt{5}}, \ B = -\frac{1}{\sqrt{5}}$. Таким чином, отримуємо знамениту формулу Біне елемента послідовності чисел Фібоначчі

$$a_k = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^k - \left(\frac{1-\sqrt{5}}{2}\right)^k}{\sqrt{5}}, \ k \geq 0.$$

Число $\phi = \frac{1+\sqrt{5}}{2} = 1{,}618$ задає відношення золотого перерізу.

Приклад 3

Розв'язати рекурентне співвідношення

$$\begin{cases} a_0 = 5, \ a_1 = 0; \\ a_k = a_{k-1} + 6a_{k-2}, k \geq 2. \end{cases}$$

Розв'язання

Характеристичний многочлен співвідношення має вигляд

$$x^2 = x + 6 \text{ або } x^2 - x - 6 = 0.$$

Його корені $\lambda_1 = 3$, $\lambda_2 = -2$ різні. На основі коренів записуємо формулу елемента послідовності

$$a_k = A \cdot 3^k + B \cdot (-2)^k.$$

Із початкових умов $a_0 = 5,\ \ a_1 = 0$ знайдемо коефіцієнти *A* і *B*.

$$a_0 = A \cdot 3^0 + B \cdot (-2)^0 = A + B = 5.$$
$$a_1 = A \cdot 3^1 + B \cdot (-2)^1 = 3A - 2B = 0.$$

Отримуємо систему рівнянь

$$\begin{cases} A + B = 5 \\ 3A - 2B = 0 \end{cases}.$$

Її розв'язок A=2 і B=3. Отже $a_k = 2 \cdot 3^k + 3 \cdot (-2)^k, k \geq 0$.

Приклад 4

Розв'язати рекурентне співвідношення

$$\begin{cases} a_1 = 2,\ \ a_2 = -4; \\ a_k = 4a_{k-1} - 4a_{k-2}, k \geq 3. \end{cases}$$

Розв'язання

Характеристичний многочлен співвідношення має вигляд

$$x^2 = 4x - 4 \text{ або } x^2 - 4x + 4 = 0.$$

Його корінь $\lambda_1 = 2$ кратності 2. Записуємо формулу елемента послідовності

$$a_k = (A + B \cdot k) \cdot 2^k .$$

$A + B \cdot k$ – це многочлен першого степеня від *k*. Із початкових умов $a_1 = 2,\ \ a_2 = -4$ знайдемо коефіцієнти *A* і *B*.

$$a_1 = (A + B \cdot 1) \cdot 2^1 = 2 \cdot (A + B) = 2 .$$
$$a_2 = (A + B \cdot 2) \cdot 2^2 = 4 \cdot (A + 2B) = -4.$$

Отримуємо систему рівнянь

$$\begin{cases} A + B = 1 \\ A + 2B = -1 \end{cases}.$$

Її розв'язок A=3 і B=-2. Отже $a_k = (3 - 2 \cdot k) \cdot 2^k , k \geq 1.$

Рекурентні співвідношення в рекурсивних алгоритмах

Оцінка ефективності рекурсивних алгоритмів часто приводить до рекурентних співвідношень, які потрібно розв'язувати.

Для прикладу розглянемо рекурсивну процедуру сортування злиттям. Ця процедура-функція *mergesort* в якості вхідних даних використовує список елементів довжини *n* і повертає цей список відсортованим. Ця функція використовує також процедуру *merge* (злиття), у якої вхідними даними є два відсортовані списки L_1 і L_2. Процедура *merge*(L_1, L_2) об'єднує списки L_1 і L_2 в один відсортований список (таке об'єднання описане в темі 10). Об'єднання двох списків довжиною *n*/2 виконується за час порядку O(n).

Псевдокод процедури сортування злиттям:

function *mergesort* (L: LIST; n:integer): LIST

{L – список типу LIST довжини n.

```
Вважається, що n є степенем числа 2}
var
        L1, L2: LIST
begin
        if n=1 then
                return (L);
        else begin
                розбиття L на дві частини L1 і L2, кожна довжиною n/2;
                return (merge(mergesort(L1, n/2), mergesort(L2, n/2)));
                end
    end
```

Позначимо через *T(n)* час виконання процедури *mergesort* в найгіршому випадку. Аналізуючи псевдокод процедури, можна записати рекурентну нерівність, яка обмежує зверху *T(n)*:

$$T(n) \leq \begin{cases} c_1, \text{ якщо } n = 1, \\ 2 \cdot T(n/2) + c_2 \cdot n, \text{ якщо } n > 1. \end{cases} \quad (1)$$

У нерівностях (1) константа c_1 відповідає фіксованій кількості кроків, що виконує алгоритм над списком L довжини 1. Якщо $n > 1$, час виконання процедури *mergesort* можна розбити на дві частини. Перша частина складається з перевірки того, що $n \neq 1$, розбиття списку L на дві частини і виклику процедури *merge*. Тому можна вибрати таку константу c_2, що час виконання цієї частини процедури *mergesort* буде обмежене величиною $c_2 \cdot n$. Друга частина процедури *mergesort* складається із двох рекурсивних викликів цієї процедури для списків довжини $n/2$, які вимагають часу $2 \cdot T(n/2)$. Звідси отримуємо другу нерівність в (1).

При розгляді рекурентних співвідношень зазвичай вводяться два спрощення.

По-перше, приймається, що час роботи алгоритму *T(n)* визначається лише для цілочислових n, оскільки в більшості алгоритмів кількість вхідних елементів виражається цілим числом.

По-друге, оскільки час роботи алгоритму з вхідними даними фіксованого розміру виражається константою, то в рекурентних співвідношеннях для достатньо малих n зазвичай справедлива тотожність $T(n) = O(1)$. Тому для зручності граничні умови рекурентних співвідношень, як правило, відкидаються та приймається, що для малих n час роботи алгоритму є *T(n)* константою.

Існують три різні підходи до розв'язання рекурентних співвідношень.

1. Знаходження такої функції *f(n)*, яка б мажорувала *T(n)* для всіх значень *n* (тобто для всіх $n \geq 1$ повинна виконуватися нерівність $T(n) \leq f(n)$).
2. У рекурентне співвідношення в праву частину послідовно підставляють вираз для *T(m)*, $m<n$, так, щоб позбутися в правій частині всіх виразів *T(m)* для $m>1$, залишивши тільки $T(1)$. Оскільки $T(1)$ завжди є константою, то в результаті

отримаємо формулу для *T(n)*, яка містить *n* та константи. Така формула називається «замкнутою формою» для *T(n)*.

3. Третій підхід полягає у використанні загальних розв'язків певних рекурентних співвідношень.

Приклад 5

Розглянемо перший підхід розв'язання рекурентних співвідношень на прикладі співвідношення (1).

Припустимо, що $T(n) = a \cdot n \cdot log_2 n$, де a – поки не визначений параметр. Доведення проводимо методом математичної індукції по *n*.

Підставимо *n*=1 в *T(n)*. $T(1) = a \cdot 1 \cdot log_2 1 = 0$, а повинно бути $T(1) \le c_1$. Попробуємо використати іншу функцію: $T(n) = a \cdot n \cdot log_2 n + b$. При *n*=1 ця функція підходить, якщо покласти $b \ge c_1$.

У відповідності з методом математичної індукції припускаємо, що для всіх $k < n$ виконується нерівність

$$T(k) \le a \cdot k \cdot log_2 k + b \qquad (2)$$

і попробуємо довести, що $T(n) \le a \cdot n \cdot log_2 n + b$.

Нехай $n \ge 2$. Із нерівностей (1) маємо $T(n) \le 2 \cdot T(n/2) + c_2 \cdot n$. Поклавши $k = n/2$, використовуємо в останній нерівності оцінку (2). Отримуємо

$$T(n) \le 2 \cdot \left(a \cdot \frac{n}{2} \cdot log_2 \frac{n}{2} + b\right) + c_2 \cdot n = 2 \cdot \left(a \cdot \frac{n}{2} \cdot (log_2 n - log_2 2) + b\right) + c_2 \cdot n =$$
$$= a \cdot n \cdot log_2 n - a \cdot n + c_2 \cdot n + 2b.$$

Якщо покласти $a = c_2 + b$, то отримаємо

$$a \cdot n \cdot log_2 n - a \cdot n + c_2 \cdot n + 2b = a \cdot n \cdot log_2 n - (c_2 + b) \cdot n + c_2 \cdot n + 2b =$$
$$= a \cdot n \cdot log_2 n + (2 - n) \cdot b.$$

Таким чином, при $a \ge c_2 + b$ отримуємо нерівність $T(n) \le a \cdot n \cdot log_2 n + b$.

Оцінка $T(n) \le a \cdot n \cdot log_2 n + b$ буде справедливою, якщо будуть виконуватися нерівності $b \ge c_1$ і $a \ge c_2 + b$. У даному випадку ці нерівності будуть виконуватися, якщо покласти $b = c_1$ і $a = c_1 + c_2$. Звідси робимо висновок, що для всіх $n \ge 1$ виконується нерівність

$$T(n) \le (c_1 + c_2) \cdot n \cdot log_2 n + c_1.$$

Іншими словами, $T(n)$ має порядок росту $O(n \cdot log_2 n)$.

У цьому прикладі показано, що функція $T(n)$ зростає не швидше ніж $O(n \cdot log_2 n)$. Однак, це не означає, що отримана точна асимптотична оцінка. Для цього потрібно детальніше проаналізувати алгоритм *mergesort* і отримати оцінку $\Omega(n \cdot log_2 n)$.

Також зауважимо, що коли ми припускаємо, що $T(n)$ має порядок $O(f(n))$, але індукцією не можемо довести нерівність $T(n) \le c \cdot f(n)$, то це ще не означає, що $T(n) \ne O(f(n))$. Можливо, потрібно добавити константу до функції $c \cdot f(n)$.

Приклад 6

Розглянемо другий підхід розв'язання рекурентних співвідношень на прикладі співвідношення (1). Він базується на принципі підстановки і на практиці часто приводить до складних сум, які іноді важко буде аналізувати.

Замінимо n на $n/2$ в співвідношенні (3)

$$T(n) \le 2 \cdot T(n/2) + c_2 \cdot n. \quad (3)$$

Отримаємо

$$T(n/2) \le 2 \cdot T(n/4) + c_2 \cdot n/2. \quad (4)$$

Підставивши (4) в початкове співвідношення (3), будемо мати

$$T(n) \le 2 \cdot (2 \cdot T(n/4) + c_2 \cdot n/2) + c_2 \cdot n = 4T(n/4) + 2c_2 \cdot n \quad (5)$$

Аналогічно, замінюючи n на $n/4$ в співвідношенні (3), отримуємо оцінку для $T(n/4)$: $T(n/4) \le 2 \cdot T(n/8) + c_2 \cdot n/4$. Підставляючи цю оцінку в (5), тримаємо

$$T(n) \le 8T(n/8) + 3c_2 \cdot n \quad (6)$$

Індукцією по i для будь-якого i можна легко отримати наступне співвідношення

$$T(n) \le 2^i \cdot T(n/2^i) + i \cdot c_2 \cdot n \quad (7)$$

Припустимо, що n є степенем числа 2, наприклад $n = 2^k$. Тоді при $i=k$ в правій частині нерівності (7) буде стояти $T(1)$:

$$T(n) \le 2^k \cdot T(1) + k \cdot c_2 \cdot n \quad (8)$$

Оскільки $n = 2^k$, то $k = log_2 n$. Так як $T(1) \le c_1$, то із (8) випливає, що

$$T(n) \le c_1 \cdot n + c_2 \cdot n \cdot log_2 n \quad (9)$$

Нерівність (9) показує верхню межу для $T(n)$, це і доводить, що $T(n)$ має порядок росту не більше $O(n \cdot log_2 n)$.

Основний метод

Основний метод стосується знаходження розв'язку рекурентних співвідношень вигляду

$$T(n) = a \cdot T(n/b) + f(n), \quad (10)$$

де $a \ge 1$ та $b > 1$ – константи, а $f(n)$ – асимптотично додатна функція.

Рекурентне співвідношення (10) описує час роботи алгоритму, в якому задача розміру n розбивається на a допоміжних підзадач, розміром n/b кожна. Отримані в результаті розбиття a підзадач розв'язуються рекурсивним методом, причому час їх розв'язку становить $T(n/b)$. Час, необхідний на розбиття задачі та об'єднання результатів цих підзадач, описується функцією $f(n)$.

Строго кажучи, при визначенні наведеного вище рекурентного співвідношення допущена неточність, оскільки число n/b може не бути цілим. Однак заміна кожного з a доданків $T(n/b)$ виразом $T(\lfloor n/b \rfloor)$ або $T(\lceil n/b \rceil)$ не впливає на асимптотичну поведінку рішення.

Основний метод ґрунтується на наступній теоремі.

Теорема. Нехай $a \geq 1$ та $b > 1$ – константи, $f(n)$ – довільна функція, а $T(n)$ – функція, визначена на множні невід'ємних цілих чисел за допомогою рекурентного співвідношення $T(n) = a \cdot T(n/b) + f(n)$, де вираз n/b інтерпретується або як $\lfloor n/b \rfloor$, або як $\lceil n/b \rceil$. Тоді асимптотичну поведінку функції $T(n)$ можна виразити наступним чином.

1. Якщо $f(n) = O(n^{\log_b a - \varepsilon})$ для деякої сталої $\varepsilon > 0$, то $T(n) = \Theta(n^{\log_b a})$.
2. Якщо $f(n) = \Theta(n^{\log_b a})$, то $T(n) = \Theta(n^{\log_b a} \cdot \log_2 n)$.
3. Якщо $f(n) = \Omega(n^{\log_b a + \varepsilon})$ для деякої сталої $\varepsilon > 0$, і якщо $a \cdot f(n/b) \leq c \cdot f(n)$ для деякої сталої $c < 1$ і всіх достатньо великих n, то $T(n) = \Theta(f(n))$.

Ідея основної теореми полягає в наступному. В кожному з трьох означених випадків функція $f(n)$ порівнюється з функцією $n^{\log_b a}$. Інтуїтивно зрозуміло, що асимптотична поведінка розв'язку рекурентного співвідношення визначається більшою з двох функцій. Якщо більшою є функція $n^{\log_b a}$ (випадок 1), то розв'язок – $T(n) = \Theta(n^{\log_b a})$. Якщо швидше зростає функція $f(n)$ (випадок 3), то розв'язок – $T(n) = \Theta(f(n))$. Якщо ж обидві функції зрівнянні (випадок 2), то відбувається множення на логарифмічний додаток і розв'язок – $T(n) = \Theta(n^{\log_b a} \cdot \log_2 n)$.

У першому випадку функція $f(n)$ повинна бути не просто менше функції $n^{\log_b a}$, а поліноміально менше. Це означає, що функція $f(n)$ повинна бути асимптотично менше функції $n^{\log_b a}$в n^{ε} разів. Аналогічно, в третьому випадку функція $f(n)$ повинна бути поліноміально більшою, а окрім того задовольняти умові регулярності: $a \cdot f(n/b) \leq c \cdot f(n)$. Ця умова виконується для більшості поліноміально обмежених функцій.

Важливо розуміти, що цими трьома випадками поведінка функції $f(n)$ не обмежується. Між випадками 1 та 2 є проміжок, в якому функція $f(n)$ менше за $n^{\log_b a}$, але не поліноміально менше. Аналогічний проміжок є між випадками 2 та 3, коли функція $f(n)$ більше $n^{\log_b a}$, але не поліноміально більше. Якщо функція $f(n)$ потрапляє в один з цих проміжків, або якщо для неї не виконуються умови регулярності випадку 3, основний метод не можна застосовувати для рекурентних співвідношень.

Приклад 7

1. Розглянемо наступне рекурентне співвідношення:

$$T(n) = 9 \cdot T(n/3) + n.$$

У цьому випадку $a = 9$, $b = 3$, а $f(n) = n$, так що $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$. Оскільки $f(n) = O(n^{\log_3 9 - \varepsilon})$, де $\varepsilon = 1$, можна застосувати випадок 1 основної теореми і зробити висновок, що $T(n) = \Theta(n^2)$.

2. Тепер розглянемо наступне співвідношення:

$$T(n) = T(2n/3) + 1,$$

в якому a = 1, b = 3/2, f(n) = 1, $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$. Тут виконується випадок 2, адже $f(n) = \Theta(n^{\log_b a}) = \Theta(1)$, тому $T(n) = \Theta(\log_2 n)$.

3. Для рекурентного співвідношення

$$T(n) = 3 \cdot T(n/4) + n \cdot log_2 n$$

виконуються умови $a = 3$, $b = 4$, $f(n) = n \cdot log_2 n$, $n^{log_b a} = n^{log_4 3} = O(n^{0{,}793})$. Оскільки $f(n) = \Omega(n^{log_4 3 + \varepsilon})$, де $\varepsilon \approx 0{,}2$, застосовується випадок 3 (якщо вдасться показати виконання умов регулярності для функції *f(n)*). При достатньо великих n умова $a \cdot f\left(\frac{n}{b}\right) = 3 \cdot \left(\frac{n}{4}\right) \cdot log_2\left(\frac{n}{4}\right) \leq \frac{3}{4} \cdot n \cdot log_2 n = c \cdot f(n)$ виконується при $c = 3/4$. Відповідно умові 3, розв'язком цього співвідношення буде $T(n) = \Theta(n \cdot log_2 n)$.

4. До рекурентного співвідношення

$$T(n) = 2 \cdot T(n/2) + n \cdot log_2 n$$

основний метод не можна застосовувати, хоч воно й має потрібний вигляд: $a = 2$, $b = 2$, $f(n) = n \cdot log_2 n$, $n^{log_b a} = n^{log_2 2} = n$. Може здатись, що до нього застосовується випадок 3, адже функція $f(n) = n \cdot log_2 n$ асимптотично більша за $n^{log_b a} = n$. Однак, проблема полягає в тому, що дана функція не поліноміально більша. Відношення $f(n)/n^{log_b a} = (n \cdot log_2 n)/n = log_2 n$ асимптотично менше функції n^{ε} для будь-якого $\varepsilon > 0$. Відповідно, дане рекурентне співвідношення знаходиться в «проміжному стані» між випадками 2 та 3.

5. Для методу сортування злиттям рекурентне співвідношення має вигляд

$$T(n) = 2 \cdot T(n/2) + n,$$

де a = 2, b = 2, f(n) = n, $n^{log_b a} = n^{log_2 2} = n$. Тут виконується випадок 2, адже $f(n) = \Theta(n^{log_b a}) = \Theta(n)$, тому $T(n) = \Theta(n \cdot log_2 n)$.

6. Для методу Штрассена добутку матриць рекурентне співвідношення має вигляд

$$T(n) = 7 \cdot T(n/2) + \Theta(n^2),$$

де a = 7, b = 2, f(n) = n^2, $n^{log_b a} = n^{log_2 7}$. Тут виконується випадок 1, оскільки $f(n) = O(n^{log_2 7 - \varepsilon})$, для $\varepsilon < 0{,}8$. Тому $T(n) = \Theta(n^{log_2 7})$.

Задачі

Розв'язати лінійні однорідні рекурентні співвідношення зі сталими коефіцієнтами.

1. а) $\begin{cases} a_0 = 11, \ a_1 = 36 \\ a_k = 3a_{k-1} + 40a_{k-2}, k \geq 2 \end{cases}$ б) $\begin{cases} a_0 = 3, \ a_1 = 72 \\ a_k = 16a_{k-1} - 64a_{k-2}, k \geq 2 \end{cases}$
2. а) $\begin{cases} a_0 = -2, \ a_1 = -9 \\ a_k = 11a_{k-1} - 30a_{k-2}, k \geq 2 \end{cases}$ б) $\begin{cases} a_0 = 4, \ a_1 = 9 \\ a_k = -18a_{k-1} - 81a_{k-2}, k \geq 2 \end{cases}$

Розв'язати рекурентні співвідношення

11. $T(n) = 8 \cdot T(n/2) + n^2$.
12. $T(n) = 2 \cdot T(n/2) + n$.
13. $T(n) = 2 \cdot T(n/2) + n^2$.

☺ Індивідуальні завдання

А

Розв’язати лінійні однорідні рекурентні співвідношення зі сталими коефіцієнтами.

3. а) $\begin{cases} a_0 = 9,\ a_1 = -7 \\ a_k = -4a_{k-1} + 5a_{k-2}, k \geq 2 \end{cases}$ б) $\begin{cases} a_0 = 5,\ a_1 = -6 \\ a_k = -12a_{k-1} - 36a_{k-2}, k \geq 2 \end{cases}$
4. а) $\begin{cases} a_0 = 9,\ a_1 = -31 \\ a_k = -6a_{k-1} + 7a_{k-2}, k \geq 2 \end{cases}$ б) $\begin{cases} a_0 = 4,\ a_1 = -24 \\ a_k = 12a_{k-1} - 36a_{k-2}, k \geq 2 \end{cases}$
5. а) $\begin{cases} a_0 = 8,\ a_1 = 29 \\ a_k = 5a_{k-1} + 14a_{k-2}, k \geq 2 \end{cases}$ б) $\begin{cases} a_0 = 5,\ a_1 = 7 \\ a_k = 14a_{k-1} - 49a_{k-2}, k \geq 2 \end{cases}$
6. а) $\begin{cases} a_0 = 11,\ a_1 = 2 \\ a_k = a_{k-1} + 12a_{k-2}, k \geq 2 \end{cases}$ б) $\begin{cases} a_0 = 6,\ a_1 = 55 \\ a_k = 10a_{k-1} - 25a_{k-2}, k \geq 2 \end{cases}$
7. а) $\begin{cases} a_0 = 1,\ a_1 = -8 \\ a_k = -7a_{k-1} - 12a_{k-2}, k \geq 2 \end{cases}$ б) $\begin{cases} a_0 = 7,\ a_1 = -16 \\ a_k = -8a_{k-1} - 16a_{k-2}, k \geq 2 \end{cases}$
8. а) $\begin{cases} a_0 = -1,\ a_1 = 2 \\ a_k = 7a_{k-1} - 12a_{k-2}, k \geq 2 \end{cases}$ б) $\begin{cases} a_0 = 5,\ a_1 = 4 \\ a_k = 8a_{k-1} - 16a_{k-2}, k \geq 2 \end{cases}$
9. а) $\begin{cases} a_0 = 6,\ a_1 = 9 \\ a_k = 3a_{k-1} + 10a_{k-2}, k \geq 2 \end{cases}$ б) $\begin{cases} a_0 = 6,\ a_1 = 12 \\ a_k = 6a_{k-1} - 9a_{k-2}, k \geq 2 \end{cases}$
10. а) $\begin{cases} a_0 = 4,\ a_1 = 3 \\ a_k = -a_{k-1} + 6a_{k-2}, k \geq 2 \end{cases}$ б) $\begin{cases} a_0 = 2,\ a_1 = -10 \\ a_k = -4a_{k-1} - 4a_{k-2}, k \geq 2 \end{cases}$
11. а) $\begin{cases} a_0 = 0,\ a_1 = -3 \\ a_k = 7a_{k-1} - 10a_{k-2}, k \geq 2 \end{cases}$ б) $\begin{cases} a_0 = 4,\ a_1 = -15 \\ a_k = -6a_{k-1} - 9a_{k-2}, k \geq 2 \end{cases}$
12. а) $\begin{cases} a_0 = 6,\ a_1 = -9 \\ a_k = -3a_{k-1} + 10a_{k-2}, k \geq 2 \end{cases}$ б) $\begin{cases} a_0 = 3,\ a_1 = -5 \\ a_k = -10a_{k-1} - 25a_{k-2}, k \geq 2 \end{cases}$

Б

Розв’язати рекурентні співвідношення

1. $T(n) = 4 \cdot T(n/2) + n.$
2. $T(n) = 5 \cdot T(n/2) + n.$
3. $T(n) = 10 \cdot T(n/3) + n.$
4. $T(n) = 3 \cdot T(n/3) + n.$
5. $T(n) = 9 \cdot T(n/3) + n^2.$
6. $T(n) = 4 \cdot T(n/2) + n^2.$
7. $T(n) = 3 \cdot T(n/2) + 1.$
8. $T(n) = 4 \cdot T(n/3) + 1.$
9. $T(n) = T(3n/4) + 1.$
10. $T(n) = 5 \cdot T(n/5) + n.$

Тема 10: Сортування

Теоретичні відомості

Сортуванням або упорядкуванням списку об'єктів називається розташування цих об'єктів за зростанням або спаданням згідно певного лінійного відношення порядку. Відомо багато методів сортування, що відрізняються швидкодією та обсягом оперативної пам'яті, що при цьому використовується. Серед цих методів можна виділити методи **внутрішнього** та **зовнішнього** сортування. При внутрішньому сортуванні всі дані, що сортуються, повністю розміщуються в оперативній пам'яті комп'ютера, де можна отримати доступ до даних у будь-якому порядку. Зовнішнє сортування застосовується тоді, коли обсяг даних, що треба відсортувати, досить великий, і їх всіх не можна розмістити в оперативній пам'яті.

Будемо розглядати методи внутрішнього сортування. Їх прийнято поділяти на дві групи: елементарні (прямі) та удосконалені.

Найвідомішими елементарними методами сортування є:

- сортування вставкою (включенням);
- сортування вибором;
- сортування обміном (бульбашкове сортування).

З удосконалених методів сортування найчастіше використовуються такі:

- швидке сортування або метод Хоара;
- сортування включенням зі спадним приростом або метод Шелла;
- сортування за допомогою дерев або пірамідальне сортування;
- сортування методом злиття.

Розглянемо постановку задачі внутрішнього сортування. Будемо вважати об'єктами сортування записи, що містять одне або декілька полів. Одне з полів, що називається **ключем**, має такий тип даних, для якого визначене відношення лінійного порядку '≤' ('≥'). Найчастіше тип ключа є числовим, символьним або рядковим. Зазвичай тип ключа може бути будь-яким, аби лише для даних цього типу можна було б визначити відношення «менше» чи «менше або рівно» («більше» чи «більше або рівно»).

Задача сортування полягає в упорядкуванні послідовності записів таким чином, щоб значення ключового поля утворювали не спадну (не зростаючу) послідовність. Іншими словами, записи $R_1, R_2, \dots, R_n$ зі значеннями ключів $k_1, k_2, \dots, k_n$ треба розташувати в такому порядку $R_{i1}, R_{i2}, \dots, R_{in}$, щоб $k_{i1} \le k_{i2} \le \dots \le k_{in}$. Не вимагається, щоб усі записи були різними. Якщо є записи з однаковими значеннями ключів, то в упорядкованій послідовності вони розташовуються поруч один з одним у будь-якому порядку.

Сортування обміном (метод «бульбашок»)

Щоб описати основну ідею методу, припустимо що треба відсортувати елементи послідовності (масиву) X_1, X_2,…, X_n за зростанням. Виконаємо перший прохід послідовності наступним чином. На кожному кроці порівнюємо j-й елемент послідовності з (j+1)-им, змінюючи j від 1 до n-1. Якщо j-й елемент більший за (j+1)-ий, то переставляємо їх місцями (виконуємо обмін), інакше не виконуємо перестановки. У результаті виконання першого проходу найбільший елемент послідовності буде знаходитися на останньому (n-му) місці. Другий прохід виконується аналогічно, але він виконується для підпослідовності елементів X_1, X_2,…, X_{n-1}, тобто без останнього елемента. Аналогічно виконуються наступні проходи. При кожному черговому проході кількість елементів підпослідовності зменшується на 1. Всього виконується n-1 проходів. Часова складність (ефективність) методу «бульбашок» дорівнює $O(n^2)$ для послідовності з n елементів. Фрагмент цього алгоритму псевдомовою має вигляд

```
for i:=n-1 downto 1 do
  for j:=1 to i do
        if X[j]>X[j+1] then ПЕРЕСТАВИТИ(X[j] , X[j+1]).
```

Приклад 1

Упорядкувати масив a із 12 елементів методом обміну за спаданням: 6, 11, 3, 15, 18, 13, 1, 0, 4, 11, 6, 3. Сортування реалізуємо у вигляді таблиці.

Індекси елементів масиву												**i**	**j**	**Коментар**
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**			
6	11	3	15	18	13	1	0	4	11	6	3			Початковий масив
6	11	3	15	18	13	1	0	4	11	6	3	11	1	a[1]<a[2], перестановка
11	6	3	15	18	13	1	0	4	11	6	3	11	2	a[2]>a[3]
11	6	3	15	18	13	1	0	4	11	6	3	11	3	a[3]<a[4], перестановка
11	6	15	3	18	13	1	0	4	11	6	3	11	4	a[4]<a[5], перестановка
11	6	15	18	3	13	1	0	4	11	6	3	11	5	a[5]<a[6], перестановка
11	6	15	18	13	3	1	0	4	11	6	3	11	6	a[6]>a[7]
11	6	15	18	13	3	1	0	4	11	6	3	11	7	a[7]>a[8]
11	6	15	18	13	3	1	0	4	11	6	3	11	8	a[8]<a[9], перестановка
11	6	15	18	13	3	1	4	0	11	6	3	11	9	a[9]<a[10], перестановка
11	6	15	18	13	3	1	4	11	0	6	3	11	10	a[10]<a[11], перестановка
11	6	15	18	13	3	1	4	11	6	0	3	11	11	a[11]<a[12], перестановка
11	6	15	18	13	3	1	4	11	6	3	0			

Перший прохід масиву при i=11 завершений. На останньому місці найменше число. Другий прохід виконуємо аналогічно, але уже над масивом із 11 елементів.

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**i**	**j**	
11	6	15	18	13	3	1	4	11	6	3	0	10	1	a[1]>a[2]
11	6	15	18	13	3	1	4	11	6	3	0	10	2	a[2]<a[3], перестановка
11	15	6	18	13	3	1	4	11	6	3	0	10	3	a[3]<a[4], перестановка
11	15	18	6	13	3	1	4	11	6	3	0	10	4	a[4]<a[5], перестановка

11	15	18	13	6	3	1	4	11	6	3	0	10	5	a[5]>a[6]
11	15	18	13	6	3	1	4	11	6	3	0	10	6	a[6]>a[7]
11	15	18	13	6	3	1	4	11	6	3	0	10	7	a[7]<a[8], перестановка
11	15	18	13	6	3	4	1	11	6	3	0	10	8	a[8]<a[9], перестановка
11	15	18	13	6	3	4	11	1	6	3	0	10	9	a[9]<a[10], перестановка
11	15	18	13	6	3	4	11	6	1	3	0	10	10	a[10]<a[11], перестановка
11	15	18	13	6	3	4	11	6	3	1	0			

Другий прохід масиву при i=10 завершений. Аналогічно виконуються інші проходи.

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**i**	
11	15	18	13	6	3	4	11	6	3	1	0		
15	18	13	11	6	4	11	6	3	3	1	0	9	3-й прохід
18	15	13	11	6	11	6	4	3	3	1	0	8	4-й прохід
18	15	13	11	11	6	6	4	3	3	1	0	7	5-й прохід
18	15	13	11	11	6	6	4	3	3	1	0	6	6-й прохід
18	15	13	11	11	6	6	4	3	3	1	0	5	7-й прохід
18	15	13	11	11	6	6	4	3	3	1	0	4	8-й прохід
18	15	13	11	11	6	6	4	3	3	1	0	3	9-й прохід
18	15	13	11	11	6	6	4	3	3	1	0	2	10-й прохід
18	15	13	11	11	6	6	4	3	3	1	0	1	11-й прохід

Сортування за спаданням завершено

Сортування вставками

Нехай знову треба відсортувати елементи послідовності (масиву) $X_1, X_2,\ldots, X_n$ за зростанням. Суть методу вставок полягає в тому, що на i-му етапі ми «вставляємо» i-й елемент X_i у потрібну позицію серед елементів $X_1, X_2,\ldots, X_{i-1}$, які вже впорядковані. Пошук відповідної позиції здійснюється так, щоб послідовність елементів $X_1, X_2,\ldots, X_{i-1}, X_i$ після вставки стала впорядкованою. Доцільно ввести додатковий елемент X_0, значення ключа якого буде меншим за значення ключа будь-якого елемента $X_1, X_2,\ldots, X_n$. Часова складність методу вставок дорівнює $O(n^2)$ для послідовності з n елементів. Фрагмент цього алгоритму псевдомовою має вигляд

```
X[0]:= -∞;
for i:=2 to n do
begin
    j:=i:
    while X[j-1]>X[j] do
    begin
        ПЕРЕСТАВИТИ(X[j-1] , X[j]);
        j:=j-1
    end
end.
```

Приклад 2

Упорядкувати масив a із 12 елементів вставками за зростанням: 6, 11, 3, 15, 18, 13, 1, 0, 4, 11, 6, 3. Добавляємо до масиву елемент a[0] = -1, який менший елементів масиву. Сортування реалізуємо у вигляді таблиці.

Індекси елементів масиву													**i**	**j**	**Коментар**
0	1	2	3	4	5	6	7	8	9	10	11	12			
-1	6	11	3	15	18	13	1	0	4	11	6	3	Початковий масив. Підмасив з елемента a[1] вважається упорядкованим за зростанням		
вставка a[2]															
-1	6	11	3	15	18	13	1	0	4	11	6	3	2	2	a[1]<a[2], вставка a[2] завершена
вставка a[3]															
-1	6	11	3	15	18	13	1	0	4	11	6	3	3	3	a[2]>a[3], перестановка
-1	6	3	11	15	18	13	1	0	4	11	6	3		2	a[1]>a[2], перестановка
-1	3	6	11	15	18	13	1	0	4	11	6	3		1	a[0]<a[1], вставка a[3] завершена
вставка a[4]															
-1	3	6	11	15	18	13	1	0	4	11	6	3	4	4	a[2]<a[4], вставка a[4] завершена
вставка a[5]															
-1	3	6	11	15	18	13	1	0	4	11	6	3	5	5	a[4]<a[5], вставка a[5] завершена
вставка a[6]															
-1	3	6	11	15	18	13	1	0	4	11	6	3	6	6	a[5]>a[6], перестановка
-1	3	6	11	15	13	18	1	0	4	11	6	3		5	a[4]>a[5], перестановка
-1	3	6	11	13	15	18	1	0	4	11	6	3		4	a[3]<a[4], вставка a[6] завершена
вставка a[7]															
-1	3	6	11	13	15	18	1	0	4	11	6	3	7	7	a[6]>a[7], перестановка
-1	3	6	11	13	15	1	18	0	4	11	6	3		6	a[5]>a[6], перестановка
-1	3	6	11	13	1	15	18	0	4	11	6	3		5	a[4]>a[5], перестановка
-1	3	6	11	1	13	15	18	0	4	11	6	3		4	a[3]>a[4], перестановка
-1	3	6	1	11	13	15	18	0	4	11	6	3		3	a[2]>a[3], перестановка
-1	3	1	6	11	13	15	18	0	4	11	6	3		2	a[1]>a[2], перестановка
-1	1	3	6	11	13	15	18	0	4	11	6	3		1	a[0]<a[1], вставка a[7] завершена
вставка a[8]															
-1	1	3	6	11	13	15	18	0	4	11	6	3	8	8	a[7]>a[8], перестановка
-1	1	3	6	11	13	15	0	18	4	11	6	3		7	a[6]>a[7], перестановка
-1	1	3	6	11	13	0	15	18	4	11	6	3		6	a[5]>a[6], перестановка
-1	1	3	6	11	0	13	15	18	4	11	6	3		5	a[4]>a[5], перестановка
-1	1	3	6	0	11	13	15	18	4	11	6	3		4	a[3]>a[4], перестановка
-1	1	3	0	6	11	13	15	18	4	11	6	3		3	a[2]>a[3], перестановка
-1	1	0	3	6	11	13	15	18	4	11	6	3		2	a[1]>a[2], перестановка

-1	0	1	3	6	11	13	15	18	4	11	6	3		1	a[0]<a[1], вставка a[8] завершена
вставка a[9]															
-1	0	1	3	6	11	13	15	18	4	11	6	3	9	9	a[8]>a[9], перестановка
-1	0	1	3	6	11	13	15	4	18	11	6	3		8	a[7]>a[8], перестановка
-1	0	1	3	6	11	13	4	15	18	11	6	3		7	a[6]>a[7], перестановка
-1	0	1	3	6	11	4	13	15	18	11	6	3		6	a[5]>a[6], перестановка
-1	0	1	3	6	4	11	13	15	18	11	6	3		5	a[4]>a[5], перестановка
-1	0	1	3	4	6	11	13	15	18	11	6	3		4	a[3]<a[4], вставка a[9] завершена
вставка a[10]															
-1	0	1	3	4	6	11	13	15	18	11	6	3	10	10	a[9]>a[10],перестановка
-1	0	1	3	4	6	11	13	15	11	18	6	3		9	a[8]>a[9], перестановка
-1	0	1	3	4	6	11	13	11	15	18	6	3		8	a[7]>a[8], перестановка
-1	0	1	3	4	6	11	11	13	15	18	6	3		7	a[6]<=a[7], вставка a[10] завершена
вставка a[11]															
-1	0	1	3	4	6	11	11	13	15	18	6	3	11	11	a[10]>a[11], перестановка
-1	0	1	3	4	6	11	11	13	15	6	18	3		10	a[9]>a[10],перестановка
-1	0	1	3	4	6	11	11	13	6	15	18	3		9	a[8>a[9], перестановка
-1	0	1	3	4	6	11	11	6	13	15	18	3		8	a[7]>a[8], перестановка
-1	0	1	3	4	6	11	6	11	13	15	18	3		7	a[6]>a[7], перестановка
-1	0	1	3	4	6	6	11	11	13	15	18	3		6	a[5]<=a[6], вставка a[11] завершена
вставка a[12]															
-1	0	1	3	4	6	6	11	11	13	15	18	3	12	12	a[11]>a[12], перестановка
-1	0	1	3	4	6	6	11	11	13	15	3	18		11	a[10]>a[11], перестановка
-1	0	1	3	4	6	6	11	11	13	3	15	18		10	a[9]>a[10],перестановка
-1	0	1	3	4	6	6	11	11	3	13	15	18		9	a[8]>a[9], перестановка
-1	0	1	3	4	6	6	11	3	11	13	15	18		8	a[7]>a[8], перестановка
-1	0	1	3	4	6	6	3	11	11	13	15	18		7	a[6]>a[7], перестановка
-1	0	1	3	4	6	3	6	11	11	13	15	18		6	a[5]>a[6], перестановка
-1	0	1	3	4	3	6	6	11	11	13	15	18		5	a[4]>a[5], перестановка
-1	0	1	3	3	4	6	6	11	11	13	15	18		4	a[3]<=a[4], вставка a[12] завершена
	0	1	3	3	4	6	6	11	11	13	15	18	Відсортований масив		

Сортування вибором

Ідея цього методу полягає в тому, що на і-му етапі сортування вибирається найменший елемент серед елементів $X_i,\ldots,X_n$, який переставляється місцями з елементом X_i. Після і-го етапу всі елементи $X_1,\ldots, X_i$ будуть упорядковані за зростанням. Часова складність методу вставок дорівнює $O(n^2)$ для послідовності з n елементів. Фрагмент цього алгоритму псевдомовою має вигляд

```
for i:=1 to n-1 do
   begin
         index:=i;
         for j:=i+1 to n do
               if X[j]<X[index] then index:=j;
   ПЕРЕСТАВИТИ(X[i], X[index])
end.
```

Приклад 3

Упорядкувати масив a із 10 елементів вибором за зростанням: 6, 11, 3, 15, 18, 13, 1, 0, 4, 11. Змінна index використовується для запам'ятовування індексу найменшого елемента. Сортування реалізуємо у вигляді таблиці.

Індекси елементів масиву										i	j	index	Коментар
1	2	3	4	5	6	7	8	9	10				
6	11	3	15	18	13	1	0	4	11				Початковий масив. Серед елементів від a[2] до a[10] шукаємо найменший.
6	11	3	15	18	13	1	0	4	11	1	2	1	a[1]<a[2], index не змінився
6	11	3	15	18	13	1	0	4	11		3	1	a[1]>a[3], index змінився
6	11	3	15	18	13	1	0	4	11		4	3	a[3]<a[4], index не змінився
6	11	3	15	18	13	1	0	4	11		5	3	a[3]<a[5], index не змінився
6	11	3	15	18	13	1	0	4	11		6	3	a[3]<a[6], index не змінився
6	11	3	15	18	13	1	0	4	11		7	7	a[3]>a[7], index змінився
6	11	3	15	18	13	1	0	4	11		8	8	a[7]>a[8], index змінився
6	11	3	15	18	13	1	0	4	11		9	8	a[8]<a[9], index не змінився
6	11	3	15	18	13	1	0	4	11		10	8	a[8]<a[10], index не змінився
0	11	3	15	18	13	1	6	4	11	1		8	перестановка найменшого елемента a[8] з a[1]
0	11	3	15	18	13	1	6	4	11	2	3	2	a[2]>a[3], index змінився
0	11	3	15	18	13	1	6	4	11		4	3	a[3]<a[4], index не змінився
0	11	3	15	18	13	1	6	4	11		5	3	a[3]<a[5], index не змінився
0	11	3	15	18	13	1	6	4	11		6	3	a[3]<a[6], index не змінився
0	11	3	15	18	13	1	6	4	11		7	7	a[3]>a[7], index змінився
0	11	3	15	18	13	1	6	4	11		8	7	a[7]<a[8], index не змінився
0	11	3	15	18	13	1	6	4	11		9	7	a[7]<a[9], index не змінився
0	11	3	15	18	13	1	6	4	11		10	7	a[7]<a[10], index не змінився
0	1	3	15	18	13	11	6	4	11	2		7	перестановка найменшого елемента a[7] з a[2]
0	1	3	15	18	13	11	6	4	11	3	4	3	a[3]<a[4], index не змінився
0	1	3	15	18	13	11	6	4	11		5	3	a[3]<a[5], index не змінився
0	1	3	15	18	13	11	6	4	11		6	3	a[3]<a[6], index не змінився
0	1	3	15	18	13	11	6	4	11		7	3	a[3]<a[7], index не змінився
0	1	3	15	18	13	11	6	4	11		8	3	a[3]<a[8], index не змінився
0	1	3	15	18	13	11	6	4	11		9	3	a[3]<a[9], index не змінився
0	1	3	15	18	13	11	6	4	11		10	3	a[3]<a[10], index не змінився

0	1	3	15	18	13	11	6	4	11	3		3	перестановка найменшого елемента a[3] з a[3]
0	1	3	15	18	13	11	6	4	11	4	5	4	a[4]<a[5], index не змінився
0	1	3	15	18	13	11	6	4	11		6	6	a[4]>a[6], index змінився
0	1	3	15	18	13	11	6	4	11		7	7	a[6]>a[7], index змінився
0	1	3	15	18	13	11	6	4	11		8	8	a[7]>a[8], index змінився
0	1	3	15	18	13	11	6	4	11		9	9	a[8]>a[9], index змінився
0	1	3	15	18	13	11	6	4	11		10	9	a[9]<a[10], index не змінився
0	1	3	4	18	13	11	6	15	11	4		9	перестановка найменшого елемента a[9] з a[4]
0	1	3	4	18	13	11	6	15	11	5	6	5	a[5]>a[6], index змінився
0	1	3	4	18	13	11	6	15	11		7	6	a[6]>a[7], index змінився
0	1	3	4	18	13	11	6	15	11		8	7	a[7]>a[8], index змінився
0	1	3	4	18	13	11	6	15	11		9	8	a[8]<a[9], index не змінився
0	1	3	4	18	13	11	6	15	11		10	8	a[8]<a[10], index не змінився
0	1	3	4	6	13	11	18	15	11	5		8	перестановка найменшого елемента a[8] з a[5]
0	1	3	4	6	13	11	18	15	11	6	7	6	a[6]>a[7], index змінився
0	1	3	4	6	13	11	18	15	11		8	7	a[7]<a[8], index не змінився
0	1	3	4	6	13	11	18	15	11		9	7	a[7]<a[9], index не змінився
0	1	3	4	6	13	11	18	15	11		10	7	a[7]<=a[10], index не змінився
0	1	3	4	6	13	11	18	15	11	6		7	перестановка найменшого елемента a[7] з a[6]
0	1	3	4	6	11	13	18	15	11	7	8	7	a[7]<a[8], index не змінився
0	1	3	4	6	11	13	18	15	11		9	7	a[7]<a[9], index не змінився
0	1	3	4	6	11	13	18	15	11		10	10	a[7]>a[10], index змінився
0	1	3	4	6	11	11	18	15	13	7		10	перестановка найменшого елемента a[10] з a[7]
0	1	3	4	6	11	11	18	15	13	8	9	8	a[8]>a[9], index змінився
0	1	3	4	6	11	11	18	15	13		10	9	a[9]>a[10], index змінився
0	1	3	4	6	11	11	13	15	18	8		10	перестановка найменшого елемента a[10] з a[8]
0	1	3	4	6	11	11	13	15	18	9	10	9	a[9]<a[10], index не змінився
0	1	3	4	6	11	11	13	15	18	9		9	перестановка найменшого елемента a[9] з a[9]
0	1	3	4	6	11	11	13	15	18	Відсортований масив			

Швидке сортування.

Алгоритм швидкого сортування базується на методі «розділяй та володарюй» побудови алгоритмів. Основна ідея алгоритму полягає в тому, щоб на кожному кроці поділити елементи послідовності на дві частини, а потім кожну з частин сортувати окремо.

Розглянемо алгоритм швидкого сортування масиву X[1], X[2],…, X[n] за зростанням. З невеликими поправками його можна застосувати також до сортування файлу. Основними діями в алгоритмі швидкого сортування є:

1) вибір опорного елемента;

2) розбиття масиву або його частини на дві половини.

На початку роботи алгоритму в масиві вибирається деякий *опорний елемент* (ключ) V, відносно якого весь масив треба розділити на дві половини. Часто в якості опорного елемента V вибирається середній елемент масиву. Далі елементи масиву переставляються так, щоб для деякого індексу j усі переставлені елементи X[1],…, X[j] були меншими за V, а всі елементи X[j+1],…, X[n] були більшими або рівними V. Отримуємо дві половини X[1],…, X[j] та X[j+1],…, X[n] початкового масиву. Тепер застосовуємо описані дії до кожної з частин X[1],…, X[j] та X[j+1],…, X[n] окремо і кожну з них теж розбиваємо на дві менші частини. Далі до отриманих менших частин знову застосовуємо описані дії і т. д. На кожному кроці роботи алгоритму розмір частин масиву зменшується. Врешті решт розмір усіх частин масиву не перевищуватиме 1, і алгоритм припинить роботу. Отримаємо упорядкований за зростанням масив. Алгоритм швидкого сортування є рекурсивним.

Нехай маємо якусь частину X[i],…, X[j] масиву. Для неї спочатку детально опишемо вибір опорного елемента V. У якості V виберемо більший із двох елементів масиву X[i],…, X[j], якщо він є. Пошук V починаємо з початку частини масиву. Функція, що повертає індекс опорного елемента V, має вигляд

```
function Find(i, j : integer) : integer;
var k: integer;
begin
        for k:=i+1 to j do
                if X[k]>X[i] then
                                begin Find:=k; exit end
                        else if X[k]<X[i] then
                                begin Find:=i; exit end;
        Find:=0; { різних елементів немає }
end;
```

Тепер опишемо такі перестановки елементів масиву X[i],…, X[j], щоб зліва у масиві знаходилися елементи менші за опорний елемент V, а справа – більші або рівні. Нехай V=X[k], i ≤ k ≤j, причому індекс k буде знайдений за допомогою функції Find. Щоб виконати необхідні перестановки, введемо два вказівники L та R. Ці вказівники будуть пробігати по індексах масиву X[i],…, X[j]. Якщо L та R вибрані (зафіксовані), то вони будуть вказувати відповідно на лівий і правий кінці тієї частини масиву X, де в даний час ми переставляємо елементи. При цьому вважаємо, що вже всі елементи X[i],…, X[L-1], які розташовані зліва від X[L], мають значення менші за V. Відповідно елементи X[R+1],…, X[j], які розташовані справа від X[R], мають значення більші або рівні V. Нам необхідно переставляти елементи X[L], X[R].

Спочатку покладемо L=i та R=j. Далі будемо повторювати наступні дії, що переміщують вказівник L вправо, а вказівник R вліво до тих пір, поки вказівники не зустрінуться.

1. Вказівник (індекс елемента масиву) L зміщується вправо, поки не знайдеться елемент X[L], який не менший за опорний елемент V. Вказівник R зміщується вліво, поки не знайдеться елемент X[R], який менший за опорний елемент V.
2. Виконується перевірка: якщо L>R (на практиці можлива тільки ситуація, коли L=R+1), то перестановки елементів масиву X[i],…, X[j] закінчуються.
3. Якщо L<R (випадок L=R неможливий), то переставляємо місцями елементи X[L] та X[R]. Вказівник L зміщуємо вправо на одну позицію від попереднього положення, а вказівник R- на одну позицію вліво. Далі процес продовжується з пункту 1.

Наступна функція виконує описану перестановку елементів масиву X[i],…, X[j] та повертає індекс L, що вказує точку поділу цього масиву на дві частини відносно заданого опорного елемента V.

```
Function Перестановка Масиву(i, j: integer; V: < тип елементів масиву >) : integer;
var L, R : integer;
begin
  L:=i;   R:=j;
  repeat
        ПЕРЕСТАВИТИ(X[L], X[R]);
        while X[L]<V do L:=L+1;
        while X[R]>=V do R:=R-1;
  until L>R;
  Перестановка Масиву:=L;
end;
```

Тепер наведемо ескіз рекурсивної процедури швидкого сортування.

```
procedure QuickSort( i, j : integer);
var V: < тип елементів масиву >; ІндексОпорногоЕлемента, k : integer;
begin
  ІндексОпорногоЕлемента:= Find(i, j);
  if ІндексОпорногоЕлемента <>0 then {якщо всі елементи рівні,
                                                  то нічого не робити}
        begin
              V:=X[ІндексОпорногоЕлемента];
              k:= ПерестановкаМасиву(i, j, V);
              QuickSort( i, k-1);
              QuickSort( k, j);
        end;
end;
```

Для сортування елементів усього масиву X[1], X[2],…, X[n] треба просто викликати процедуру QuickSort(1, n).

Часова складність алгоритму швидкого сортування для послідовності з n елементів дорівнює $O(n^2)$ в гіршому випадку. Але в середньому його ефективність дорівнює $O(n*\log_2 n)$. Тому з практичної точки зору він є дуже ефективним і найчастіше використовується при сортуванні.

Приклад 4

Упорядкувати масив a із 10 елементів за зростанням методом швидкого сортування: 6, 11, 3, 15, 18, 13, 1, 0, 4, 11. Сортування реалізуємо у вигляді таблиці.

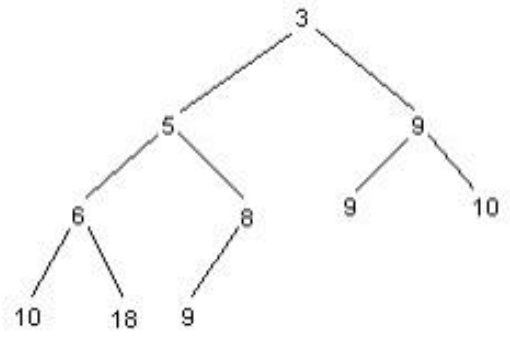
Індекси елементів масиву										V	L	R	Коментар
1	2	3	4	5	6	7	8	9	10				
6	11	3	15	18	13	1	0	4	11	Початковий масив			
6	11	3	15	18	13	1	0	4	11	11	1	10	перестановка X[L], X[R]
11	11	3	15	18	13	1	0	4	6				
11	11	3	15	18	13	1	0	4	6	11	1		while X[L]<V do L:=L+1;
11	11	3	15	18	13	1	0	4	6	11		10	while X[R]>=V do R:=R-1;
6	11	3	15	18	13	1	0	4	11	11	1	10	перестановка X[L], X[R]
6	11	3	15	18	13	1	0	4	11	11	2		while X[L]<V do L:=L+1;
6	11	3	15	18	13	1	0	4	11	11	2	9	while X[R]>=V do R:=R-1;
6	4	3	15	18	13	1	0	11	11	11	2	9	перестановка X[L], X[R]
6	4	3	15	18	13	1	0	11	11	11	4		while X[L]<V do L:=L+1;
6	4	3	15	18	13	1	0	11	11	11	4	8	while X[R]>=V do R:=R-1;
6	4	3	0	18	13	1	15	11	11	11	4	8	перестановка X[L], X[R]
6	4	3	0	18	13	1	15	11	11	11	5		while X[L]<V do L:=L+1;
6	4	3	0	18	13	1	15	11	11	11	5	7	while X[R]>=V do R:=R-1;
6	4	3	0	1	13	18	15	11	11	11	5	7	перестановка X[L], X[R]
6	4	3	0	1	13	18	15	11	11	11	6		while X[L]<V do L:=L+1;
6	4	3	0	1	13	18	15	11	11	11	6	5	L>R, k=L=6
6	4	3	0	1	13	18	15	11	11	Розбили масив на дві частини. Рівень 1			
6	4	3	0	1						Працюємо з лівою частиною			
6	4	3	0	1						6	1	5	перестановка X[L], X[R]
1	4	3	0	6						6	5		while X[L]<V do L:=L+1;
1	4	3	0	6						6	5	4	while X[R]>=V do R:=R-1;
1	4	3	0	6						6	5	4	L>R, k=L=5
1	4	3	0	6						Розбили масив на дві частини. Рівень 2			
1	4	3	0							Працюємо з лівою частиною			
1	4	3	0							4	1	4	перестановка X[L], X[R]
0	4	3	1							4	2		while X[L]<V do L:=L+1;
0	4	3	1							4	2	4	while X[R]>=V do R:=R-1;
0	1	3	4							4	2	4	перестановка X[L], X[R]
0	1	3	4							4	4		while X[L]<V do L:=L+1;
0	1	3	4							4	4	3	while X[R]>=V do R:=R-1;
0	1	3	4							4	4	3	L>R, k=L=4

0	1	3	4							Розбили масив на дві частини. Рівень 3			
0	1	3								Працюємо з лівою частиною			
3	1	0								1	1	3	перестановка X[L], X[R]
3	1	0								1	1		while X[L]<V do L:=L+1;
3	1	0								1	1	3	while X[R]>=V do R:=R-1;
0	1	3								1	1	3	перестановка X[L], X[R]
0	1	3								1	2		while X[L]<V do L:=L+1;
0	1	3								1	2	1	while X[R]>=V do R:=R-1;
0	1	3								1	2	1	L>R, k=L=2
0	1	3								Розбили масив на дві частини. Рівень 4			
0										Працюємо з лівою частиною. Ця частина з одного елемента. Обрив рекурсії			
	1	3								Працюємо з правою частиною рівня 4			
	3	1								3	2	3	перестановка X[L], X[R]
	3	1								3	2		while X[L]<V do L:=L+1;
	3	1								3	2	3	while X[R]>=V do R:=R-1;
	1	3								3			перестановка X[L], X[R]
	1	3								3	3		while X[L]<V do L:=L+1;
	1	3								3	3	2	while X[R]>=V do R:=R-1;
	1	3								3	3	2	L>R, k=L=3
	1	3								Розбили масив на дві частини. Рівень 5			
	1									Працюємо з лівою частиною. Ця частина з одного елемента. Обрив рекурсії			
		3								Працюємо з правою частиною рівня 5. Ця частина з одного елемента. Обрив рекурсії			
			4							Працюємо з правою частиною рівня 3. Ця частина з одного елемента. Обрив рекурсії			
				6						Працюємо з правою частиною рівня 2. Ця частина з одного елемента. Обрив рекурсії			
0	1	3	4	6						Відсортована ліва частина рівня 1			
					13	18	15	11	11	Працюємо з правою частиною рівня 1			
					11	18	15	11	13	18	6	10	перестановка X[L], X[R]
					11	18	15	11	13	18	7		while X[L]<V do L:=L+1;
					11	18	15	11	13	18	7	10	while X[R]>=V do R:=R-1;
					11	13	15	11	18	18	7	10	перестановка X[L], X[R]
					11	13	15	11	18	18	10		while X[L]<V do L:=L+1;
					11	13	15	11	18	18	10	9	while X[R]>=V do R:=R-1;
					11	13	15	11	18	18	10	9	L>R, k=L=10
					11	13	15	11	18	Розбили масив на дві частини. Рівень 2			
					11	13	15	11		Працюємо з лівою частиною			
					11	13	15	11		13	6	9	перестановка X[L], X[R]
					11	13	15	11		13	7		while X[L]<V do L:=L+1;
					11	13	15	11		13	7	9	while X[R]>=V do R:=R-1;
					11	11	15	13		13	7	9	перестановка X[L], X[R]
					11	11	15	13		13	8		while X[L]<V do L:=L+1;
					11	11	15	13		13	8	7	while X[R]>=V do R:=R-1;
					11	11	15	13		13	8	7	L>R, k=L=8

					11	11	15	13		Розбили масив на дві частини. Рівень 3				
					11	11				Працюємо з лівою частиною. Ця частина містить однакові елементи. Обрив рекурсії				
							15	13		Працюємо з правою частиною рівня 3				
							13	15		15	8	9	перестановка X[L], X[R]	
							13	15		15	9		while X[L]<V do L:=L+1;	
							13	15		15	9	8	while X[R]>=V do R:=R-1;	
							13	15		15	9	8	L>R, k=L=9	
							13	15		Розбили масив на дві частини. Рівень 4				
							13			Працюємо з лівою частиною. Ця частина з одного елемента. Обрив рекурсії				
								15		Працюємо з правою частиною рівня 4. Ця частина з одного елемента. Обрив рекурсії				
									18	Працюємо з правою частиною рівня 2. Ця частина з одного елемента. Обрив рекурсії				
					11	11	13	15	18	Відсортована права частина рівня 1				
0	1	3	4	6	11	11	13	15	18	Весь відсортований масив				

Пірамідальне сортування

Пірамідою називається збалансоване за висотою бінарне дерево висоти h, у якому значення будь-якого вузла не більше (не менше) за значення його синів, і всі листки рівня h максимально зміщені вліво. Прикладом піраміди є зображене дерево. Висота цього дерева дорівнює 3, на третьому рівні розташовані листки 10, 18, 9, які максимально зміщені вліво, а попередні рівні дерева заповнені повністю. Таке дерево, тобто піраміду, можна повністю представити одновимірним масивом X[1], X[2],…, X[n]. У такому масиві елементи X[2*i], X[2*i+1] є відповідно лівим та правим синами елемента X[i]. Для зображеного дерева масив його вузлів буде такий



3, 5, 9, 6, 8, 9, 10, 10, 18, 9.

У піраміді найменший (найбільший) елемент розташований на вершині (у корені дерева), а у відповідному масиві він є першим. На зображеній піраміді значення кожного вузла-батька не більше за значення синів. Якщо будувати піраміду, користуючись відношенням «не менше», то на вершині піраміди буде розташований найбільший елемент.

Розглянемо суть алгоритму пірамідального сортування. Нехай задано масив X[1], X[2],…, X[n] і його треба відсортувати за спаданням. Для цього будемо використовувати піраміду з найменшим елементом на вершині, тобто піраміда буде будуватися з використанням відношення «не більше». Алгоритм пірамідального сортування складається з двох фаз: фаза побудови та фаза вибору.

Фаза побудови полягає в тому, щоб з масиву X[1], X[2],…, X[n] побудувати піраміду. Припустимо, що для цього масиву ми побудували збалансоване за висотою бінарне дерево, у якого всі листки найнижчого рівня максимально зміщені вліво (аналогічно зображеному дереву). Коренем цього дерева є X[1], вузлами першого рівня є елементи X[2], X[3], вузлами другого рівня є елементи X[4], X[5], X[6], X[7] і т. д. Побудоване дерево не обов'язково є пірамідою, бо ми не знаємо, чи кожен вузол-батько є не більшим за своїх синів. Але друга половина масиву, а саме, X[n div 2 +1], X[n div 2 +2],…, X[n] у цьому дереві утворює піраміду, бо кожний із цих елементів не має синів. Використовуючи останній факт, перебудуємо дерево так, щоб отримати піраміду.

Отже, послідовність X[n div 2 +1], X[n div 2 +2],…, X[n] є пірамідою. Добавимо у її початок елемент X[n div 2] і отримаємо послідовність

X[n div 2], X[n div 2 +1], X[n div 2 +2],…, X[n],

яку перебудуємо у піраміду. Потім добавляємо наступний попередній елемент і послідовність

X[n div 2-1], X[n div 2], X[n div 2 +1], X[n div 2 +2],…, X[n]

знову перебудуємо у піраміду. Продовжуючи цей процес далі, із масиву X[1], X[2],…, X[n] побудуємо піраміду. Таким чином, на кожному кроці фази побудови піраміди до послідовності X[i+1],…, X[n], яка є пірамідою, добавляємо попередній елемент масиву X[i] та утворюємо послідовність X[i], X[i+1],…, X[n], яку перебудовуємо у піраміду.

Опишемо цей крок перебудови детальніше. Після добавлення елемента X[i] до піраміди X[i+1],…, X[n] утворюється послідовність X[i], X[i+1],…, X[n], яка вже не обов'язково є пірамідою. Порушив піраміду елемент X[i], тому його треба «проштовхнути» вниз по дереву, щоб утворилася піраміда. Для цього порівнюємо X[i] з меншим із його синів X[2*i], X[2*i+1]. Якщо X[i] не більший за синів, то нічого робити не треба, бо послідовність X[i], X[i+1],…, X[n] вже є пірамідою. Якщо X[i] більший за якогось із синів, то переставляємо X[i] із меншим із синів. Елемент X[i] опинився на новому місці, і для нього можуть існувати наступні сини. Якщо X[i] знову порушує піраміду, то його ще раз треба переставити з меншим із синів. Елемент X[i] потрібно переставляти («проштовхувати») до тих пір, поки не отримаємо піраміду. Процедура проштовхування має вигляд:

```
procedure ПРОШТОВХНУТИ( i, j: integer);
    {Елементи X[i+1],…, X[j] утворюють піраміду, а елемент X[i] добавляється.
    Процедура перебудовує послідовність X[i], X[i+1],…,X[j] у піраміду}
var r: integer;          {вказує поточну позицію елемента X[i]}
begin
    r:=i;   {початкова ініціалізація}
    while r <= j div 2 do
        if j = 2*r then
            begin {елемент у позиції r має одного сина в позиції 2*r}
                if X[r]>X[2*r] then ПЕРЕСТАВИТИ(X[r], X[2*r]);
```

```
                r:=j {передчасний вихід із циклу while}
            end
            else {елемент у позиції r має двох синів у позиціях 2*r та 2*r+1}
                if X[r]>X[2*r] and X[2*r]<=X[2*r+1] then
                    begin {перестановка елемента в позиції r з лівим сином}
                        ПЕРЕСТАВИТИ(X[r], X[2*r]);
                        r:= 2*r
                    end
                else if X[r]>X[2*r+1] and X[2*r+1]<X[2*r] then
                begin {перестановка елемента в позиції r з правим сином}
                    ПЕРЕСТАВИТИ(X[r], X[2*r+1]);
                    r:= 2*r+1
                end
            else {елемент в позиції r не порушує піраміду}
                r:= j {вихід із циклу while}
    end;
```

Із використанням цієї процедури фаза побудови піраміди з масиву X[1], X[2],…, X[n] має вигляд **for** i:= n div 2 **downto** 1 **do** ПРОШТОВХНУТИ(i, n).

Розглянемо тепер другу фазу алгоритму – фазу вибору. У результаті виконання першої фази отримали піраміду X[1], X[2],…, X[n]. Переставляємо елементи X[1], X[n] місцями. Таким чином, на останньому місці буде знаходитися найменший елемент. Тепер розглядаємо послідовність X[1], X[2],…, X[n-1] без останнього елемента. Якщо вона не є пірамідою, то порушує піраміду елемент X[1]. Застосувавши до послідовності X[1], X[2],…,X[n-1] процедуру ПРОШТОВХНУТИ, перетворюємо цю послідовність у піраміду. У піраміді X[1], X[2],…, X[n-1] знову переставляємо елементи X[1], X[n-1] місцями. Продовжуючи цей процес далі та зменшуючи на кожному кроці послідовність, яку треба перебудовувати у піраміду, отримаємо відсортований початковий масив за спаданням. Весь алгоритм пірамідального сортування масиву X[1], X[2],…, X[n] має вигляд:

```
    {Фаза побудови}
for i:= n div 2 downto 1 do ПРОШТОВХНУТИ(i, n).
    {Фаза вибору}
for i:= n downto 2 do
begin
    ПЕРЕСТАВИТИ(X[1], X[i]);
    ПРОШТОВХНУТИ(1, n-1).
end
```

Часова складність алгоритму пірамідального сортування для послідовності з n елементів дорівнює $O(n*\log_2 n)$ як у гіршому випадку так і в середньому. Хоч він є дуже ефективним, але на практиці віддають перевагу швидкому сортуванню.

Приклад 5

Упорядкувати масив a із 10 елементів за зростанням пірамідальним сортуванням: 6, 11, 3, 15, 18, 13, 1, 0, 4, 11. Сортування реалізуємо у вигляді таблиці. На вершині піраміди повинен бути найбільший елемент, тобто батько повинен бути не меншим за своїх синів.

Фаза побудови піраміди – Б – індекс батька, ЛС – індекс лівого сина, ПС – індекс правого сина

Індекси елементів масиву										**i**	**Б**	**ЛС**	**ПС**	**Коментар**
1	2	3	4	5	6	7	8	9	10					
6	11	3	15	18	13	1	0	4	11	Початковий масив. Елементи a[6], …, a[10] утворюють піраміду				
6	11	3	15	18	13	1	0	4	11	5	5	10		Батько a[5] більший за сина.
6	11	3	15	18	13	1	0	4	11	4	4	8	9	Батько a[4] більший за синів.
6	11	3	15	18	13	1	0	4	11	3	3	6	7	Батько a[3] не більший за синів. Порушення піраміди
6	11	13	15	18	3	1	0	4	11	3				Проштовхуємо a[3]. Перестановка a[3], a[6]. Далі a[6] немає синів
6	11	13	15	18	3	1	0	4	11	2	2	4	5	Батько a[2] не більший за синів. Порушення піраміди
6	18	13	15	11	3	1	0	4	11	2				Проштовхуємо a[2]. Перестановка a[2], a[5].
6	18	13	15	11	3	1	0	4	11	2	5	10		a[5] не менший за сина
6	18	13	15	11	3	1	0	4	11	1	1	2	3	Батько a[1] не більший за синів. Порушення піраміди
18	6	13	15	11	3	1	0	4	11	1				Проштовхуємо a[1]. Перестановка a[1], a[2].
18	6	13	15	11	3	1	0	4	11	1	2	4	5	Батько a[2] не більший за синів. Порушення піраміди
18	15	13	6	11	3	1	0	4	11	1				Проштовхуємо a[2]. Перестановка a[2], a[4].
18	15	13	6	11	3	1	0	4	11	1	4	8	9	Батько a[4] більший за синів.
18	15	13	6	11	3	1	0	4	11	Піраміда побудована				

Фаза вибору

1	2	3	4	5	6	7	8	9	10	i	Б	ЛС	ПС	
18	15	13	6	11	3	1	0	4	11	10				На вершині піраміди найбільший елемент. Перестановка a[1], a[10].
11	15	13	6	11	3	1	0	4	18	10	1	2	3	Елемент a[1] може порушити піраміду. Батько a[1] не більший за синів. Порушення піраміди
15	11	13	6	11	3	1	0	4	18	10				Проштовхуємо a[1]. Перестановка a[1], a[2].
15	11	13	6	11	3	1	0	4	18	10	2	4	5	Батько a[2] не менший за синів. Побудована піраміда для елементів a[1],…, a[9]
15	11	13	6	11	3	1	0	4	18	9				На вершині піраміди найбільший елемент.
4	11	13	6	11	3	1	0	15	18	9				Перестановка a[1], a[9].
4	11	13	6	11	3	1	0	15	18	9	1	2	3	Елемент a[1] може порушити піраміду. Батько a[1] не більший за синів. Порушення піраміди
13	11	4	6	11	3	1	0	15	18	9				Проштовхуємо a[1]. Перестановка a[1], a[3].
13	11	4	6	11	3	1	0	15	18	9	3	6	7	Батько a[3] не менший за синів. Побудована піраміда для елементів a[1],…, a[8]. На вершині піраміди найбільший елемент.
0	11	4	6	11	3	1	13	15	18	8				Перестановка a[1], a[8].
0	11	4	6	11	3	1	13	15	18	8	1	2	3	Елемент a[1] може порушити піраміду. Батько a[1] не більший за синів. Порушення піраміди
0	11	4	6	11	3	1	13	15	18	8				Проштовхуємо a[1]. Перестановка a[1], a[2].
11	0	4	6	11	3	1	13	15	18	8	2	4	5	Батько a[2] не більший за синів. Порушення піраміди. Проштовхуємо a[2]. Перестановка a[2], a[5].
11	11	4	6	0	3	1	13	15	18	8	5			Батько a[5] не має синів. Побудована піраміда для елементів a[1],…, a[7]
11	11	4	6	0	3	1	13	15	18	7				На вершині піраміди найбільший елемент.
1	11	4	6	0	3	11	13	15	18	7				Перестановка a[1], a[7].
1	11	4	6	0	3	11	13	15	18	7	1	2	3	Елемент a[1] може порушити піраміду. Батько

														а[1] не більший за синів. Порушення піраміди
11	1	4	6	0	3	11	13	15	18	7				Проштовхуємо а[1]. Перестановка а[1], а[2].
11	1	4	6	0	3	11	13	15	18	7	2	4	5	Батько а[2] не більший за синів. Порушення піраміди.
11	6	4	1	0	3	11	13	15	18	7				Проштовхуємо а[2]. Перестановка а[2], а[4].
11	6	4	1	0	3	11	13	15	18	7	4			Батько а[4] не має синів. Побудована піраміда для елементів а[1],…, а[6]. На вершині піраміди найбільший елемент.
3	6	4	1	0	11	11	13	15	18	6				Перестановка а[1], а[6].
3	6	4	1	0	11	11	13	15	18	6	1	2	3	Елемент а[1] може порушити піраміду. Батько а[1] не більший за синів. Порушення піраміди
6	3	4	1	0	11	11	13	15	18	6				Проштовхуємо а[1]. Перестановка а[1], а[2].
6	3	4	1	0	11	11	13	15	18	6	2	4	5	Батько а[2] не менший за синів. Побудована піраміда для елементів а[1],…, а[5]. На вершині піраміди найбільший елемент.
0	3	4	1	6	11	11	13	15	18	5				Перестановка а[1], а[5].
0	3	4	1	6	11	11	13	15	18	5	1	2	3	Елемент а[1] може порушити піраміду. Батько а[1] не більший за синів. Порушення піраміди
4	3	0	1	6	11	11	13	15	18	5				Проштовхуємо а[1]. Перестановка а[1], а[3].
4	3	0	1	6	11	11	13	15	18	5	3			Батько а[3] не має синів. Побудована піраміда для елементів а[1],…, а[4]. На вершині піраміди найбільший елемент.
1	3	0	4	6	11	11	13	15	18	4				Перестановка а[1], а[4].
1	3	0	4	6	11	11	13	15	18	4	1	2	3	Елемент а[1] може порушити піраміду. Батько а[1] не більший за синів. Порушення піраміди
3	1	0	4	6	11	11	13	15	18	4				Проштовхуємо а[1]. Перестановка а[1], а[2].
3	1	0	4	6	11	11	13	15	18	4	2			Батько а[2] не має синів. Побудована піраміда для елементів а[1],…, а[3]. На

														вершині піраміди найбільший елемент.
0	1	3	4	6	11	11	13	15	18	3				Перестановка a[1], a[3].
0	1	3	4	6	11	11	13	15	18	3	1	2		Елемент a[1] може порушити піраміду. Батько a[1] не більший за синів. Порушення піраміди
1	0	3	4	6	11	11	13	15	18	3				Проштовхуємо a[1]. Перестановка a[1], a[2].
1	0	3	4	6	11	11	13	15	18	3	2			Батько a[2] не має синів. Побудована піраміда для елементів a[1], a[2]. На вершині піраміди найбільший елемент.
0	1	3	4	6	11	11	13	15	18	2				Перестановка a[1], a[2].
0	1	3	4	6	11	11	13	15	18	2	1			Піраміда
0	1	3	4	6	11	11	13	15	18	Відсортований масив				

Сортування методом злиття

Нехай дано два упорядковані масиви (файли) $X[1] \le X[2] \le \ldots \le X[n]$ та $Y[1] \le Y[2] \le \ldots \le Y[m]$. В основі сортування методом злиття лежить об'єднання цих масивів у один відсортований масив $Z[1] \le Z[2] \le \ldots \le Z[n+m]$. Для цього масиви X та Y паралельно переглядаються, і на кожному кроці менше з двох значень одного з масивів заноситься до Z. Фрагмент цього алгоритму має вигляд:

```
i:=1; j:=1;
while (i<=n) and (j<=m) do
   if X[i]<Y[j] then
         begin Z[i+j-1]:=X[i]; i:=i+1 end
   else
         begin Z[i+j-1]:=Y[j]; j:=j+1 end;
for i:=i to n do Z[j+i-1]:=X[i];{якщо масив X не вичерпано}
for j:=j to m do Z[i+j-1]:=Y[j];{якщо масив Y не вичерпано}
```

Дж. фон Нейман запропонував метод сортування злиттям, у якому реалізовано принцип «розділяй та володарюй». Масив ділиться навпіл, до кожної половини застосовується рекурсивно та сама процедура сортування злиттям, а відсортовані частини з'єднуються в один упорядкований масив. Отже, базовою операцією методу є злиття двох упорядкованих масивів у один. Час злиття упорядкованих масивів лінійно залежить від їх сумарної довжини. У цілому, алгоритм фон Неймана потребує виконання $O(n*\log_2 n)$ базових операцій над елементами масиву розмірності n.

☺ Індивідуальні завдання

А

Упорядкувати масив із 10 елементів за спаданням

1	2	3	4	5	6	7	8	9	10	Номер варіанту	Метод сортування
6	11	3	15	18	13	1	0	4	11	1	Метод бульбашок
11	5	3	15	18	13	1	1	4	16	2	Метод вставок
4	11	3	15	8	13	1	0	4	6	3	Метод вибору
11	7	3	15	8	13	1	2	4	6	4	Метод бульбашок
6	11	3	5	18	13	1	0	4	11	5	Метод вставок
6	11	3	14	18	13	1	7	4	1	6	Метод вибору
6	7	3	15	12	13	1	0	4	11	7	Метод бульбашок
6	4	3	5	11	13	1	10	11	11	8	Метод вставок
6	4	3	6	18	3	1	0	11	14	9	Метод вибору
6	4	3	15	18	13	1	0	8	11	10	Метод бульбашок

Б

Упорядкувати масив із 10 елементів за спаданням методом швидкого сортування

1	2	3	4	5	6	7	8	9	10	Номер варіанту
6	7	3	15	8	13	1	0	4	11	1
11	15	3	15	18	13	1	11	4	16	2
14	11	3	15	8	13	1	0	4	6	3
11	7	13	15	8	13	1	2	4	16	4
6	11	3	15	18	13	1	0	4	1	5
6	11	3	14	18	3	1	7	4	1	6
2	7	3	15	12	3	11	0	4	11	7
6	4	3	5	11	13	11	10	12	11	8
6	5	3	16	18	3	11	0	11	14	9
6	4	13	15	18	13	1	0	18	11	10

В

Упорядкувати масив із 10 елементів за зростанням методом пірамідального сортування

1	2	3	4	5	6	7	8	9	10	Номер варіанту
6	7	3	15	8	13	1	0	4	11	1
11	15	3	15	18	13	1	11	4	16	2
14	11	3	15	8	13	1	0	4	6	3
11	7	13	15	8	13	1	2	4	16	4
6	11	3	15	18	13	1	0	4	1	5
6	11	3	14	18	3	1	7	4	1	6
2	7	3	15	12	3	11	0	4	11	7
6	4	3	5	11	13	11	10	12	11	8
6	5	3	16	18	3	11	0	11	14	9
6	4	13	15	18	13	1	0	18	11	10

Тема 11: Пошук і хешування

Теоретичні відомості

Задача пошуку є фундаментальною в комбінаторних алгоритмах. У загальному її можна сформулювати так: «Дослідити множину Т з метою відшукання елемента, що задовольняє деяку умову С». Ефективність алгоритму пошуку суттєво залежить від структури множини Т.

Для отримання відповіді на питання «скільки існує способів…», «перерахуйте всі можливі…» та інше, зазвичай треба дослідити множини всіх можливих розв'язків, тобто виконати так званий вичерпний пошук. Існують загальні методи вичерпного пошуку – пошук із поверненням (бектрекінг) та метод решета. Пошук із поверненням використовується у широкому класі задач, що включають граматичний розбір, ігри та складання розкладів. На кожному кроці вичерпного пошуку робиться спроба розширити частковий розв'язок із метою отримання шуканого розв'язку. Метод решета є логічним доповненням вичерпного пошуку і полягає у виключенні тих об'єктів, що не є розв'язками. Цей метод корисний перш за все для теоретико-числових задач.

Важливим класом алгоритмів пошуку є алгоритми на графах, зокрема деревах. До них відносять алгоритми пошуку найкоротших шляхів між вершинами графа, пошуки в глибину і ширину, пошук максимального потоку в мережі, алгоритми обходу дерев.

Будемо розглядати структури даних, які є масивами або файлами. Для цих структур найпростішими алгоритмами пошуку є послідовний та бінарний (двійковий) пошук.

Послідовний пошук

Під послідовним пошуком розуміють дослідження елементів в тому порядку, у якому вони розташовані в послідовності (масиві, файлі). Нехай X[1], X[2],…, X[n] – масив із n елементів, і в цьому масиві треба знайти елемент Z, якщо він є. Алгоритм послідовного пошуку полягає у послідовному порівнянні елементів масиву X, починаючи з першого, із Z до тих пір, поки не знайдемо Z або не з'ясуємо, що Z у масиві немає. Доцільно доповнити масив елементом, який шукаємо, тобто присвоїти X[n+1]:=Z. Тоді алгоритм послідовного пошуку матиме вигляд:

```
X[n+1]:=Z;
i:=1;
while Z<>X[i] do i:=i+1;
if i<=n then {елемент X[i] співпадає із Z}
        else {Z не міститься в масиві}
```

До основних алгоритмів для послідовностей елементів відносять алгоритми пошуку мінімального чи максимального елемента. Такі алгоритми легко реалізувати через послідовний пошук, вибравши за початковий мінімальний (максимального) елемент перший елемент послідовності. Для прикладу, алгоритм пошуку мінімального елемента масиву X має вигляд:

```
K:=1; {K – індекс мінімального елемента}
Min:=X[1];
for i:=1 to n do
  if Min>X[i] then
        begin Min:=X[i]; K:=i end;
```

Даний алгоритм знаходить мінімальний елемент та його індекс.

Ефективність алгоритму послідовного пошуку в гіршому випадку рівна O(n), бо треба переглядати всі елементи масиву, якщо шуканого елемента немає.

Приклад 1

У масиві a із 10 елементів послідовним пошуком відшукати елемент 4: 6, 11, 3, 15, 18, 13, 1, 0, 4, 11. Доповнюємо масив 11-м елементом, який будемо шукати.

Індекси елементів масиву											**i**	**Коментар**
1	2	3	4	5	6	7	8	9	10	Z		
6	11	3	15	18	13	1	0	4	11	4	Початковий масив	
6	11	3	15	18	13	1	0	4	11	4	1	a[1]≠4
6	11	3	15	18	13	1	0	4	11	4	2	a[2]≠4
6	11	3	15	18	13	1	0	4	11	4	3	a[3]≠4
6	11	3	15	18	13	1	0	4	11	4	4	a[4]≠4
6	11	3	15	18	13	1	0	4	11	4	5	a[5]≠4
6	11	3	15	18	13	1	0	4	11	4	6	a[6]≠4
6	11	3	15	18	13	1	0	4	11	4	7	a[7]≠4
6	11	3	15	18	13	1	0	4	11	4	8	a[8]≠4
6	11	3	15	18	13	1	0	4	11	4	9	a[9]=4. i<=10. a[9] шукане число.

Приклад 2

У масиві a із 10 елементів послідовним пошуком відшукати елемент 7: 6, 11, 3, 15, 18, 13, 1, 0, 4, 11. Доповнюємо масив 11-м елементом, який будемо шукати.

Індекси елементів масиву											**i**	**Коментар**
1	2	3	4	5	6	7	8	9	10	Z		
6	11	3	15	18	13	1	0	4	11	7	Початковий масив	
6	11	3	15	18	13	1	0	4	11	7	1	a[1]≠7
6	11	3	15	18	13	1	0	4	11	7	2	a[2]≠7
6	11	3	15	18	13	1	0	4	11	7	3	a[3]≠7
6	11	3	15	18	13	1	0	4	11	7	4	a[4]≠7
6	11	3	15	18	13	1	0	4	11	7	5	a[5]≠7
6	11	3	15	18	13	1	0	4	11	7	6	a[6]≠7
6	11	3	15	18	13	1	0	4	11	7	7	a[7]≠7
6	11	3	15	18	13	1	0	4	11	7	8	a[8]≠7
6	11	3	15	18	13	1	0	4	11	7	9	a[9]≠7
6	11	3	15	18	13	1	0	4	11	7	10	a[10]≠7
6	11	3	15	18	13	1	0	4	11	7	11	a[11]=7. i>10. Числа 7 у масиві немає

Бінарний пошук

Цей вид пошуку застосовується до упорядкованих послідовностей. Нехай дано упорядкований масив X[1]≤X[2]≤ … ≤X[n], у якому треба знайти елемент Z. Спочатку Z порівнюється із середнім елементом масиву, тобто з елементом X[(n+1) div 2]. Якщо Z = X[(n+1) div 2], то пошук завершується. Якщо Z < X[(n+1) div 2], то розглядаємо ліву частину масиву, тобто X[1], X[2],…, X[(n+1) div 2 -1], а якщо Z > X[(n+1) div 2], то розглядаємо праву частину масиву, тобто X[(n+1) div 2 +1], X[(n+1) div 2 +2],…, X[n]. У вибраній частині шукаємо Z. Для цього Z знову порівнюємо із середнім елементом вибраної чистини масиву, і якщо знову Z не співпадає із середнім елементом, то визначаємо нову частину масиву, у якій будемо шукати Z. На кожному кроці алгоритму вибрана частина масиву для пошуку Z ділиться навпіл. Тому алгоритм припинить свою роботу, якщо буде знайдено Z або розмір частини масиву стане рівним 1. Фрагмент алгоритму бінарного пошуку для масиву X має вигляд.

```
J:=0;          {J – індекс елемента масиву X, який буде співпадати з Z}
L:=1;          {L – індекс лівого краю частини масиву, вибраної для пошуку}
R:=n;          {R – індекс правого краю частини масиву, вибраної для пошуку}
while L<=R do
begin
      m:= (L+R) div 2;     {m – індекс середнього елемента}
      if Z<X[m] then R:= m-1
            else if Z>X[m] then L:= m+1
                  else begin J:= m; break; end
end;
if J>0 then {елемент Z співпадає з X[J]}
      else {елемент Z не міститься в масиві}
```

Якщо початковий масив упорядкований за спаданням, то в цьому фрагменті алгоритму треба поміняти вибір лівого та правого країв масиву, у якому здійснюється пошук. Алгоритм бінарного пошуку дуже ефективний, і його часова складність $O(\log_2 n)$.

Приклад 3

У впорядкованому за спаданням масиві a із 11 елементів бінарним пошуком відшукати елемент 4: 16, 13, 11, 11, 10, 8, 6, 4, 4, 2,1.

Індекси елементів масиву											Z	L	R	J	Коментар
1	2	3	4	5	6	7	8	9	10	11					
16	13	11	11	10	8	6	4	4	2	1	4	1	11	0	Початковий масив
16	13	11	11	10	8	6	4	4	2	1		1	11		m=6, a[6]>Z, L=7
16	13	11	11	10	8	6	4	4	2	1		7	11	9	m=9, a[9]=Z, J=9, пошук завершений

Приклад 4

У впорядкованому за спаданням масиві a із 11 елементів бінарним пошуком відшукати елемент 7: 16, 13, 11, 11, 10, 8, 6, 4, 4, 2,1.

Індекси елементів масиву											Z	L	R	J	Коментар
1	2	3	4	5	6	7	8	9	10	11					
16	13	11	11	10	8	6	4	4	2	1	7	1	11	0	Початковий масив.
16	13	11	11	10	8	6	4	4	2	1		1	11		m=6, a[6]>Z, L=7
16	13	11	11	10	8	6	4	4	2	1		7	11		m=9, a[9]<Z, R=8
16	13	11	11	10	8	6	4	4	2	1		7	8		m=7, a[7]<Z, R=6
16	13	11	11	10	8	6	4	4	2	1		7	6	0	L>R, цикл **while** L<=R припинив роботу, J=0 – елемента Z у масиві немає

Хешування

Хешування використовується для представлення множин у вигляді хеш-таблиці. За допомогою хеш-таблиць можна ефективно реалізувати операції пошуку, добавлення і вилучення елементів із множини.

Розрізняють дві форми хешування. Одна з них називається *відкритим*, або *зовнішнім хешуванням*, у якій відсутнє обмеження на розмір множини. Друга називається *закритим*, або *внутрішнім хешуванням* і використовує обмеження на розмір множини.

Відкрите хешування

На Рис. 11.1 зображена базова структура даних при відкритому хешуванні. Основна ідея полягає в тому, що вся множина розбивається на скінченну кількість класів (підмножин). Класи на малюнку названі сегментами та їх B штук. Хеш-таблиця сегментів – це масив вказівників з індексами від 0 до B-1 на списки елементів кожного сегмента. Розподіл елементів множини по сегментах здійснюється за допомогою хеш-функції *h*. Для будь-якого елемента *x* заданої множини функція *h(x)* приймає цілочислове значення із інтервалу від 0 до B-1, що відповідає сегменту, якому належить елемент *x*. Елемент *x* часто називають ключем, *h(x) – хеш-значенням x*.

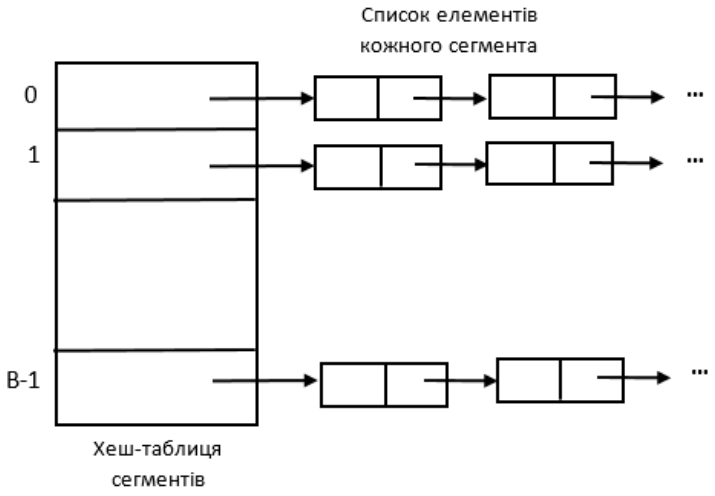


Рис. 11.1. Організація даних при відкритому хешуванні.

Хеш-таблиця сегментів проіндексована номерами сегментів 0, 1, …, B-1 і містить заголовки для B списків. Елемент *x* *i*-го списку, це елемент початкової множини, для якого *h(x)=i*.

Якщо списки елементів сегментів приблизно однакового розміру, то середня довжина списків буде рівна N/B, де N – кількість елементів початкової множини. Якщо можна оцінити величину N і вибрати B дуже близьким до N, то в кожному списку буде один-два елементи. Тоді час виконання операцій пошуку, добавлення та вилучення елементів у списки буде малою сталою величиною.

Не завжди ясно, як вибрати хеш-функцію *h* так, щоб вона приблизно порівно розподіляла елементи початкової множини по всіх сегментах.

Приклад 5

Задати множину із 20-ти натуральних чисел і побудувати для неї хеш-таблицю із 10-ти сегментів, вибравши хеш-функцію *h*.

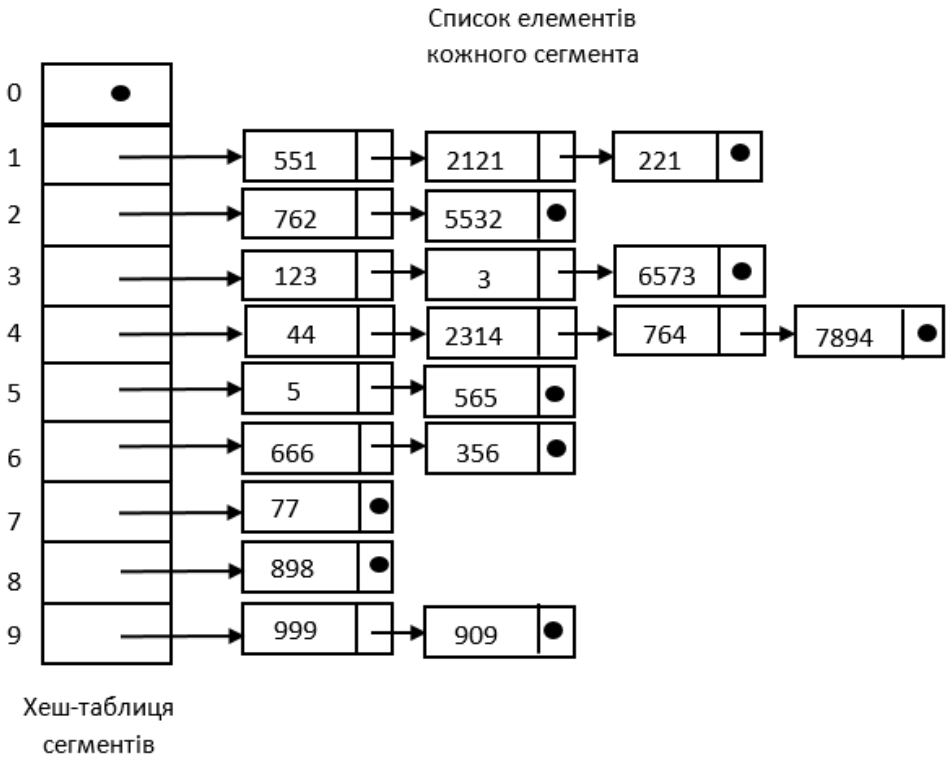
Розв'язання

1. Виберемо такі 20 натуральних чисел: 123, 3, 762, 44, 551, 6573, 2314, 5532, 5, 666, 356, 999, 909, 764, 2121, 221, 7894, 565, 898, 77. Номери сегментів будуть такі: 0, 1, , 9. Виберемо таку хеш-функцію *h*: *h(x)=x* mod 10 – остача від ділення числа *x* на 10. Результати обчислення функції *h* подані в наступній таблиці:

№	Число	*h(x)*
1	123	3
2	3	3
3	762	2
4	44	4
5	551	1
6	6573	3
7	2314	4
8	5532	2
9	5	5
10	666	6

№	Число	*h(x)*
11	356	6
12	999	9
13	909	9
14	764	4
15	2121	1
16	221	1
17	7894	4
18	565	5
19	898	8
20	77	7

Із заданих 20-ти чисел 3 числа мають остачу 1, 2 – остачу 2, 3 – остачу 3, 4 – остачу 4, 2 – остачу 5, 2 – остачу 6, 1 – остачу 7, 1 – остачу 8, 2 – остачу 9. Зобразимо представлення множини чисел у вигляді хеш-таблиці.



Запропонована хеш-функція $h(x)=x \bmod 10$ не рівномірно розподіляє елементи по сегментах.

Основними операціями, що застосовуються до хеш-таблиць, є пошук, добавлення та вилучення елементів. Виконання кожної такої операції починається з того, що для елемента x заданої множини обчислюється значення хеш-функції $h(x)$, тобто знаходиться номер сегмента для x. Після цього операції пошуку, добавлення чи вилучення елемента x виконується над списком елементів знайденого сегмента. Якщо списки майже однакового розміру і короткі, то фактично ці операції виконуються за фіксований час.

2. Розглянемо тепер іншу хеш-функцію, яку можна застосовувати до множини слів. Будемо вважати цифри від 0 до 9 символами, а задані 20 чисел – словами із цих символів. Коди символів наведені в наступній таблиці.

Символ	0	1	2	3	4	5	6	7	8	9
Код	48	49	50	51	52	53	54	55	56	57

Розглянемо тепер наступну хеш-функцію $h(x)$, псевдокод якої має вигляд

```
sum:=0;
for i:=1 to n do
        sum:=sum+ord(x[i]);
h:=sum mod B;
```

Функція ord(x[i]) обчислює код і-го символу слова x, яке складається із n символів. У змінній sum накопичується сума кодів символів цього слова. Значенням функції $h(x)$ є остача від ділення суми sum на кількість сегментів.

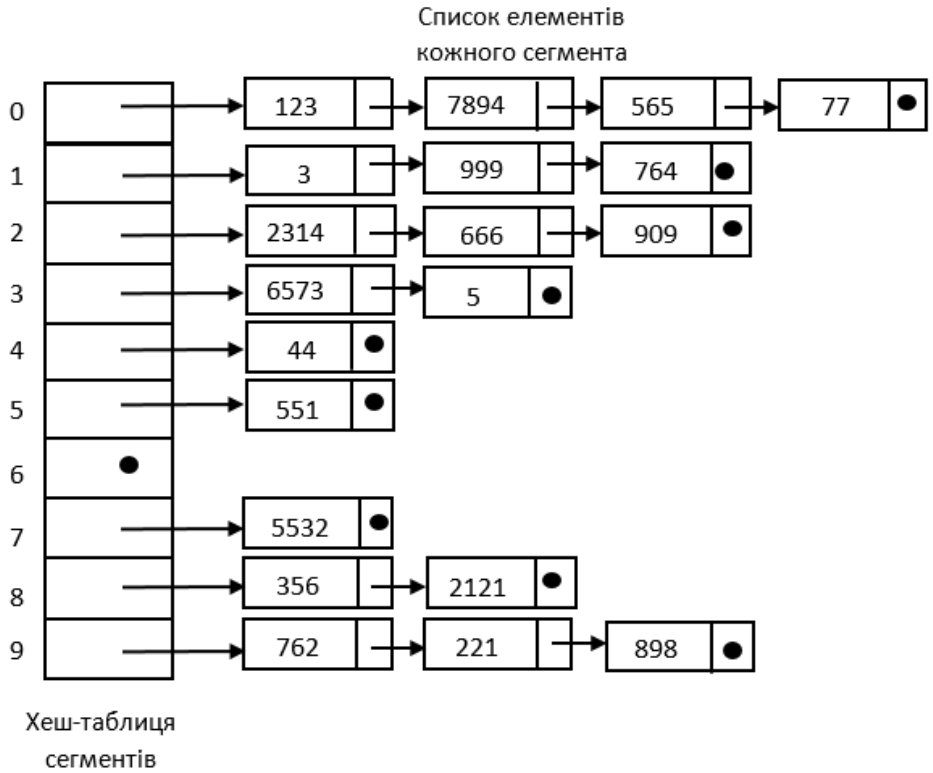
Для прикладу розглянемо обчислення функції *h(x)* для слова 123. Значення змінної sum на основі кодів символів буде дорівнювати sum = 49+50+51=150. Тоді *h(123)* = 150 mod 10 = 0 при B=10.

Результати обчислення функції *h* при B=10 подані в наступній таблиці:

№	Число	sum	h(x)
1	123	150	0
2	3	51	1
3	762	159	9
4	44	104	4
5	551	155	5
6	6573	213	3
7	2314	202	2
8	5532	207	7
9	5	53	3
10	666	162	2

№	Число	sum	h(x)
11	356	158	8
12	999	171	1
13	909	162	2
14	764	161	1
15	2121	198	8
16	221	149	9
17	7894	220	0
18	565	160	0
19	898	169	9
20	77	110	0

Хеш-таблиця цієї множини слів має вигляд:



Як бачимо, розглянута хеш-функція теж не завжди рівномірно розподіляє елементи по сегментах. Однак вона придатна для розподілу слів по сегментах, а не тільки чисел.

Для більш рівномірного розподілу елементів по сегментах можна використати хеш-функцію, яка визначається для заданого числа x наступним чином: число x підноситься до квадрату; із отриманого числа x^2 утворюють нове число y, вибираючи декілька середніх цифр; далі покладають $h(x)=y$ mod B.

Закрите хешування

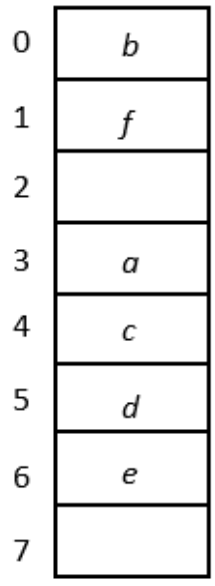
Закриті хеш-таблиці особливо ефективні, коли максимальні розміри вхідного набору даних вже відомі.

При закритому хешуванні в таблиці сегментів зберігаються безпосередньо елементи заданої множини, а не заголовки списків. У цьому випадку хеш-таблиця аналогічна одновимірному масиву. В кожному сегменті може зберігатися тільки один елемент множини. При закритому хешуванні застосовується методика **повторного хешування**. Ця методика потрібна для вирішення *колізій*. Якщо захочемо добавити в таблицю елемент *x* в сегмент з номером *h(x)*, який уже зайнятий іншим елементом, то виникає *колізія*. Для елемента *x* потрібно знайти інший вільний сегмент. У відповідності з методикою повторного хешування вибирається послідовність інших номерів сегментів $h_1(x), h_2(x), \ldots$ куди можна помістити елемент *x*. Кожний із цих сегментів послідовно перевіряється, поки не буде знайдений вільний. Якщо вільних сегментів немає, то значить, що таблиця заповнена і елемент *x* не можна вставити.

Приклад 6

Нехай хеш-таблиця складається із восьми сегментів B=8. Елементи *a, b, c, d, e, f* мають хеш-значення *h(a)*=3, *h(b)*=0, *h(c)*=4, *h(d)*=3, *h(e)*=5, *h(f)*=0. Застосуємо просту методику повторного хешування, яка називається *лінійним хешуванням*. При лінійному хешуванні $h_i(x) = (h(x) + i)\ mod\ B$.

Будемо вважати, що спочатку вся хеш-таблиця порожня, тобто в кожний сегмент поміщено спеціальне значення *empty* (порожній), яке не співпадає ні з одним елементом заданої множини. Тепер послідовно вставимо елементи *a, b, c, d, e, f* в порожню таблицю: елемент *a* потрапляє в сегмент 3, бо *h(a)*=3; елемент *b* – в сегмент 0 (*h(b)*=0); а елемент *c* – в сегмент 4 (*h(c)*=4). Для елемента *d h(d)*=3, але сегмент 3 вже зайнятий. Виникла колізія. Застосовуємо функцію $h_1(d) = (h(d) + 1)\ mod\ 8 = (3 + 1)\ mod\ 8 = 4$. Елемент *d* потрібно помістити в сегмент 4, який також зайнятий.



Хеш-таблиця сегментів

Далі застосовуємо функцію

$$h_2(d) = (h(d) + 2)\ mod\ 8 = (3 + 2)\ mod\ 8 = 5.$$

Сегмент 5 вільний, заносимо туди d. Для елемента *e h(e)*=5, але сегмент 5 вже зайнятий. Застосовуємо функцію $h_1(e) = (h(e) + 1)\ mod\ 8 = (5 + 1)\ mod\ 8 = 6.$

Сегмент 6 вільний, заносимо туди e. Для елемента *f h(f)*=0, але сегмент 0 вже зайнятий. Застосовуємо функцію $h_1(f) = (h(f) + 1)\ mod\ 8 = (0 + 1)\ mod\ 8 = 1.$

Сегмент 1 вільний, заносимо туди *f*. Результат заповнення хеш-таблиці показаний на малюнку.

При пошуку елемента x потрібно переглянути всі місцезнаходження $h(x), h_1(x), h_2(x), \ldots$ поки не буде знайдений x або поки не зустрінеться порожній сегмент. В останньому випадку елемента x в таблиці немає.

Однак, якщо в таблиці виконувати операції вилучення елементів, то при досягненні порожнього сегмента ми, не знайшовши x, не можемо бути впевнені, що елемента x в таблиці немає. Потрібно в сегмент, у якому вилучили елемент, помістити спеціальну константу *deleted* (вилучений). Константи *empty* і *deleted* мають різне призначення. Константа *empty* міститься у сегментах, які ніколи не містити елементів. Тому при пошуку елемента x в таблиці цей пошук потрібно продовжити, якщо попали в сегмент з константою *deleted* і припинити пошук, якщо попали в сегмент з константою *empty*. Останній випадок означає, що елемента x в таблиці немає.

Методика повторного лінійного хешування приводить до групування заповнених сегментів у великі неперервні блоки, а значить пошук елементів стає неефективним. Фактично будь-яка методика повторного хешування, де чергова проба залежить від попередньої має групуючі властивості лінійного хешування. Можлива найпростіша методика, для якої проблеми «групування» не існує: для цього достатньо покласти $h_i(x) = (h(x) + d_i)\ mod\ B$, де $d_1, d_2, \ldots, d_{B-1}$ – «випадкові» перестановки чисел 1, 2, …, B-1.

☺ Індивідуальні завдання

А

У впорядкованому за не спаданням масиві із 10 елементів бінарним пошуком відшукати задані елементи.

1	2	3	4	5	6	7	8	9	10	Номер варіанту	Відшукати елементи
4	5	8	8	9	11	14	14	21	22	1	5, 15
4	6	6	10	10	14	15	18	22	25	2	5, 25
2	4	6	7	8	17	20	21	32	40	3	1, 32
3	5	8	10	10	10	14	17	20	23	4	4, 5
2	4	7	10	11	11	17	22	24	26	5	3, 7
4	6	9	10	14	14	15	16	21	22	6	2, 22
3	5	6	8	10	12	15	15	15	16	7	2, 16
3	6	8	11	13	16	18	20	24	25	8	2, 24
2	5	6	9	11	15	16	18	19	20	9	1, 20
2	4	6	8	10	12	14	16	18	20	10	2, 19

Б

Задати множину із 20-ти натуральних чисел і побудувати для неї хеш-таблицю із 10-ти сегментів, вибравши два види хеш-функцій h відповідно до прикладу 5.

B

Нехай хеш-таблиця складається із восьми сегментів B=8. Для елементів *a, b, c, d, e, f, g* задані хеш-значення. Заповнити закриту хеш-таблицю, застосовуючи методику повторного лінійного хешування (приклад 6).

h(a)	h(b)	h(c)	h(d)	h(e)	h(f)	h(g)	Номер варіанту
3	4	2	0	3	1	4	1
2	3	0	0	4	2	0	2
4	5	6	0	4	5	6	3
7	0	1	7	2	1	0	4
4	5	6	3	5	7	0	5
1	3	4	4	3	2	7	6
4	7	3	4	5	4	7	7
0	2	3	3	3	4	5	8
2	4	5	4	5	5	6	9
3	4	5	4	6	7	7	10

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Ахо А., Хопкрофт Д., Ульман Д. Структуры данных и алгоритмы. М.: Изд.дом «Вильямс», 2010.
2. Креневич А.П. Алгоритми і структури даних. Підручник. К.: ВПЦ "Київський Університет", 2021. 200 с.
3. Кублій Л.І. Алгоритми і структури даних. Основи алгоритмізації. Підручник. Київ: КПІ ім. Ігоря Сікорського, 2022
4. Матвієнко М. П. Теорія алгоритмів. Навчальний посібник. К.: Видавництво Ліра-К, 2017, 340 с.
5. Паращук С.Д., Сурков К.Ю., Ізвалов О.В. Практикум із програмування мовою С++: Навчальний посібник. Кропивницький: ЕТІ імені Роберта Ельворті, 2023, 260 с.
6. Теорія алгоритмів :: Рекурентні співвідношення. https://courses.prometheus.org.ua/c4x/KPI/Algorithms101/asset/04_Recurrences.pdf
7. Ткачук В.М. Алгоритми і структура даних: Навчальний посібник. Івано-Франківськ : Видавництво Прикарпатського національного університету імені Василя Стефаника, 2016.- 286 с.
8. Шкільняк С.С. Математична логіка; Основи теорії алгоритмів: Навчальний посібник. К.: ДП «Вид. дім «Персонал», 2009. – 280 с.

Формат 60x84 $^{1}/_{8}$
Обл. вид. арк. 4,16