ВИЩИЙ НАВЧАЛЬНИЙ ЗАКЛАД
«УНІВЕРСИТЕТ ЕКОНОМІКИ ТА ПРАВА «КРОК»

В.В. Троцько

ТЕОРІЯ АЛГОРИТМІВ

Навчально - методичний посібник

Київ – 2023 рік

УДК 510.5
Т70

Рекомендовано до друку Вченою радою
Університету «КРОК»
(протокол №2 від 27 жовтня 2022 року)

Рецензенти:
А.М. Котенко - *кандидат технічних наук доцент кафедри систем інформаційного та кібернетичного захисту Державного університету телекомунікацій;*
І.О. Чернозубкін - *кандидат технічних наук провідний науковий співробітник науково-дослідного відділу науково-дослідного управління науково-дослідного центру Центрального науково - дослідного інституту Збройних Сил України.*

В.В. Троцько
Т70 Теорія алгоритмів: Навчально - методичний посібник. – Київ: Університет економіки та права «КРОК», 2023 – 126 с.
ISBN 978-966-170-075-7

У навчально - методичному посібнику викладено найважливіші теми дисципліни «Теорія алгоритмів». Посібник містить теоретичний матеріал, що складається з восьми розділів та завдань для лабораторних робіт для закріплення отриманих знань на практиці. Посібник призначений для здобувачів вищої освіти та всіх, хто цікавиться питанням комп'ютерних обчислень.

ISBN 978-966-170-075-7

ЗМІСТ

ВСТУП

В інформаційних науках теорія алгоритмів відіграє важливу роль. Зародившись в 30-х роках XX сторіччя і розвиваючись до теперішнього часу вона поєднує в собі математичні принципи реалізації комп'ютерних рішень з практичними можливостями інформаційних систем. Сьогодні положення теорії алгоритмів реалізують практично в усіх комп'ютерних програмах, починаючи з операційних систем і закінчуючи засобами шифрування інформації.

В цьому посібнику розглянуті базові положення теорії алгоритмів, що містять в собі визначення алгоритмів та засоби їх подання, базові принципи теорії започатковані з перших років її створення, засоби аналітичні асимптотичного та часового аналізу, евристичні алгоритми та їх особливості, рекурсивні алгоритми, а також положення теорії для здійснення шифрування даних. Основний акцент в підручнику робиться на зв'язок між теоретичними положеннями та можливістю їх практичної реалізації. Для закріплення знань студентам пропонується комплекс лабораторних робіт, які можна виконувати як в аудиторіях, так і в режимі дистанційного навчання. В цих роботах також окрім інструкцій з виконання наводяться окремі теоретичні положення, що пов'язані з основним матеріалом, який надано в посібнику. Всі прикладні комп'ютерні програми, які використовуються для проведення лабораторних робіт використовуються на основі ліцензій на вільне програмне забезпечення або з дозволу авторів цих програм.

Розділ 1
ПОНЯТТЯ АЛГОРИТМУ. ВІЗУАЛІЗАЦІЯ АЛГОРИТМІВ. ПСЕВДОКОДИ

В теорії алгоритмів та інформаційних науках взагалі відсутнє канонічне визначення поняття алгоритм. Ряд авторів використовують різні визначення [1]. Це свідчить про неузгодженість окремих неформальних понять цього терміну. Сьогодні формальне визначення алгоритму є одним із завдань обчислювальної математики і інформатики над яким працюють теоретики. Якщо алгоритмом називати набір інструкцій для виконання деякого завдання це не буде фатальною помилкою для практикуючих програмістів тим більше, що таке визначення не змінює сутності розуміння алгоритму.

Основні вимоги до алгоритмів формулюють наступним чином [1].

1. Кожен алгоритм має справу з даними – вхідними, проміжними, вихідними. Для того, щоб уточнити поняття даних, фіксується кінцевий алфавіт вихідних символів (цифри, букви і т. п.) і вказуються правила побудови алгоритмічних об'єктів. Типовим використовуваним засобом є індуктивна побудова. Наприклад, визначення ідентифікатора в мові C++: ідентифікатор – це або буква, або ідентифікатор, до якого приписана праворуч або буква, або цифра. Слова кінцевої довжини в кінцевих алфавітах – найбільш звичайний тип алгоритмічних даних, а число символів в слові – природна міра об'єму даних. Інший випадок алгоритмічних об'єктів – формули. Прикладом можуть служити формули алгебри предикатів і алгебри висловлювань. У цьому випадку не кожне слово в алфавіті буде формулою.

2. Алгоритм для розміщення даних вимагає пам'яті. Пам'ять зазвичай вважається однорідною і дискретною, тобто вона складається з однакових комірок, причому кожна комірка може містити один символ даних, що дозволяє узгодити одиниці виміру обсягу даних і пам'яті.

3. Алгоритм складається з окремих елементарних кроків, причому множина різних кроків, з яких складений алгоритм, кінцеві. Типовий приклад множини елементарних кроків – система команд процесора.

4. Послідовність кроків алгоритму детермінована, тобто після кожного кроку вказується, який крок слід виконувати далі, або вказується, коли слід роботу алгоритму вважати закінченою.

5. Алгоритм повинен бути результативним, тобто зупинятися після кінцевого числа кроків (залежного від вхідних даних) з наданням результату. Дана властивість іноді називають збіжністю алгоритму.

6. Алгоритм передбачає наявність механізму реалізації, який за описом алгоритму породжує процес обчислення на основі вхідних даних. Передбачається, що опис алгоритму та механізм його реалізації кінцеві. Можна помітити аналогію з обчислювальними машинами. Вимога 1 відповідає цифровій природі ПК, вимога 2 – пам'ять ПК, вимога 3 – програмі машини, вимога 4 – її логічній природі, вимоги 5, 6 – обчислювальному пристрою і його можливостям.

Алгоритм можна описати словесно, де кожен пункт описуватиме відповідну дію. Такий спосіб використовують в окремих випадках, які вимагають ряду додаткових пояснень. Наприклад, алгоритм пошуку коренів квадратного рівняння можна подати наступним чином.

1. Знайти дискримінант D за формулою D= -4ac.

2. Якщо D<0, то квадратне рівняння не має коренів.

3. Якщо D=0, то рівняння має один корінь. Здійснити розрахунок значення кореня за формулою:

$$x_1 = \frac{-b}{2 \cdot a}. \qquad (1.1)$$

4. Якщо D>0, то рівняння має два корені. Здійснити розрахунок значень коренів за формулами:

$$x_1 = \frac{-b+\sqrt{D}}{2 \cdot a}, x_2 = \frac{-b-\sqrt{D}}{2 \cdot a}. \quad (1.2)$$

Після такого опису можна починати створювати код мовою високого рівня і запускати його на комп'ютері.

Однак, такий описовий спосіб алгоритмізації задач не набув широкого розповсюдження головним чином через відсутність універсальності та труднощів пояснення для різних мов програмування.

Існує також символічний спосіб який полягає в записі алгоритму за допомогою умовних символів. Такий спосіб подання алгоритму робить запис алгоритму дуже стислим, і не наочним. Для розуміння логіки роботи алгоритму записаного таким способом необхідні додаткові знання і певна практика роботи з кодами програм та готовими алгоритмічними рішеннями. Зрештою сам програмний код є способом запису алгоритму але для його розуміння (навіть якщо він містить коментарі) треба знати мову програмування на якій написаний цей код і всі особливості його реалізації з використанням такої мови. Перераховані способи не є зручними для розуміння логіки роботи того чи іншого алгоритму.

Щоб полегшити розуміння роботи того чи іншого алгоритму використовують спеціальні графічні побудови або блоки, що відображають логіку роботи алгоритмів. Використання таких блоків регулюється відповідним міжнародним стандартом [2], хоча в багатьох випадках практикуючі програмісти цього стандарту можуть строго не дотримуватися в більшості типових та розповсюджених алгоритмів для пояснення логіки роботи тієї чи іншої програми використовують саме їх. На рис. 1.1–1.3 показані окремі типові блоки для створення алгоритмів, що пов'язані з елементарними математичними та логічними операціями, а також із циклами, взяті із текстового редактора Microsoft Word. Під час кодування цикли в алгоритмах можна організовувати декількома способами як з використанням умовних розгалужень, рис. 1.2 так і з використанням стандартних блоків, рис. 1.3. Для подання алгоритмів із наведених блоків та інших стандартних графічних побудов створюють блок-схеми.

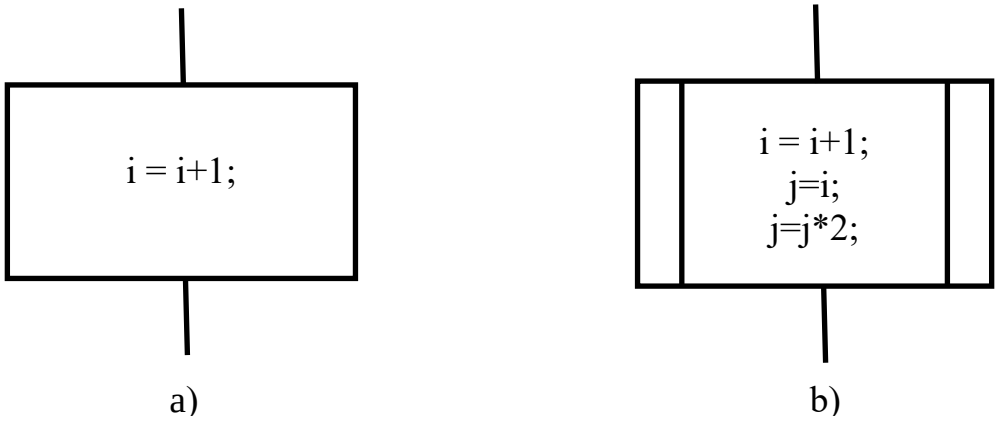


Рис. 1.1. Блоки для позначення виконуваної операції – a та групи операцій – b

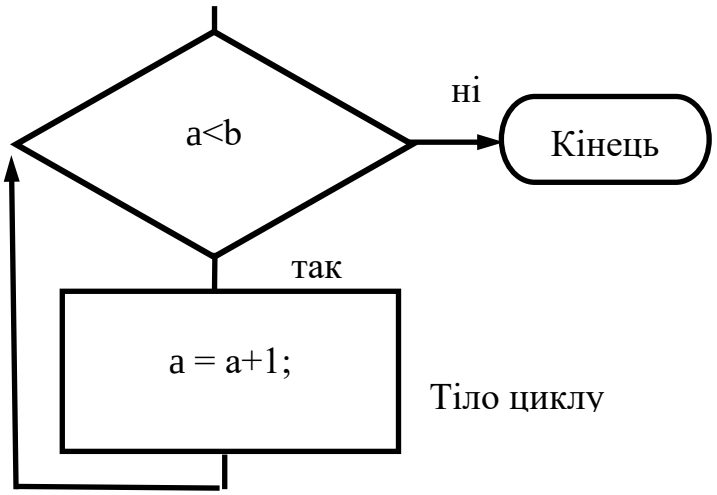


Рис. 1.2. Блок для позначення умовного розгалуження та організація циклу з його допомогою

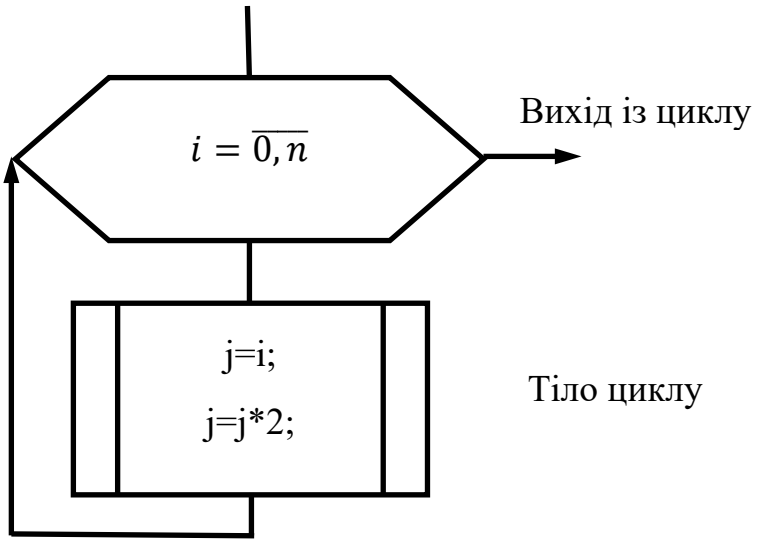


Рис. 1. 3. Приклад блоків для позначення циклу з використанням стандартного блоку для циклу

Приклад блок-схеми пошуку максимального елементу в одномірному масиві показаний на рис. 1.4.

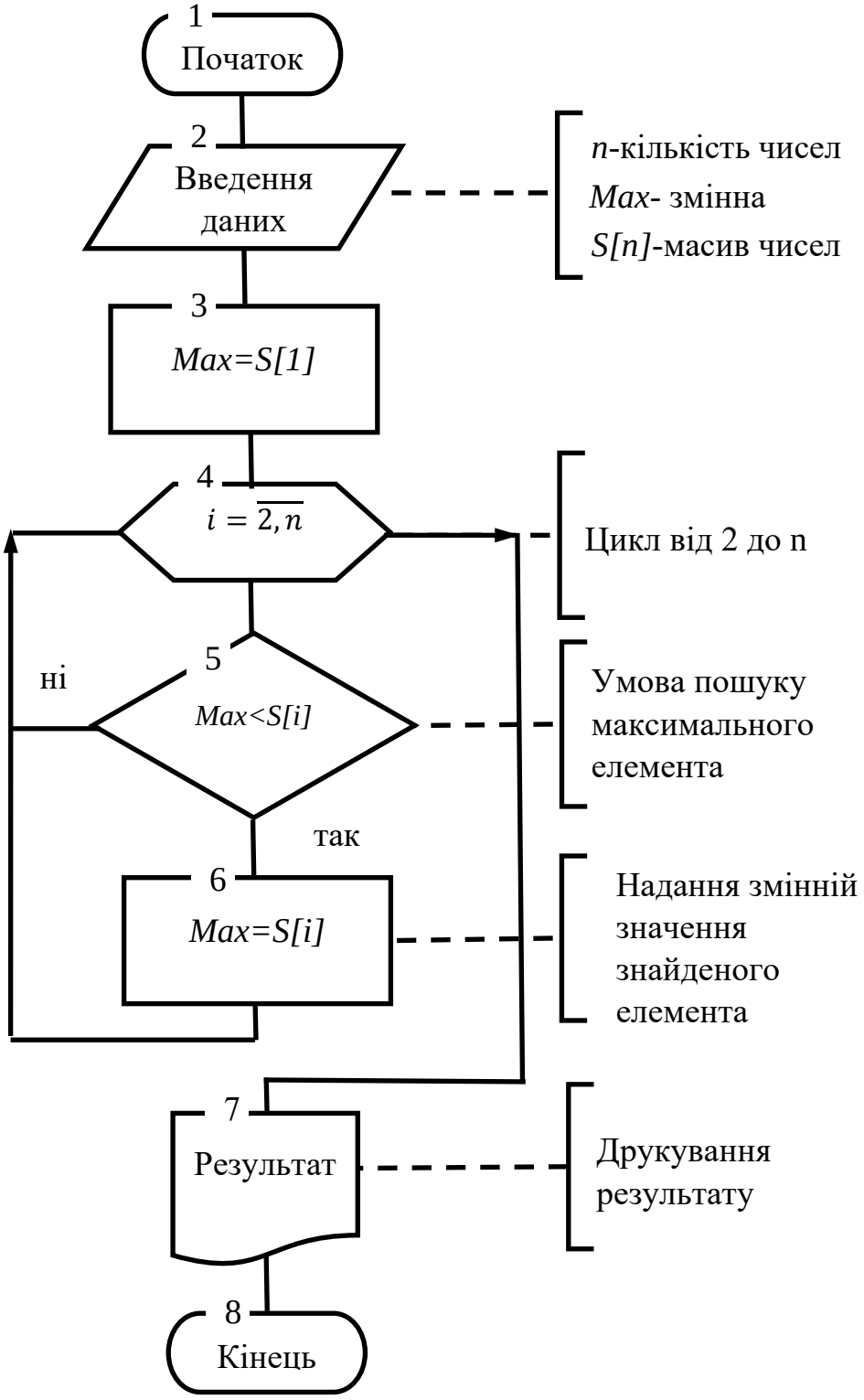


Рис. 1.4. Блок - схема алгоритму пошуку максимального елемента в одномірному масиві

Слід зазначити, що для графічного подання алгоритмів використовуються і інші графічні побудови. Зокрема, діаграми Насі - Шнейдермана, рис 1.5. та інші. Однак, використовуються вони не часто. В лабораторних робота, які наведені в додатку посібника використовуються блок-схеми із наведених стандартизованих блоків.

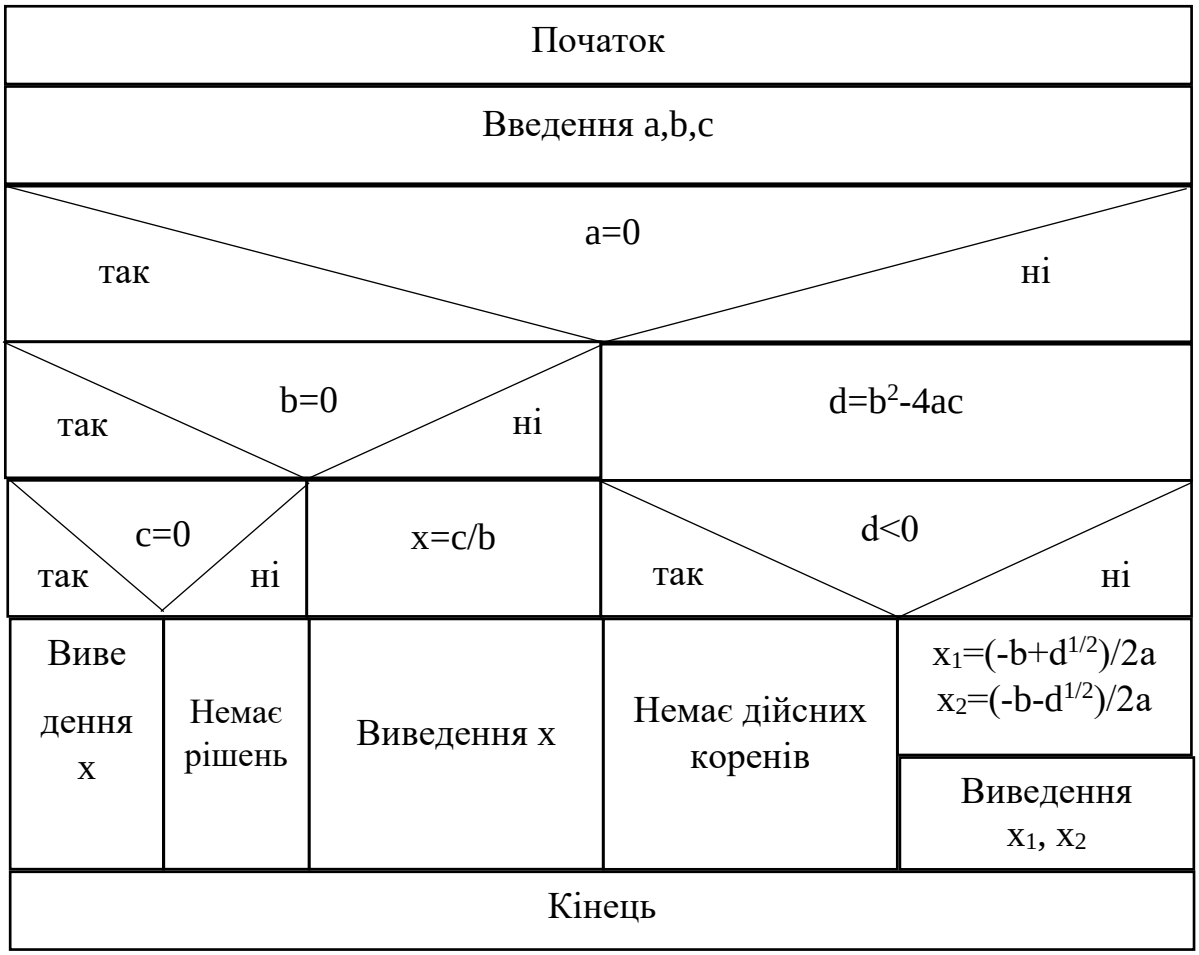


Рис. 1.5. Приклад графічного подання алгоритму обчислення коренів квадратного рівняння з допомогою діаграми Насі – Шнейдермана

При вивченні роботи алгоритмів також користуються універсальною мовою кодування високого рівня. Ця мова не застосовується в комп’ютерах. Її призначення – пояснення особливостей алгоритмів не залежно від того яка мова програмування використовується на практиці.

Такий псевдокод не містить спеціальних символів чи обов'язкових правил, таких, як наприклад, для написання змінних з малої чи великої літери або оголошення типів змінних тощо. Мета такого кодування – пояснення яким чином алгоритм може бути запрограмований мовою високого рівня. Нижче наведений фрагмент такого псевдокоду для алгоритму пошуку максимального елемента в одномірному масиві блок-схема якого показана на рис. 1.4.

MaxS (S,n; Max)//Назва програми, масив та змінні
Max ← S[1] //Присвоєння змінній значення першого елементу
For i ← 2 to n //Цикл від i=2 до змінної n
if Max < S[i] then Max ← S[i] //Умова відбору
end for //Закінчення циклу
return Max //Повернення знайденого числа
End //Закінчення програми

Блок-схеми та псевдокоди суттєво спрощують розуміння роботи алгоритмів в їх практичному виконанні. Використання цих інструментів дає можливість розібратися в тому, як працює той чи інший алгоритм не вдаючись до подробиць, пов'язаних зі специфікою використання різних мов програмування.

Контрольні питання

1. *Назвіть способи подання алгоритму.*
2. *Скільки способів задання циклу в алгоритмі можна використати користуючись блок-схемами?*
3. *Чому використання псевдокодів є зручним для роботи з алгоритмами?*
4. *В чому полягають труднощі розуміння роботи коду без графічного подання алгоритму?*

Розділ 2
МАШИНА ПОСТА ТА МАШИНА ТЮРІНГА І ЇХ ЗНАЧЕННЯ ДЛЯ ТЕОРІЇ АЛГОРИТМІВ

Не дивлячись на те, що поняття алгоритму виникло в математиці ще у глибокому середньовіччі основні наукові положення, що стосуються концепції застосування алгоритмів як частини машинних обчислень були розроблені лише у XX сторіччі. В 1930-х роках американський математик Еміль Пост та британський математик Алан Тюрінг заклали теоретичні основи алгоритмічної реалізації. Обчислювальні моделі, які були створені цими науковцями незалежно один від одного називають «машинами» через їх схожість з обчислювальними автоматами. Через схожість між собою їх також називають машиною Поста - Тюрінга [3].

В цьому посібнику в лабораторних роботах використовуються готові комп'ютерні моделі машини Поста і машини Тюрінга [4]. Машина Поста і машина Тюрінга схожі між собою і містять ряд спільних елементів. До таких елементів відносяться:

- стрічка (теоретично нескінченна) з набором комірок в яких можуть розташовуватися символи;
- каретка, що може рухатися над стрічкою і здатна виконувати дії над символами;
- дії над символами стрічки можна задавати (в сучасному розумінні програмувати).

Однак, існують і певні відмінності. Більш детально робота цих машин подана нижче.

Машина Поста

Створення машини Поста дозволило поглянути на алгоритм як на універсальний засіб для вирішення будь-якої проблеми, що може бути математично формалізована і переведена на мову однозначних інструкцій для виконання.

Тобто, якщо конкретну проблему вдалося сформулювати значить її можливо вирішити використовуючи набір інструкцій для обчислювального пристрою, рис 2.1.

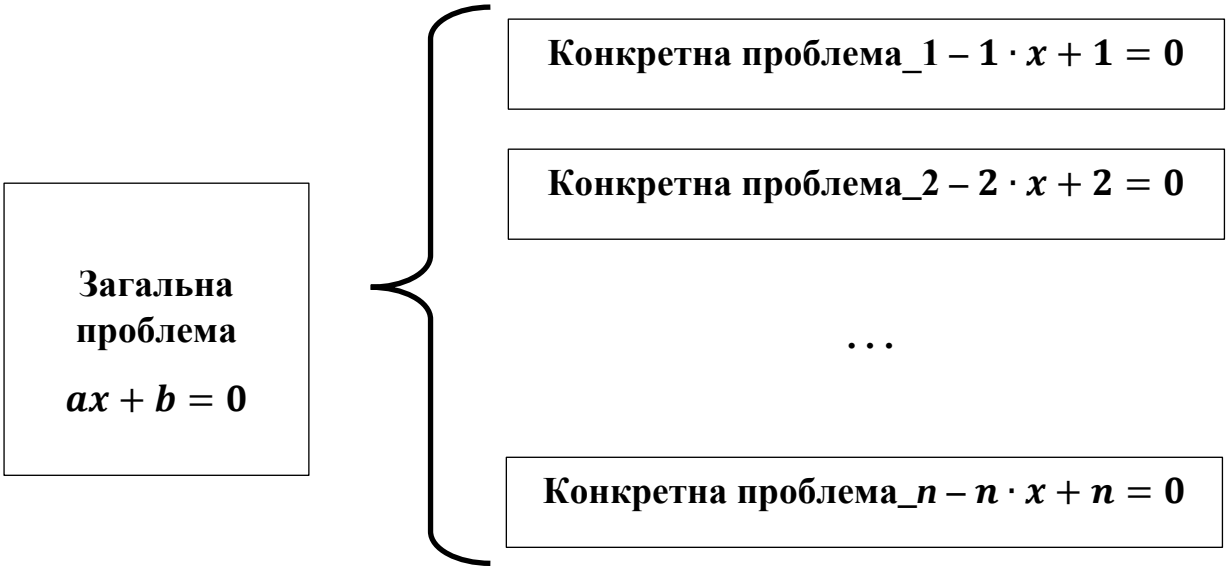


Рис. 2.1. Загальна проблема за Постом (приклад)

Постом був запропонований стандартний набір інструкцій, що містить шість пунктів.

1. Помітити комірку, якщо вона порожня.

2. Стерти мітку, якщо вона є.

3. Переміститься вліво на 1 комірку.

4. Переміститься вправо на 1 комірку.

5. Визначити чи має комірка позначку або ні, і за результатом перейти на одну з двох зазначених інструкцій.

6. Зупинитися.

Ці шість інструкцій дозволяють вирішувати будь-яку проблему за наявності введених Постом понять, до яких відносяться [1]:

•набір інструкцій застосовується до загальної проблеми, якщо для кожної конкретної проблеми не виникає колізій в інструкціях 1 і 2, тобто ніколи програма не стирає мітку в порожній комірці і не встановлює мітку в позначеній комірці;

•набір інструкцій закінчується (за кінцеве кількість інструкцій), якщо виконується інструкція (6);

•набір інструкцій задає фінітний 1 – процес, якщо набір може бути використаним, і закінчується для кожної конкретної проблеми;

•фінітний 1 – процес для загальної проблеми є 1 – рішення, якщо відповідь для кожної конкретної проблеми є правильною (це визначається зовнішньою силою – в сучасних термінах – програмістом).

Приклад роботи машини Поста

Машина Поста працює в унарній системі числення (використовує лише однакові символи).

Існує два числа в унарній системі числення p=**, q=*. Виконати віднімання p-q. Послідовність виконання цієї конкретної проблеми у відповідності з інструкцією табл. 2.1 показана на рис. 2.2.

Таблиця 2.1.

Набір інструкцій для прикладу роботи машини Поста

крок	інструкція	пояснення
1	←	крок ліворуч
2	? 1;3	якщо в комірці пусто, перейти до кроку 1, якщо ні до кроку 3
3	X	видалити мітку
4	→	крок праворуч
5	? 4;6	якщо в комірці пусто, перейти до кроку 4, якщо ні до кроку 6
6	X	видалити мітку
7	→	крок праворуч
8	? 4;6	якщо в комірці пусто, перейти до кроку 9, якщо ні до кроку 1
9	!	кінець

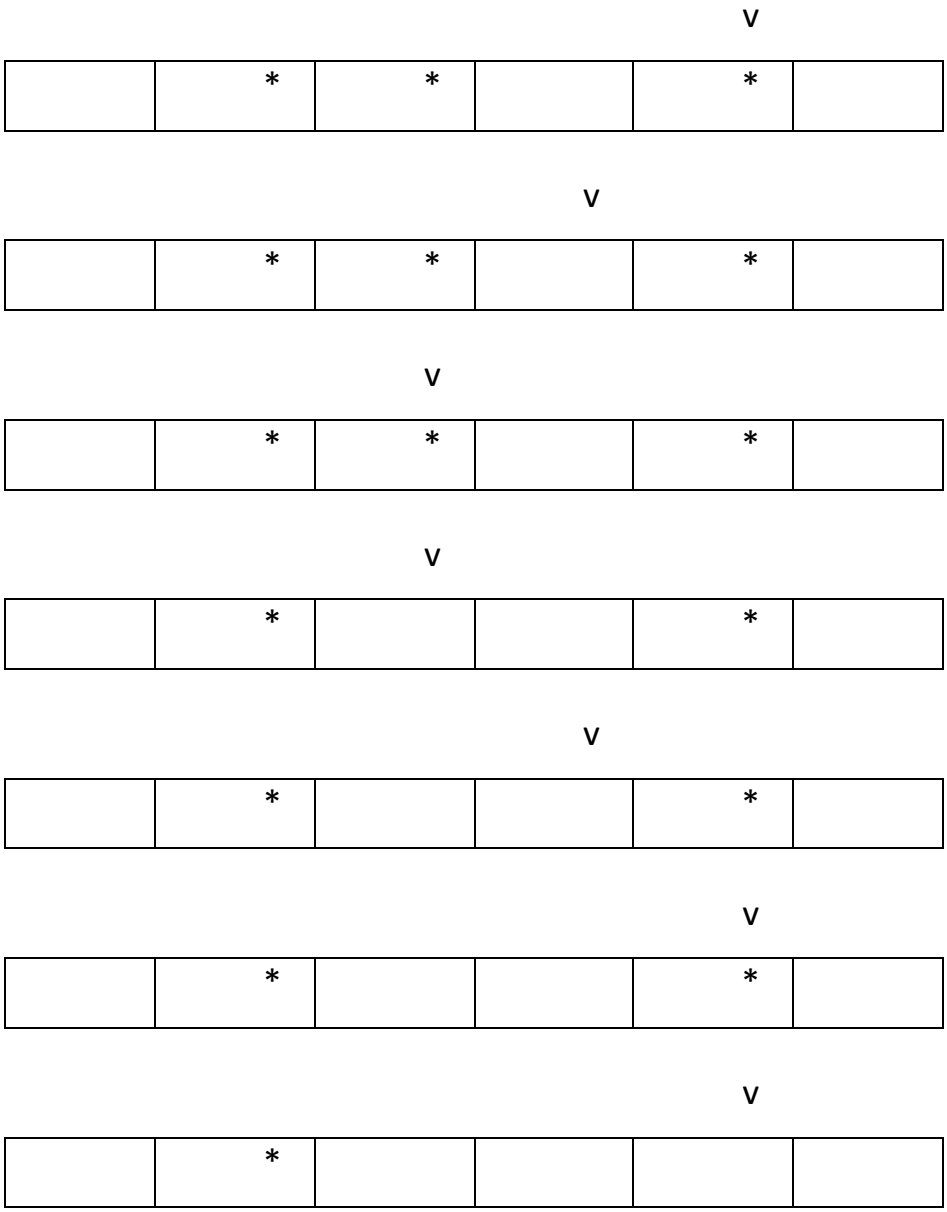


Рис. 2.2. Послідовність виконання набору інструкцій машиною Поста для вирішення конкретної проблеми віднімання

Машина Тюрінга

В машині Тюрінга використовується не унарна система позначок в комірках стрічки, а алфавіт знаків, які задає користувач. Також, в цій машині використовується так званий алфавіт станів. Зміну станів, яку задає користувач в таблиці переходів умовно можна назвати програмою.

Ця таблиця визначає поведінку каретки – перехід з клітинки в клітинку, додавання(видалення) символу алфавіту, зупинку. Машина Тюрінга є моделлю так би мовити «ближчою» до комп'ютера і не обмежується однотипним набором інструкцій, а дозволяє створювати послідовність станів, яку умовно можна назвати програмою.

Приклад роботи машини Тюрінга

Алфавіт символів заданий двома символами – літерами a та b. На стрічці розташована випадкова послідовність поєднання цих символів. Здійснити заміну послідовності на послідовність протилежних літер a→ b та b→ a. Каретка знаходиться ліворуч від послідовності, рис. 2.3.

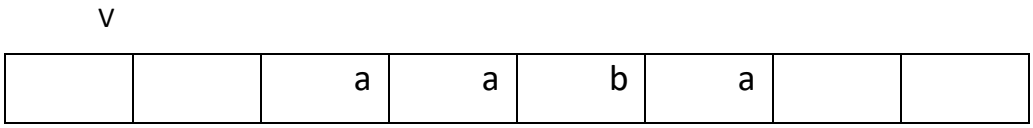


Рис. 2.3. Початковий стан машини Тюрінга прикладу

Таблиця переходів побудована наступним чином, табл. 2.2.

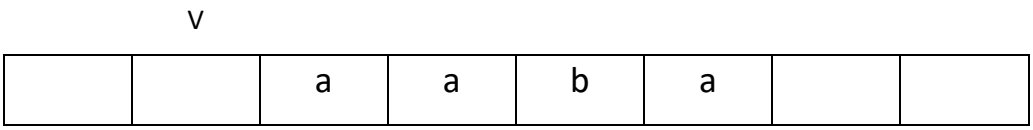
Таблиця 2.2.

Таблиця переходів для прикладу

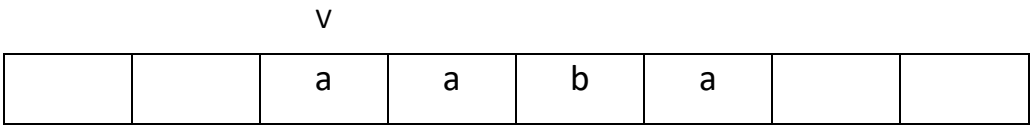
Алфавіт символів	Стан – Q1 пошук символу	Стан – Q2 зміна символу на протилежний
a	перехід до стану Q2	заміна на b
b	перехід до стану Q2	заміна на a
пусто	рух праворуч	закінчення

Послідовність роботи машини за прикладом

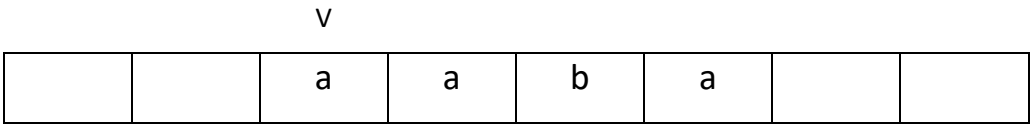
Стан Q1. Каретка над пустою коміркою. Відповідно до табл. 2.2 – рух праворуч



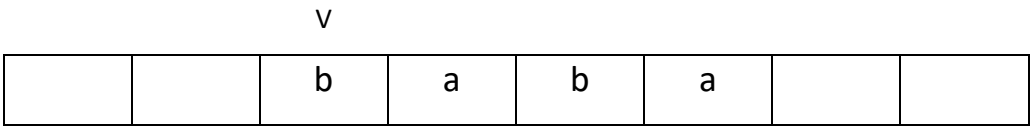
Стан Q1. Знов каретка над пустою коміркою. Відповідно до табл. 2.2 – рух праворуч



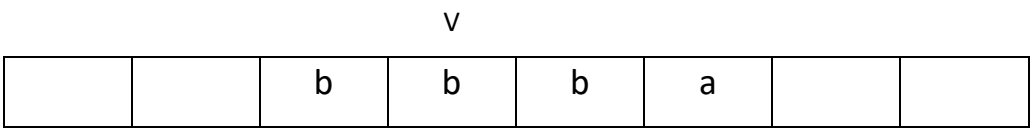
Стан Q1. Каретка над символом a. Відповідно до табл. 2.2 – перехід до стану Q2



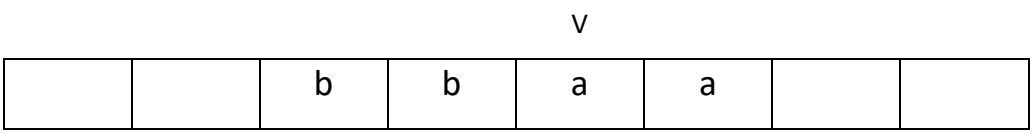
Стан Q2. Заміна символу на b.



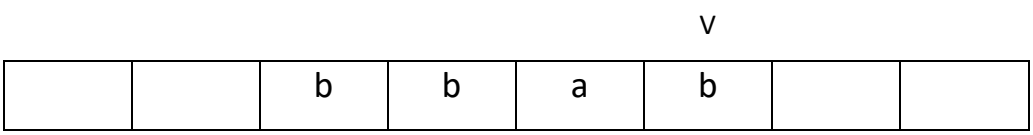
Стан Q2. Заміна символу на b.



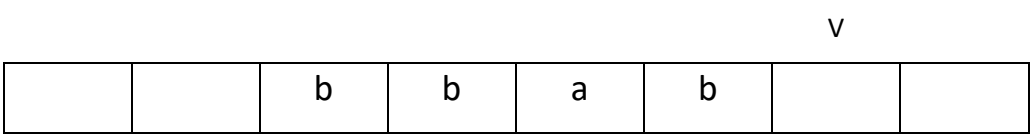
Стан Q2. Заміна символу на a.



Стан Q2. Заміна символу на b.



Стан Q2. Пуста комірка. Закінчення.



Властивості машини Тюрінга як алгоритму

На прикладі машини Тюрінга добре простежуються властивості алгоритмів.

Зрозумілість. Кожен крок вивірений. Тобто, на кожному кроці в комірку пишеться символ з алфавіту, автомат робить один рух і машина Тюрінга переходить в один з визначених станів.

Дискретність. Виконання алгоритмів як свідчить робота машини Тюрінга здійснюється покроково. Перехід до наступного кроку можливий лише після виконання попереднього. Попередній крок визначає те, яким буде наступний.

Детермінованість. Для кожної клітинки описаний лише один варіант дії. Тобто послідовність операцій переходу з комірки в комірку і дії над ними здійснюється не випадковим чином, а відповідно до чітко визначеного правила. Послідовність кроків під час виконання задачі визначена однозначно. Слід сказати, що існує недетермінована машина Тюрінга в якій допускаються варіанти переходу з однієї комірки до інших [1].

Результативність. Результат виконання задачі на машині Тюрінга буде досягнутий за фіксоване число кроків.

Масовість. Машину Тюрінга можна застосовувати для виконання конкретних задач з всіма можливими словами алфавіту. Тобто, кожна машина Тюрінга визначена над усіма допустимими словами з алфавіту.

Вплив машин Поста та Тюрінга на розвиток теорії алгоритмів

Створення машини Поста та машини Тюрінга заклало теоретичні положення теорії алгоритмів. До таких положень зокрема відносять [1].

Формулювання 1

Якщо загальна проблема 1-задана і 1-розв'язана, то, поєднуючи набори інструкцій за завданням проблеми, і її вирішенням отримуємо відповідь за номером проблеми. Це і є в термінах Поста формулюванням 1.

гіпотеза Поста – будь-які ширші (у сенсі алфавіту символів) формулювання набору інструкцій, подання та інтерпретації конкретних проблем зводяться до формулювання 1;

гіпотеза Черча (гіпотеза Черча - Тюрінга), яка формулюється наступним чином – для будь якого алгоритму завжди можна побудувати машину Тюрінга, яка його реалізує.

Таку гіпотезу принципово не можна довести, оскільки поняття «будь який алгоритм» є інтуїтивним поняттям і не може бути об’єктом математичних досліджень. Тим не менше гіпотеза Черча є основною гіпотезою теорії алгоритмів.

теорема про «зупинку» **–** проблема застосовності довільної програми до довільних вхідних даних алгоритмічно нерозв'язна [5].

Формулювання і доведення цієї теореми дозволило виділити групу математичних проблем, які не можуть бути розв’язані алгоритмічно. До них зокрема, відносяться:

- розподіл дев'яток в запису числа π;
- обчислення досконалих чисел;
- десята проблема Гільберта;
- та інші [6].

Контрольні питання

1. *Які основні елементи машини Поста та машини Тюрінга?*
2. *Чи можливо в машині Поста використати іншу систему числення крім унарної?*
3. *Які властивості алгоритмів можна простежити в машині Тюрінга?*
4. *Чому окремі проблеми не можуть бути розв’язані алгоритмічно?*

Розділ 3
ЧАС ВИКОНАННЯ АЛГОРИТМІВ. ТРУДОМІСТКІСТЬ АЛГОРИТМІВ

Головним питанням аналізу алгоритмів є – яку кількість обчислювальних ресурсів комп'ютера потребує алгоритм. Обчислювальні ресурси це пам'ять комп'ютера, швидкодія процесора, швидкість обміну даними та інші характеристики.

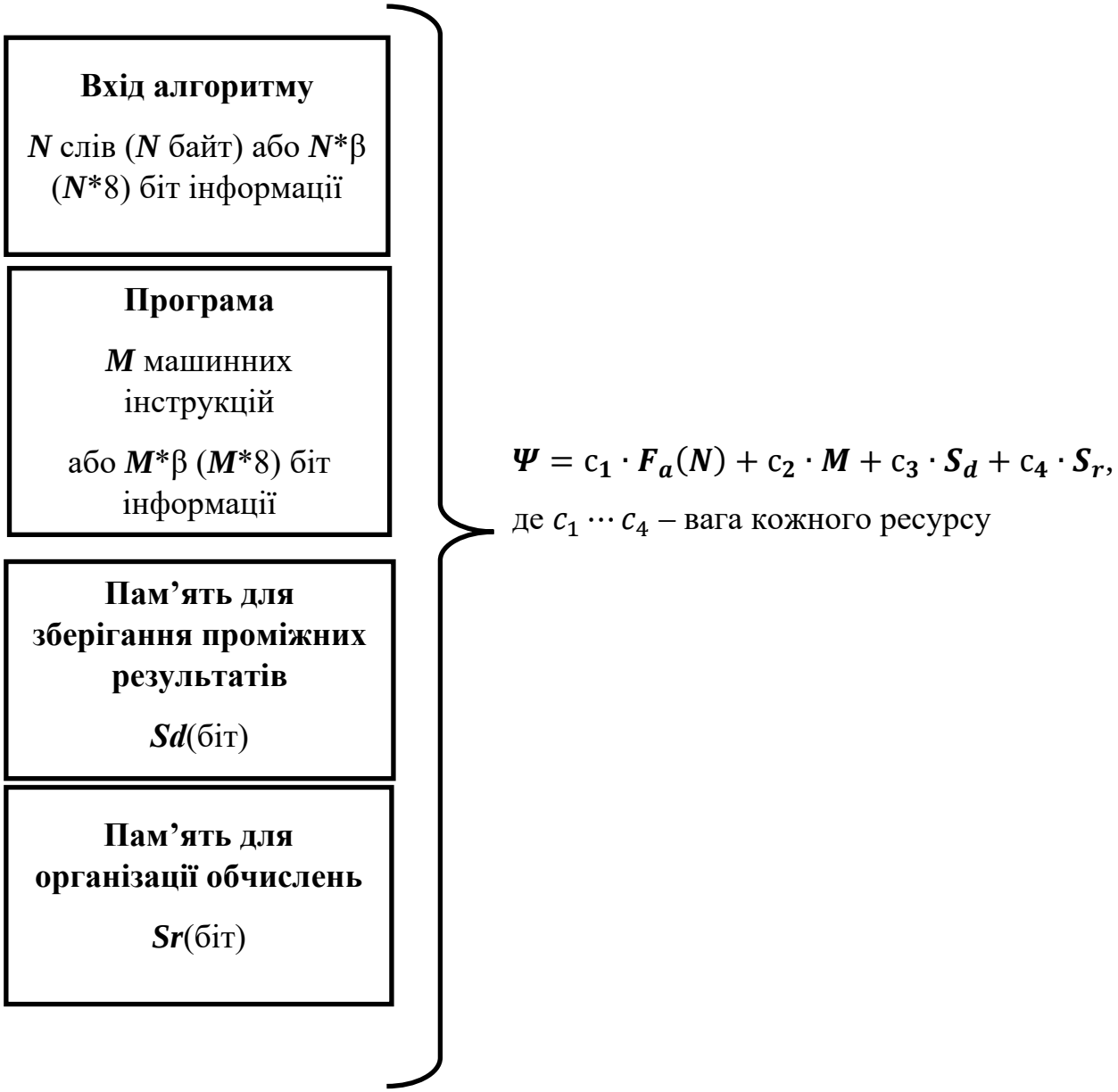


Рис. 3.1. Використання обчислювальних ресурсів для виконання алгоритму

Для області(безлічі) вхідних даних *D*, що можуть вирішуватися алгоритмічно $D \in D_N$ більшість алгоритмів можна проаналізувати за трьома варіантами їх виконання, що прив'язані до кількості слів на вході алгоритму.

Перший варіант. Найбільша кількість операцій, що використовуються алгоритмом *A* для вирішення проблеми розмірністю *N* слів (найгірший випадок на $\boldsymbol{D_N}$). Як правило, велика кількість виконуваних операцій обчислювальними засобами розглядається як негатив, оскільки будь-яка задача повинна вирішуватися якнайшвидше.

$$\boldsymbol{F_A^{\wedge}(N) = max\{F_A(D)\}} \qquad (3.1)$$
$$D \in D_N$$

Другий варіант. Середня або середньостатистична кількість операцій, що здійснюються алгоритмом *A* для вирішення конкретних проблем розмірністю **N** слів

$$\boldsymbol{F(N) = \frac{1}{M_{DN}} \cdot \sum\{F_A(D)\}} \qquad (3.2)$$

Третій варіант. Найкращий випадок – найменша кількість операцій, що здійснюються алгоритмом *A* для вирішення конкретних проблем розмірністю **N** слів

$$\boldsymbol{F_A^v(N) = min\{F_A(D)\}} \qquad (3.3)$$
$$\boldsymbol{D \in D_N}$$

На практиці ці випадки можна оцінити задаючи на вході алгоритму різну кількість вхідних даних. Однак, не існує однозначної залежності між кількістю вхідних даних і кількістю операцій, які виконує алгоритм, тому використовують більш прийнятну класифікацію алгоритмів за якою можна оцінювати як довго вони будуть виконуватися, тобто оцінити їх трудомісткість.

Залежно від впливу вхідних даних на функцію трудомісткості алгоритму існує наступна класифікація, що має практичне значення для аналізу алгоритмів.

Кількісно - залежні за трудомісткістю алгоритми

Це алгоритми, функція трудомісткості яких залежить тільки від розмірності конкретного входу, і не залежить від конкретних значень. Чим більше вхідних даних, тим повільніше виконуватиметься алгоритм.

Функція трудомісткості для кількісно-залежних алгоритмів визначена наступним чином:

$$\boldsymbol{F_A(D) = F_A(|D|) = F_A(N)}, \qquad (3.4)$$

де

D – конкретна проблема;

N – довжина входу алгоритму (кількість слів пам’яті).

Прикладами таких алгоритмів є алгоритми множення матриць, алгоритми операцій з масивами, множення матриці на вектор та інші. Нижче наведений псевдокод множення чисел у двомірному масиві. В цьому коді із збільшенням розмірності масиву пропорційно збільшується кількість операцій множення, і, відповідно, час виконання алгоритму.

```
Mult (Mas, n, m)//Назва програми, масив та змінні
n ← 101      //Присвоєння першій змінній значення
m← 101     // Присвоєння другій змінній значення
  For i ← 1 to n       //Цикл від i=1 до змінної n
    For j ← 1 to m  //Цикл від i=1 до змінної m
      Mas[i][j]=i*j;//Множення змінних в масиві
   end for m      //Закінчення циклу за змінною m
end for n                 //Закінчення циклу за змінною n
return Mas      //Повернення матриці
End                 //Закінчення програми
```

Параметрично - залежні за трудомісткістю алгоритми

В таких алгоритмах розмірність входу майже не впливає на час виконання. Час виконання таких алгоритмів залежить від конкретних значень оброблюваних слів пам'яті.

Функція трудомісткості для параметрично-залежних алгоритмів записується так:

$$\boldsymbol{F_A(D) = F_A(d_1, \cdots d_n) = F_A(P_1, \cdots P_m), m \leq n} \qquad (3.5)$$

де

$\boldsymbol{D}$ – конкретна проблема;

$\boldsymbol{d_1, \cdots d_n}$ **–** множина підпроблем;

$\boldsymbol{P_1, \cdots P_m}$ **–** множина параметрів;

$\boldsymbol{n}$ – кількість підпроблем;

$\boldsymbol{m}$ – кількість параметрів.

Прикладами таких алгоритмів є обчислення функцій за допомогою ряду Тейлора із заданою точністю.

Блок-схема алгоритму розрахунку функції синуса через ряд Тейлора із заданою точністю показана на рис. 3.2.

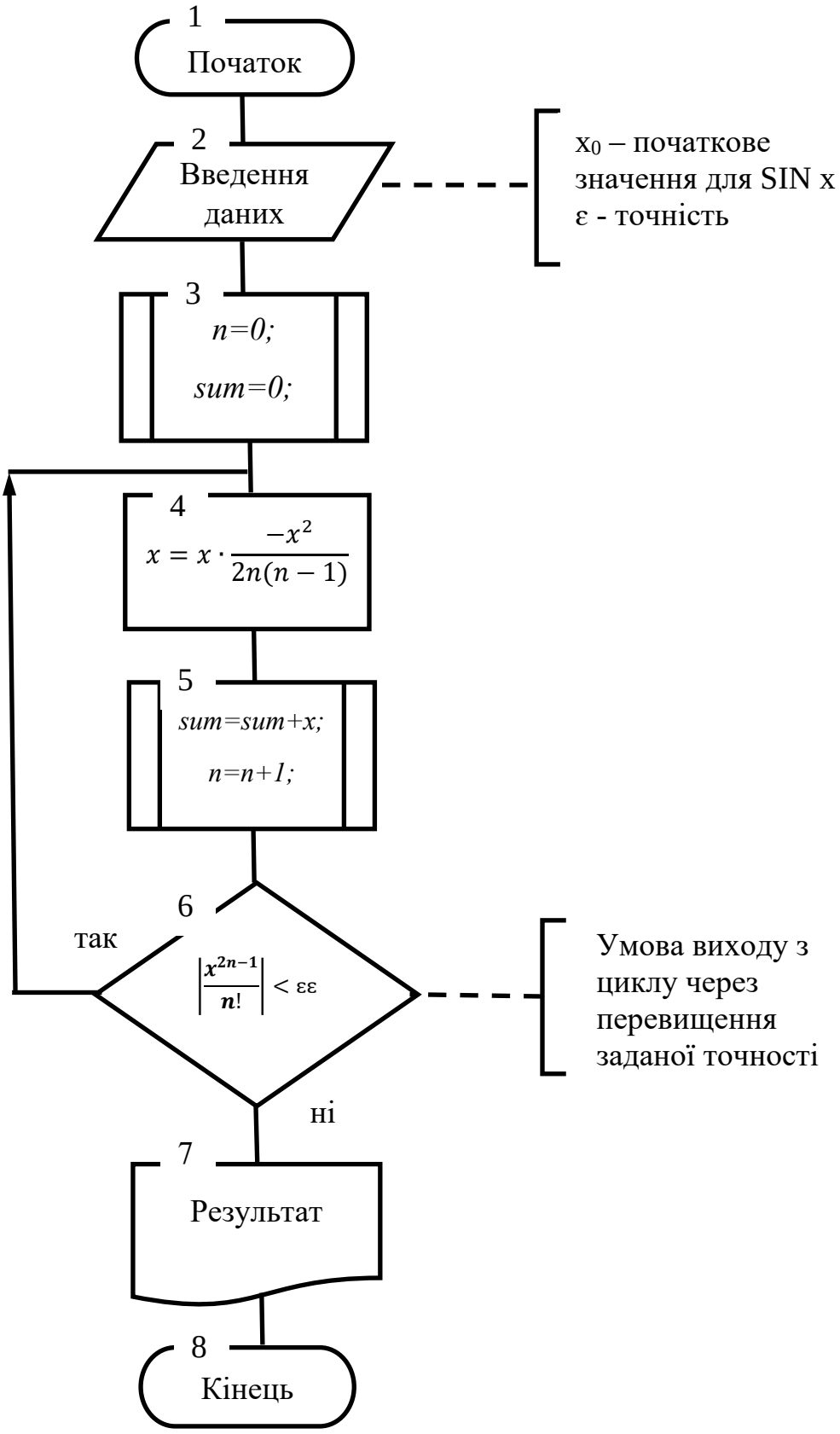


Рис. 3.2. Блок-схема алгоритму розрахунку функції синуса за допомогою ряду Тейлора

До параметрично-залежних відносять **порядково - залежні алгоритми**. В таких алгоритмах час виконання залежить від порядку розташування елементів на вході алгоритму.

Якщо

$$\boldsymbol{F_A}\left(\,{}_{\boldsymbol{i}}\boldsymbol{D_p}\right) \neq \boldsymbol{F_A}\left(\,{}_{\boldsymbol{j}}\boldsymbol{D_p}\right), \qquad (3.6)$$

де

$\boldsymbol{D_p} = \{(\boldsymbol{d_1}, \cdots \boldsymbol{d_n})\}$ – множина всіх підпорядкованих *N* входам в алгоритм підпроблем з існуючої множини $\boldsymbol{d_1}, \cdots \boldsymbol{d_n}$ для яких $\boldsymbol{D_p} = \boldsymbol{n}!$;

${}_{\boldsymbol{i}}\boldsymbol{D_p},\ {}_{\boldsymbol{j}}\boldsymbol{D_p} \in \boldsymbol{D_p}$,

то тоді алгоритм належить до порядково - залежних за трудомісткістю.

До таких алгоритмів відносяться зокрема деякі алгоритми сортування, алгоритм пошуку максимального(мінімального) значення в масиві та інші. В блок-схемі алгоритму пошуку максимального елемента в одномірному масиві, який наведеній в розділі 1, рис. 1.4 можна помітити, що якщо максимальний елемент знаходиться на початку масиву, то умова пошуку цього елемента не дозволятиме виконувати операцію присвоювання в блоці № 6 і код буде виконуватися швидше. Якщо ж цей максимальний елемент знаходитиметься в кінці масиву, а все попередні елементи будуть впорядковані за зростанням, то присвоювання буде відбуватися на кожному кроці циклу, що суттєво сповільнить виконання коду.

Кількісно - параметричні за трудомісткістю алгоритми

Значна кількість алгоритмів, які використовуються на практиці є кількісно-параметричними. В цих алгоритмах час виконання залежить як від розмірності входу так і від конкретних значень оброблюваних слів пам’яті.

Функція трудомісткості для кількісно-параметрично алгоритмів визначена наступним чином:

$$\boldsymbol{F_A(D) = F_A(\|D\|, d_1, \cdots d_n) = F_A(N, P_1, \cdots P_m)} \qquad (3.7)$$

де

$\boldsymbol{D}$ – конкретна проблема;

$\boldsymbol{N}$ – довжина входу алгоритму (кількість слів пам’яті);

$\boldsymbol{d_1, \cdots d_n}$ – множина підпроблем;

$\boldsymbol{P_1, \cdots P_m}$ – множина параметрів;

$\boldsymbol{n}$ – кількість підпроблем;

$\boldsymbol{m}$ – кількість параметрів.

Розглянемо приклад алгоритму сортування включенням. На рис. 3.3 показний приклад сортування однакової за кількістю послідовності чисел за цим алгоритмом у двох випадках з різним розташуванням вхідного масиву. Видно, що у випадку b) кількість кроків більша. Отже і час виконання буде тривалішим. Якщо кількість елементів масиву збільшувати час виконання алгоритму також зростатиме. Тобто в цьому алгоритмі сортування є подвійна залежність часу виконання, як від порядку розташування елементів, так і від їх кількості.

4	порівняння
5	
2	
1	

4	порівняння
5	
2	
1	

2	переміщення
4	
5	
1	

2	порівняння
4	
5	
1	

1	переміщення
2	
4	
5	

5	порівняння
4	
2	
1	

4	переміщення
5	
2	
1	

4	порівняння
5	
2	
1	

2	переміщення
4	
5	
1	

2	порівняння
4	
5	
1	

1	переміщення
2	
4	
5	

a) b)

Рис. 3.3. Приклад покрокового сортування включенням

Слід зазначити, що в алгоритмі сортування включенням максимальний час виконання буде у випадку, коли необхідно відсортувати масив в якому елементи розташовані у зворотному порядку.

Контрольні питання

1. *Чи може один і той же алгоритм бути кількісно-параметричним та порядково залежним?*
2. *Чим відрізняється параметрично залежний алгоритм від порядково залежного?*
3. *Які ви знаєте кількісно-залежні за трудомісткістю алгоритми?*
4. *Чи можливо перетворити параметричний за трудомісткістю алгоритм у кількісно-залежний?*

Розділ 4
АСИМПТОТИЧНИЙ АНАЛІЗ ФУНКЦІЙ В ТЕОРІЇ АЛГОРИТМІВ

Для оцінювання часу, необхідного для виконання певного алгоритму використовують такий показник як часова складність алгоритму. Часова складність певного алгоритму свідчить про те, скільки часу необхідно для виконання цього алгоритму. Час виконання алгоритму можна підрахувати шляхом додавання кількості елементарних операцій, які виконуються алгоритмом. При цьому вважається, що кожна елементарна операція виконується за фіксований проміжок часу. Таким чином загальний час виконання алгоритму можна записати формулою:

$$T_e = \sum_{i=1}^{N} t_i, \qquad (4.1)$$

де

T_e – загальний час виконання алгоритму;

t_i – час виконання і-ї елементарної операції;

N – загальна кількість елементарних операцій які виконуються алгоритмом.

Однак, підрахунок часу за такою формулою можна виконувати на конкретному комп’ютері для конкретних умов певної задачі. Якщо ж розглядати алгоритм у якості універсального засобу вирішення задачі або певного класу задач завжди наштовхуються на певні труднощі. Ці труднощі полягають у невизначеності загальної кількості операцій для різної кількості вхідних даних одного і того ж алгоритму, у різних технічних можливостях комп’ютерів щодо обробки даних тощо.

Для того, щоб оцінити настільки та чи інша задача або клас задач може бути придатним для її вирішення на комп'ютері використовують підхід, що ґрунтується на аналізі вхідних даних, який був запозичений з математики.

Згідно з цим підходом вважається, що час виконання алгоритму є функцією від кількості вхідних елементів. Тобто, $T_{\text{в}} = f(n)$, де n – кількість елементів на вході алгоритму.

Відповідно до цього за часову складністю алгоритмів класифікують за трьома ознаками, пов'язаними із кількістю вхідних даних – n.

1. Алгоритми, час виконання яких обмежений певними часовими рамками. У вигляді формули це твердження записується наступним чином

$$\boldsymbol{c_1 g(n) \leq f(n) \leq c_2 g(n)}, \qquad (4.2)$$

де

c_1, c_2 – числові константи;

$f(n)$ – функція складності алгоритму;

$g(n)$ – деяка функція.

Приклад.

Часова складність алгоритму дорівнює – $0{,}5 \cdot n^2 - 3 \cdot n$. Існування в цьому виразі n^2 наводить на думку, що цей алгоритм не може виконуватися швидше за час ніж алгоритм із часовою складністю n^2 (бо n^2 вимагає менше математичних обчислень). Перевіримо це і визначимо в яких межах будуть виконуватися алгоритми з такою часовою складністю.

Задамо $g(n) = n^2$. Тоді, нерівність (4.2) матиме наступний вигляд

$$c_1 n^2 \leq 0{,}5 \cdot n^2 - 3 \cdot n \leq c_2 n^2. \qquad (4.3)$$

Якщо розділити всі частини нерівності на n^2 то отримаємо такий вираз

$$c_1 \leq\ 0{,}5 - \frac{3}{n} \leq\ c_2. \qquad (4.4)$$

Тепер треба обрати значення c_1, c_2 та мінімальне значення n (його називають n_0) при яких нерівність буде виконуватися. Неважко здогадатися, що при $c_2 = 0{,}5$ права частина нерівності буде виконуватися для всіх n (навіть нічого не треба розраховувати), ліва частина нерівності виконується для всіх значень c_1 менших або рівних за $0{,}5 - \frac{3}{n}$. Тобто при $n_0 = 7$ $c_1 = \frac{1}{14} = 0{,}01429$, при $n_0 = 8$ $c_1 = \frac{1}{8} = 0{,}125$, при $n_0 = 9$ $c_1 = \frac{1}{6} = 0{,}1(6)$ і т.д.

В табличному вигляді фрагмент множини рішень для нерівності (4) буде виглядати наступним чином, табл. 4.1

Таблиця 4.1.

Розраховані значення для наведеного прикладу(фрагмент)

c_1	n_0	$0.5-3/n_0$	c_2
-2,50	1	-2,50	0,5
-1,00	2	-1,00	0,5
-0,50	3	-0,50	0,5
-0,25	4	-0,25	0,5
-0,10	5	-0,10	0,5
0,00	6	0,00	0,5
0,07	7	0,07	0,5
0,13	8	0,13	0,5
0,17	9	0,17	0,5
0,20	10	0,20	0,5
0,23	11	0,23	0,5
0,25	12	0,25	0,5
0,27	13	0,27	0,5
0,29	14	0,29	0,5
0,30	15	0,30	0,5
0,31	16	0,31	0,5
0,32	17	0,32	0,5
0,33	18	0,33	0,5

В графічному вигляді множина рішень для заданої часової складності алгоритму виглядає наступним чином, рис. 4.1. На цьому рисунку криві що відповідають c_1 та $0{,}5 - \frac{3}{n}$ співпадають.

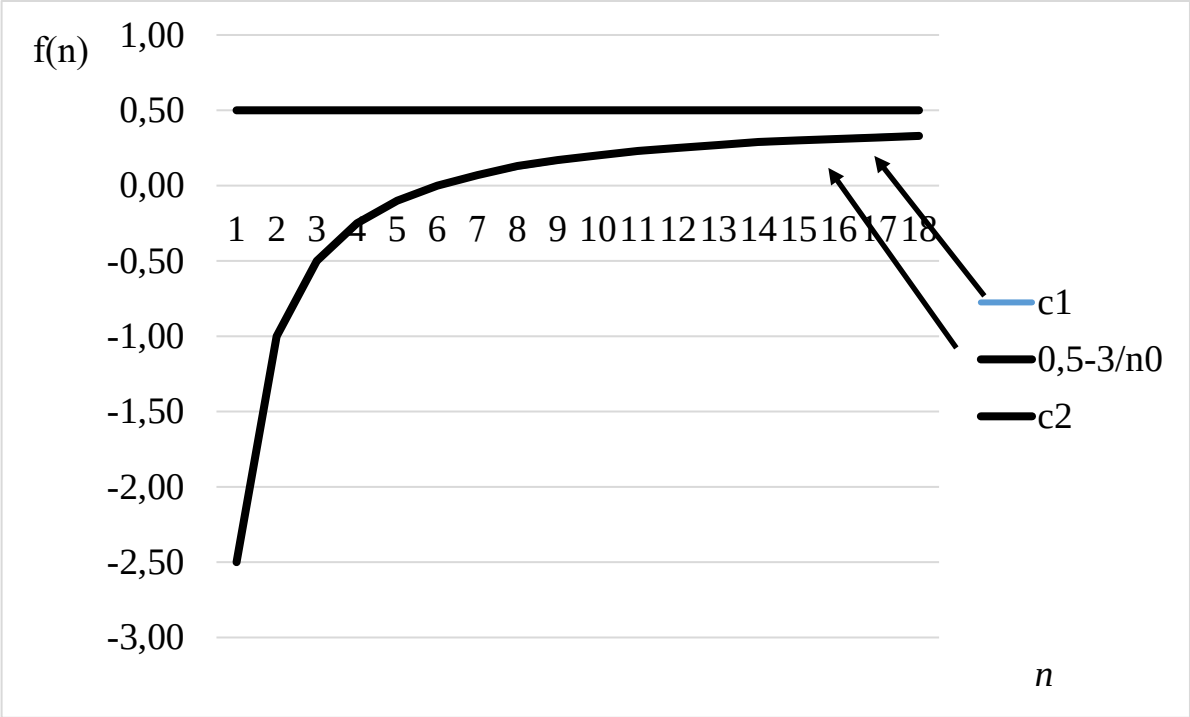


Рис. 4.1. Множина рішень для часової складності алгоритму наведеного прикладу

Отже можна стверджувати, що для часової складності $0{,}5 \cdot n^2 - 3 \cdot n$ всі алгоритми будуть виконуватися з асимптотичною точністю n^2(значення від'ємної частини $3 \cdot n$ ігнорується).

Такий підхід до оцінки складності позначається відповідним спеціальним записом(нотацією) і називається тета нотацією або тета оцінкою і позначається грецькою літерою «тета» – Θ. Для наведеного прикладу маємо $0{,}5 \cdot n^2 - 3 \cdot n \in \Theta(n^2)$. Аналіз часової складності за допомогою графіків, подібних до графіків рис. 1 називають асимптотичним аналізом. Звичайно графік асимптотичного аналізу для Θ-нотації зображають в узагальненому вигляді, рис. 4.2.

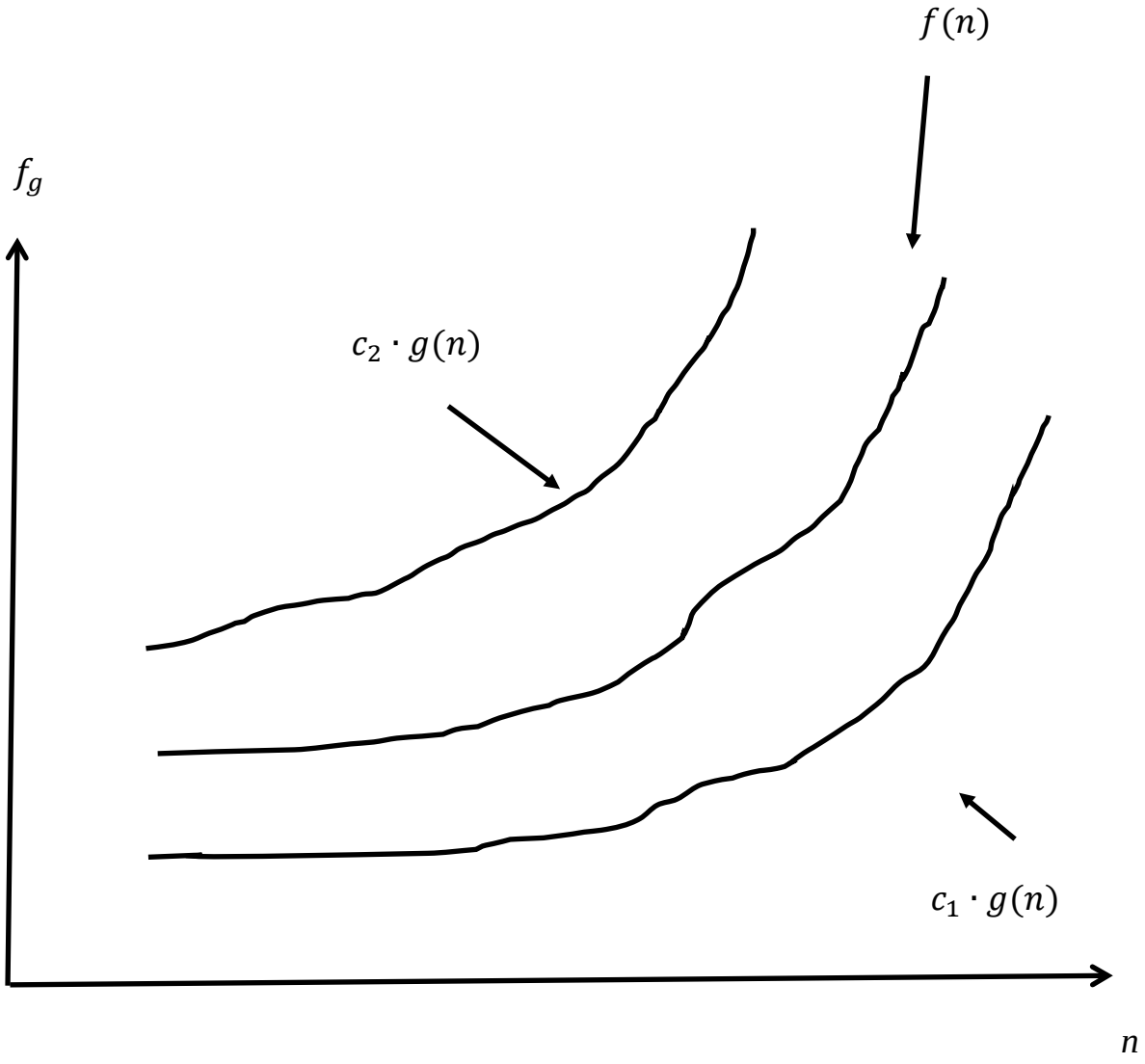


Рис. 4.2. Узагальнений вигляд Θ-нотації для f(n), коли n→∞.

Ось деякі із виразів, що записуються нотацією тета.

$5+2/n \in \Theta$ (1),

$n^6 + 3n \in \Theta(n^6)$,

$2^n + 12 \in \Theta(2^n)$, (4.5)

$3^n + 2^n \in \Theta(3^n)$,

$n^n + n \in \Theta(n^n)$.

Як видно із наведених виразів в нотаціях значення коефіцієнтів, сталих величин та значень *n* які практично не впливають на час виконання алгоритмів та ігноруються.

Подібний асимптотичний аналіз за допомогою графіків здійснюють і для інших класів часової складності алгоритмів.

Приклад конкретного програмного коду дозволяє зрозуміти сутність нотації тета. Щоб відсортувати ряд чисел 0,1,2,3,4 в зворотному порядку 4,3,2,1,0 за допомогою алгоритму сортування включенням використаємо фрагмент псевдокоду та коду мовою С++.

Псевдокод

```
//цикл за довжиною масиву arr
for j←2 length[arr]
     // включення елементу j у відсортовану послідовність a𝑟𝑟[1 … j − 1]
     key←arr[j]
    i←j-1
   while i>=0 and arr[i]>key
       arr[i+1]←arr[i]
       i←i-1
   end while
   arr[i+1]←key
end for
```

Код мовою С++

```
void InsertSort(int arr[], int n)
{
   int key, i;
   for (int k = 1; k < n; k++)          //i=1,2,3,4,5
  {
     key = arr[k];                      //i=1,2.3.4
     i = k - 1;                         //i=1,2,3,4
     while ((i >= 0)&&(arr[i]>key)) //1+2+3+4+…+n(прохід - n-1)
     {
        arr[i + 1] = arr[i];             //1+2+3+..+(n-1) - прохід – 1+(n-1)
        i = i - 1;                        //1+2+3+..+(n-1) - прохід – 1+(n-1)
      }
```

```
        arr[i + 1] = key;                    //i=1,2.3.4
    }
}
```

В цих кодах показано, що кількість проходів всередині умовного циклу while дорівнює

$$K_{while} = \frac{2 \cdot \left(1 + (n-1)\right)}{2} \cdot (n-1) \qquad (4.6)$$

Для циклу for кількість проходів дорівнюватиме

$$K_{for} = 3 \cdot (n-1) \qquad (4.7)$$

Сума цих двох складових дає вираз

$$K = n^2 + 2 \cdot n - 3 \qquad (4.8)$$

В цьому виразі для $n \to \infty$ суттєвою для обчислень є тільки перша складова. Це можна перевірити задавши n великим числом. Наприклад для n=1000000 значення у вираз розподіляться так:

$$K = 10^{12} + 2 \cdot 10^6 - 3 \text{ де } 10^{12} \gg (2 \cdot 10^6 - 3)$$

Отже, вираз $2 \cdot 10^6 - 3$ для обчислення практично не має ніякого значення бо на його обчислення витрачається тільки 1/1000000 часу, а для $n \to \infty$ ця частина взагалі ніби зникає. Тета нотація для наведеного прикладу дорівнює – $\Theta(n^2)$.

Для випадку сортування послідовності того самого ряду чисел 0,1,2,3,4 коли він вже відсортований не потребуватиме використання умовного циклу while. Для цього випадку використовується лише формула (4.7) і нотація буде визначена як $\Theta(n)$. Отже тета нотація визначає функцію з точністю до постійного множника.

2. В більшості випадків на практиці найбільшу цікавість становить саме найгірший випадок часової складності, тобто алгоритми, час виконання яких обмежений максимальним значенням. Алгоритми позначені цією нотацією не можуть виконуватися швидше певної функції. У вигляді формули вона записується наступним чином.

$$0 \leq f(n) \leq g(n) \text{ для } \forall n > n_0, \qquad (4.9)$$

де

$f(n)$ – функція складності алгоритму;

$g(n)$ – деяка функція;

n_0 – певна мінімальна кількість елементів на вході алгоритму.

Ця нотація називається О-нотацією (Big O) або нотацією Едмунда Ландау (на честь німецького математика XIX - XX сторіччя). Вона використовується набагато частіше ніж інші нотації.

У попередньому прикладі можна помітити, що один і той же алгоритм має дві тета нотації. Одна – $\Theta(n^2)$ для випадку коли вхідні дані максимально невпорядковані, інша – $\Theta(n)$ коли вони максимально впорядковані. Очевидно існують інші в залежності від порядку розташування елементів в сортованому масиві. На практиці такими нотаціями користуватися можна лише у специфічних випадках. Як правило, розробників цікавить найгірший варіант який за яким можна сказати як «працюватиме» алгоритм якщо вхідні дані організовані найбільш «незручним» чином. В такому випадку застосовується О-нотація. Отже, О-нотація показує верхню межу залежності між вхідними параметрами функції і кількістю операцій, які виконує процесор.

Якщо час роботи алгоритму не залежить від обсягу вхідних даних, то його часову складність позначають як O (1).

Наприклад, два різних коди, написані мовою Python мають одну і туж нотацію – O(1), незалежно від розмірності самого коду. Це просто набір постійної кількості операцій. Асимптотично залежність між часом і кількістю виконуваних операцій завжди буде прямою лінією паралельною ординаті, рис 4.3.

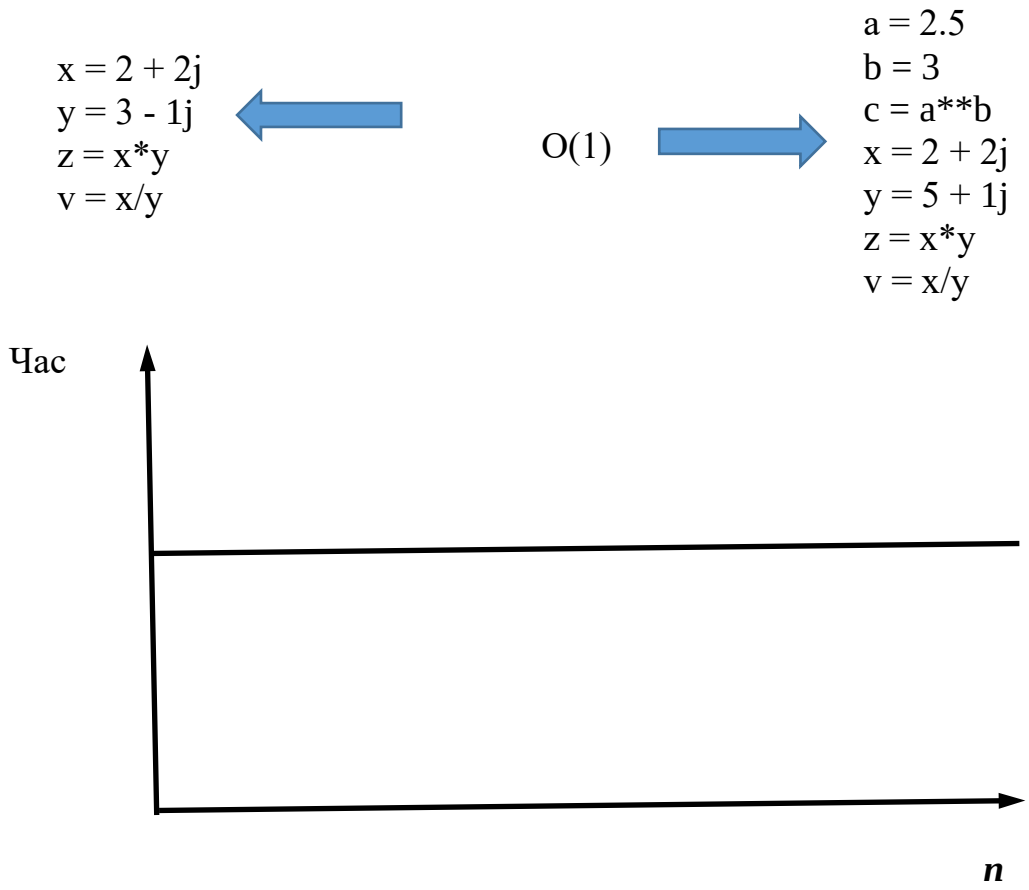


Рис. 4.3. Асимптотична складність O(1)

Лінійна складність O(n): час роботи алгоритму лінійно зростає зі збільшенням оброблюваних елементів. Якщо розглянути коди двох алгоритмів – рекурсивного і ітеративного (мова C++) то можна помітити, що не дивлячись на різну розмірність вони виконують одну і туж операцію – підсумовування ряду чисел.

```
int sum(int n)
{
  if (n==1) return 1;
  return n + sum(n-1);
}
```

O(*n*)

```
int SumSequence(int n)
{
int sum = 0;
for (int i=0; i<n; i++)
{
  sum += pSum(i, i+1);
}
  return sum;
}
  int pSum(int a, int b)
{
  return a + b;
}
```

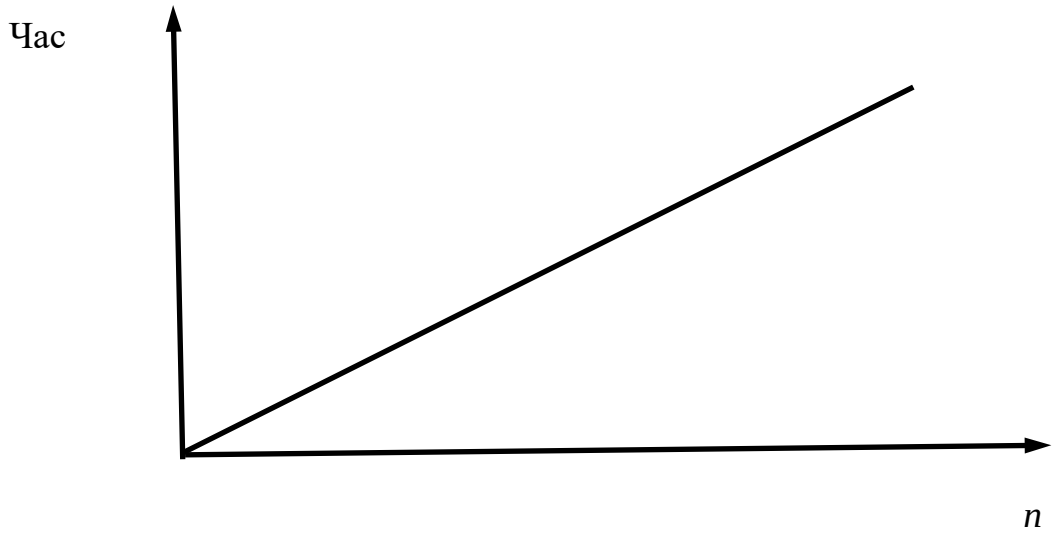


Рис. 4.4. Асимптотична складність O(n)

Асимптотична складність O(log*n*): час роботи алгоритму зростає пропорційно логарифму кількості оброблюваних елементів. Приклад — бінарний пошук у відсортованому масиві. Розглянемо масив із 16 елементів. В найгіршому випадку елемент, який треба знайти буде знаходитися на початку або у кінці списку. Відповідно до алгоритму бінарного пошуку посередині масиву здійснюється пошук елемента. На тій частині масиву де елемент менший (або більший) знову посередині здійснюється пошук елемента.

Ця процедура повторюється до знайдення необхідного елементу. Для відсортованого масиву довжиною в 16 елементів такий пошук вимагатиме 4 поділи, табл. 4.2.

Таблиця 4.2.

Послідовність поділу масиву алгоритму бінарного пошуку

16	ділення на 2	2^4
8	ділення на 2	2^3
4	ділення на 2	2^2
2	ділення на 2	2^1
1	останній пошук	2^0

Як видно із табл. 4.2 максимальна кількість поділів масиву складає – 4. Число 4 від кількості елементів є $\log_2 16$ (або log16 бо константи в нотаціях не записуються). Отже, О-нотація такого алгоритму запишеться як O(log*n*). Відповідно графік залежності часу від *n* буде логарифмічним.

Квадратична складність O(n^2): час роботи алгоритму зростає пропорційно квадрату кількості оброблюваних елементів. Приклад — алгоритм сортування вставками. Слід зазначити, що в програмних кодах з подвійним або потрійним циклом якщо значення n подано як константа нотація не змінює свою складність. Нижче показаний фрагмент коду алгоритму сортування обміном (бульбашковий алгоритм). В цьому подвійний цикл, але О-нотація цього алгоритму O(n), а не O(n^2).

Сортування вставками(фрагмент)

```
int *bubbleSort(int *array, int n, SortOrder sortOrder)
{
   bool swappedFlag = false;
   for (int i = 1; i < 1000; i++)//Цей цикл не впливає на нотацію.
   {
      swappedFlag = false;
      for (int j = 0; j < n - i; j++)
```

```
        {
            if (!isSorted(array[j], array[j + 1], sortOrder))
            {
                swapElements(array[j], array[j + 1]);
                swappedFlag = true;
            }
        }
        if (!swappedFlag)
        {
            break;
        }
    }
    return array;
}
```

Експоненціальна складність $O(x^n)$: час роботи алгоритму зростає пропорційно експоненті кількості оброблюваних елементів; приклади — т. зв. задача комівояжера, семантико-залежні задачі обробки природномовного тексту.

Найчастіше коли говорять про часову складність алгоритмів використовують саме О-нотацію, наводячи таблиці і графіки асимптотичної складності. На рис. 4.5 показаний варіант саме такого сімейства графіків.

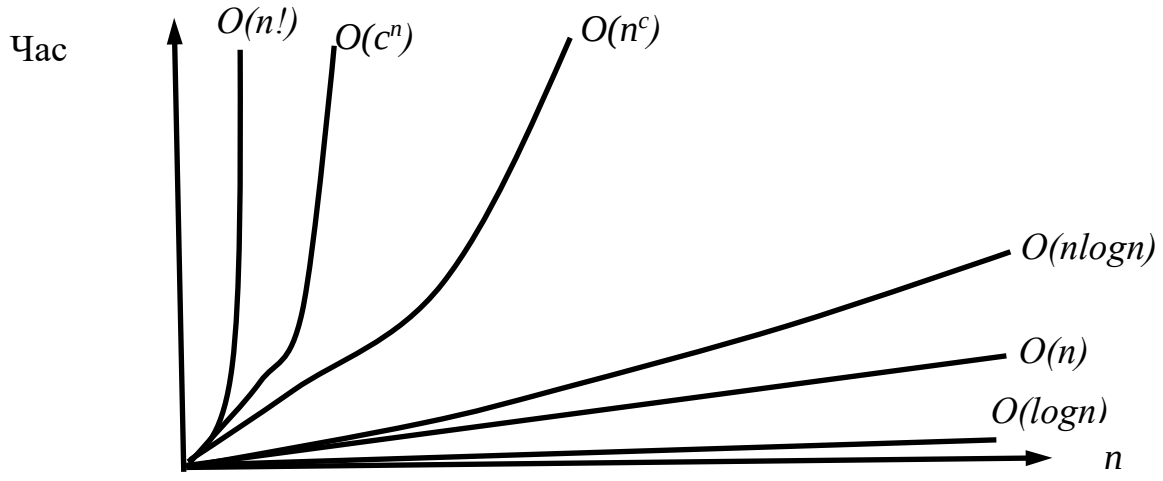


Рис. 4.5. Деякі графіки асимптотичної складності за О-нотацією

3. Нотація омега – **Ω**-нотація на відміну від О-нотації свідчить про найкращий випадок виконання алгоритму. Наприклад для розглянутого алгоритму сортування включенням це буде вже знайомий випадок коли на вхід подається вже відсортований масив. Для нього справедливе співвідношення

$$0 \leq c \cdot g(n) \leq f(n) \qquad (4.10)$$

за умови $\exists c > 0, n_0 > 0,$

де

$f(n)$ – функція складності алгоритму;

$g(n)$ – деяка функція;

n_0 – певна мінімальна кількість елементів на вході алгоритму;

c – певна константа.

В математиці існують ще нотації o(маленьке) та w-нотація [7]. Однак, в інформатиці та алгоритмізації вони розповсюдження не отримали.

Контрольні питання

1. В чому полягає сутність асимптотичного аналізу функцій в теорії алгоритмів?

2. Чи може один і той же алгоритм мати дві нотації, наприклад тета нотацію та нотацію Ландау?

3. Про що свідчить асимптотична складність О-велике?

4. Яка нотація найбільш часто використовується на практиці і чому?

Розділ 5
ЕВРИСТИЧНІ АЛГОРИТМИ ТА ЇХ ВЛАСТИВОСТІ

Евристичні або пошукові алгоритми використовуються задачах математичне вирішення яких є ускладненим через великий обсяг обчислень в алгоритмах, що ґрунтуються на детермінованих методах або методах, які не містять випадкових величин. Існує велика кількість подібних задач в різних галузях. Розповсюдженим прикладом такої задачі є задача комівояжера (Travelling Salesman Problem - TSP), що була сформульована в XIX сторіччі Вільямом Гамільтоном.

Сутність задачі полягає у знаходженні найкоротшого маршруту між точками(містами), що з'єднані одне з одним шляхами різної довжини. Необхідно пройти всі міста по одному разу і повернутися у вихідне місто. Приклад одного з найпростіших варіантів такої задачі показаний на рис. 5.1.

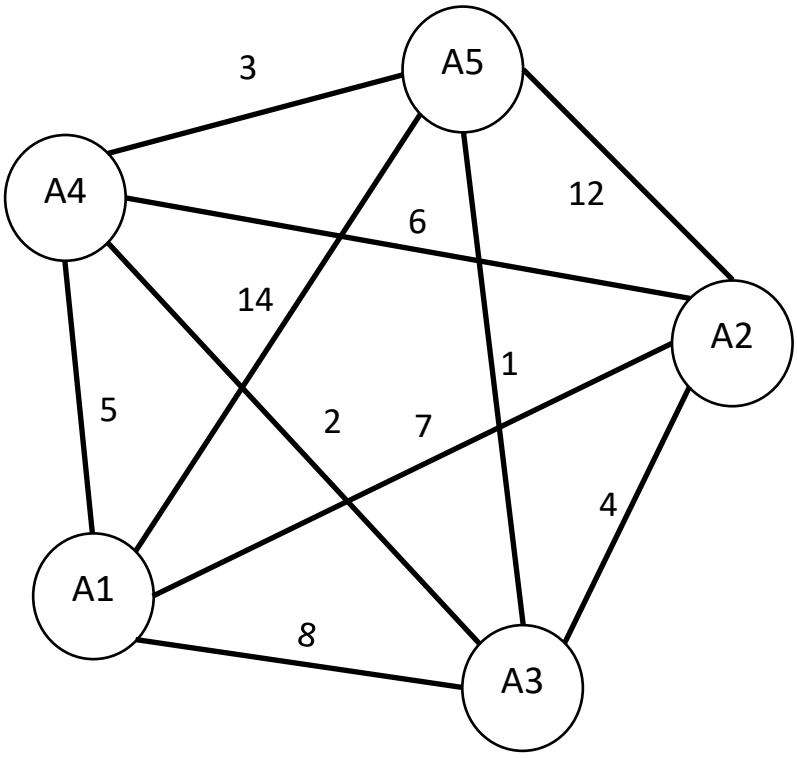


Рис. 5.1. Приклад задачі комівояжера із п'яти міст

Зазначений приклад можна подати у вигляді квадратної зворотносиметричної матриці, табл. 5.1. Це суттєво спрощує формалізацію і обчислення маршрутів.

Таблиця 5.1

Приклад зворотносиметричної матриці для задачі комівояжера

	A1	A2	A3	A4	A5
A1	0	7	8	5	14
A2	7	0	4	6	12
A3	8	4	0	2	1
A4	5	6	2	0	3
A5	14	12	1	3	0

Задача комівояжера відноситься до комбінаторних задач і має точні рішення, однак, методи які реалізують такі рішення є неефективними [8]. Наприклад, кількість варіантів маршрутів які необхідно обчислити послідовно перебираючи їх в найгіршому випадку дорівнює – $n!$, де n – кількість міст. Тобто для цієї задачі – O(n!). Такий метод повного перебору (brute force) не дозволяє отримати необхідний результат за прийнятний час навіть для найсучасніших комп'ютерів. Наприклад, для відносно невеликого числа із 100 міст кількість варіантів маршрутів приблизно дорівнює – $9{,}332621544394 \cdot 10^{157}$. Інший точний метод – метод гілок і меж так [16ож є досить повільним, тому, часто використовують евристичні алгоритми які дозволяють отримувати наближене до мінімального рішення або інколи навіть точне рішення за прийнятний час.

Евристичні алгоритми здійснюють пошук відповіді для ряду математичних задач шляхом моделювання з використанням випадкових чисел. Тобто, результат який отримується знаходять не шляхом обчислення строго визначеної детермінованої послідовності дій, а шляхом розрахунків пошуку за певною стохастичною моделлю.

В таких методах використовують генерування випадкових чисел для створення умов з метою якомога ефективнішого пошуку. Подібне моделювання дозволяє творчо підходити до вирішення таких задач.

Найбільш простим і не досить продуктивним методом пошуку є алгоритм Монте - Карло. Алгоритм з використанням методу Монте - Карло ґрунтується на перебиранні варіантів маршрутів з використанням генерування випадкових чисел комп'ютерними засобами. Кожен маршрут формується шляхом створення послідовності міст через який він проходить згенерованих випадковим чином. В кожній такій послідовності розраховується загальна довжина маршруту, порівнюється з довжиною попереднього маршруту і якщо вона менша, то залишається у якості еталону для порівняння з наступним сформованим маршрутом. Таким чином через визначену користувачем кількість згенерованих випадковим чином маршрутів відфільтровується маршрут мінімальної довжини, рис. 5.2.

Для швидкодіючих комп'ютерів таке перебирання маршрутів не є складним завданням і виконується за прийнятний час. Звісно отримати найкоротший маршрут для великої кількості міст використовуючи цей алгоритм досить складно, оскільки ймовірність отримання найкоротшого маршруту за допомогою такого алгоритму надзвичайно мала. Однак, цей алгоритм можна застосовувати в поєднанні з іншими алгоритмами. Наприклад, для кожної ітерації надавати «непродуктивним» ділянкам певного значення і уникати їх на наступних ітераціях, використовуючи спеціальні формули. В такому разі ефективність пошуку суттєво зростає і ймовірність отримання маршруту з довжиною близькою до мінімальної підвищується.

На практиці задача комівояжера в математичній постановці використовується нечасто. Частіше вона є основою для побудови більш складних розрахункових моделей.

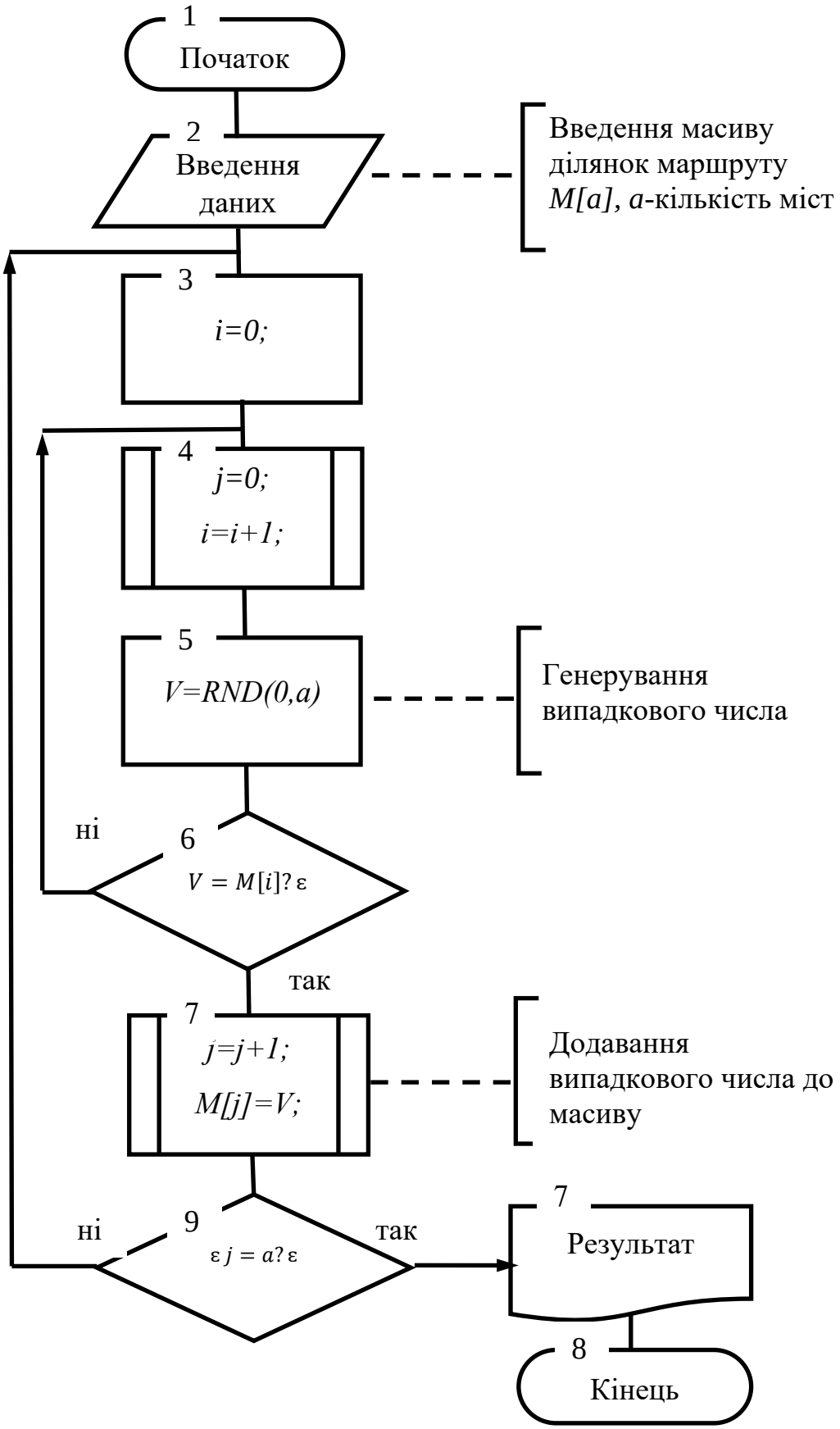


Рис. 5.2. Блок - схема алгоритму Монте - Карло для визначення довжини одного маршруту

Більш ефективним для вирішення задачі комівояжера є «жадібний» алгоритм або для задачі комівояжера його ще називають алгоритмом найближчого сусіда. Сутність «жадібного» алгоритму полягає у формуванні маршруту шляхом вибору найкоротшого відрізку шляху на кожному кроці вибору. Результат роботи такого алгоритму показаний жирними лініями на рис. 5.3.

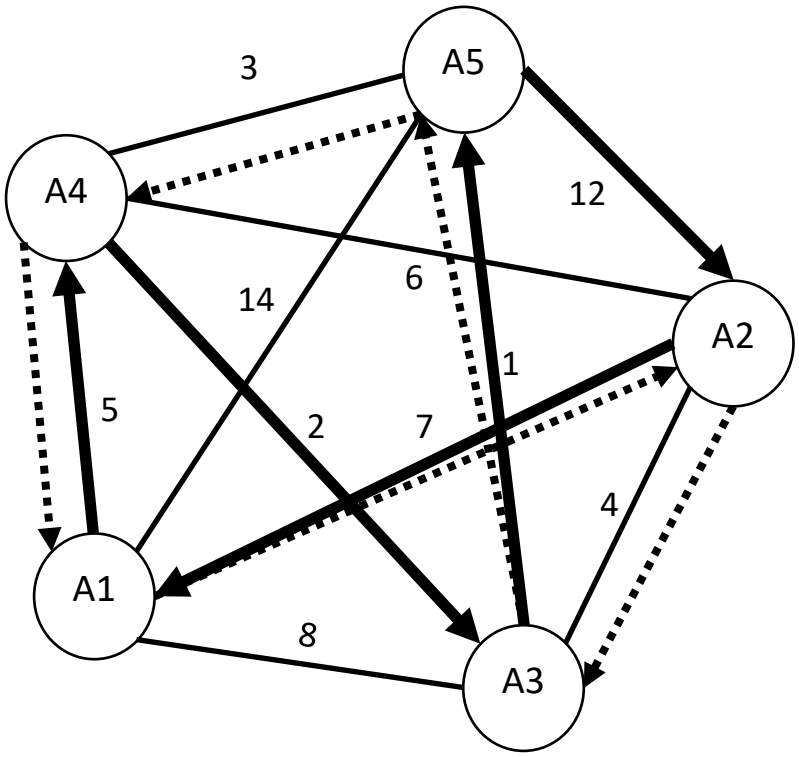


Рис. 5.3 – Приклад застосування «жадібного» алгоритму

Легко помітити, що в такому алгоритмі на кожному кроці вибору можливість обирати звужується, і на останніх кроках пошуку доводиться до вибору ділянок які залишилися (розплачуватися за «жадібність»), табл. 5.2.

Таблиця 5.2

Зменшення кількості ділянок маршруту для вибору у TSP

Кількість необраних міст, n	10000	1000	100	16	10	4	3
Кількість ділянок для вибору, m	49995000	499500	4950	120	45	6	3

Цей алгоритм є найшвидшим (маршрут формується лише один раз) але в більшості випадків дає досить середній результат. Як видно на рис. *3 отриманий «жадібним» алгоритмом маршрут не є оптимальним. Переривчастими лініями показаний оптимальний маршрут.

Більш продуктивними алгоритмами для вирішення задачі комівояжера є так звані метаевристичні алгоритми [9]. Таких алгоритмів існує дуже багато. Ці алгоритми частково імітують фізичні, біологічні процеси, поведінку колоній живих організмів, природні явища тощо. Наприклад, досить продуктивним є алгоритм імітації відпалу (Simulated annealing) . В цьому алгоритмі імітується процес термооброблення, який полягає в нагріванні матеріалу (металу або іншого матеріалу) до температури вище критичної точки, тривалій витримці за цієї температури і подальшому повільному охолодженні з метою наближення структури до рівноважного стану [10].

$$P\left(\overline{x^*} \to \overline{x_{i+1}} \middle| \overline{x_i}\right) = \begin{cases} 1, & F\left(\overline{x^*}\right) - F(\overline{x_i}) \geq 0 \\ \exp\left(-\dfrac{F\left(\overline{x^*}\right) - F(\overline{x_i})}{Q_i}\right), & F\left(\overline{x^*}\right) - F(\overline{x_i}) \geq 0 \end{cases} \qquad (*1)$$

де,
$\overline{x^*}$ – обрана точка пошуку;
$\overline{x_i}$ – поточна точка на маршруті;
$\overline{x_{i+1}}$ – наступна точка на маршруті;
$Q_i > 0$ – i-й елемент довільної спадаючої, що сходиться до нуля.

В лабораторній роботі * наведений приклад виконання алгоритму за методом Монте - Карло, «жадібного» алгоритму та алгоритму імітації відпалу. В цій роботі шляхом проведення обчислювального експерименту можна порівняти ефективність кожного із цих алгоритмів. Не дивлячись на відсутність гарантій для знаходження оптимального розв'язку використання евристичних алгоритмів значно спрощує вирішення ряду задач які важко або неможливо розв'язати з використанням інших більш математично строгих методів та отримати прийнятний (наближений до оптимального або оптимальний) результат.

Отже, основними властивостями евристичних алгоритмів є:

- відсутність єдиного математичного методу до вирішення задачі або класу задач;
- використання імітації фізичних, біологічних та інших процесів для моделювання;
- генерування випадкових чисел для здійснення моделювання.

Контрольні питання

1. *Які алгоритми називають евристичними?*
2. *Чому використання евристичних алгоритмів є більш продуктивним для окремих задач?*
3. *Чому «жадібний» алгоритм є не досить продуктивним для вирішення задачі комівояжера?*
4. *Які алгоритми називають метаевристичними?*

Розділ 6
КЛАСИ СКЛАДНОСТІ ЗАДАЧ В ТЕОРІЇ АЛГОРИТМІВ

Якщо асимптотичний аналіз дозволяє оцінити граничну поведінку алгоритмів стосовно вхідних даних та для різних умов виконання, часовий аналіз робить можливим оцінювання часу виконання конкретних алгоритмічних рішень, то класи складності задач дозволяють об'єднати складність алгоритмів в групи або множини відповідно до математично означених умов їх вирішення. Всі ці класи складності є взаємопов'язаними. До них відносять.

1. Клас складності P(Polynomial). Це множина задач, які можна вирішити алгоритмами з поліноміальним часом. Тобто ці алгоритми мать нотацію Ландау виду $O(n^{\alpha})$ де α>1. Багато дослідників стверджують, що в більшості задач цього класу значення α не перевищує шести. Прийнято вважати, що алгоритми для вирішення таких задач є «швидкими». Вважається також, що задачі цього класу вирішуються комп'ютером приблизно за той самий час, що і детермінованою машиною Тюрінга. Прикладами таких задач є:

- цілочисельні операції додавання, множення, ділення, отримання остачі ділення;
- тест числа на простоту;
- пошук найкоротшого шляху між двома точками;
- множення матриць;
- задачі лінійного програмування тощо.

2. Клас складності NP(Non-deterministic Polynomial). Задачі цього класу можуть вирішуватися за допомогою недетермінованих алгоритмів, таких, наприклад, як евристичні. Результат виконання подібних алгоритмів може бути перевірений за допомогою алгоритмів поліноміальних задачі. Однак, не за будь-яких умов цю перевірку можна здійснити.

До задач NP класу складності відносяться – задача комівояжера, задача про рюкзак, триколірне розфарбовування неорієнтованого графу, задача про клік та багато інших.

Якщо буде математично доведено, що всі задачі NP класу складності підлягають перевірці детермінованими алгоритмами це дозволить (принаймні теоретично) вирішувати задачі NP класу суттєво швидше. Ту частину NP задач яку можна звести до задач класу складності P називають NP - повними (NP - complete або NPC).

Для класу NPC доведена наступна теорема: Якщо існує задача, що належить класу NPC, для якої існує поліноміальний алгоритм розв'язку ($F = O(n^k)$, то клас P збігається з класом NP, тобто P = NP [11].

В даний час доведено існування сотень NP - повних задач [11,12], але ні для однієї з них поки не вдалося знайти поліноміального алгоритму рішення. У зв'язку з цим припускають, що існує наступне співвідношення класів $P \neq NP$, яке часто показують у вигляді діаграми, рис. 6.1. Ця діаграма демонструє не тільки співвідношення класів але й показує, що кількість задач класу NP набагато більша ніж кількість задач класу P та NPC.

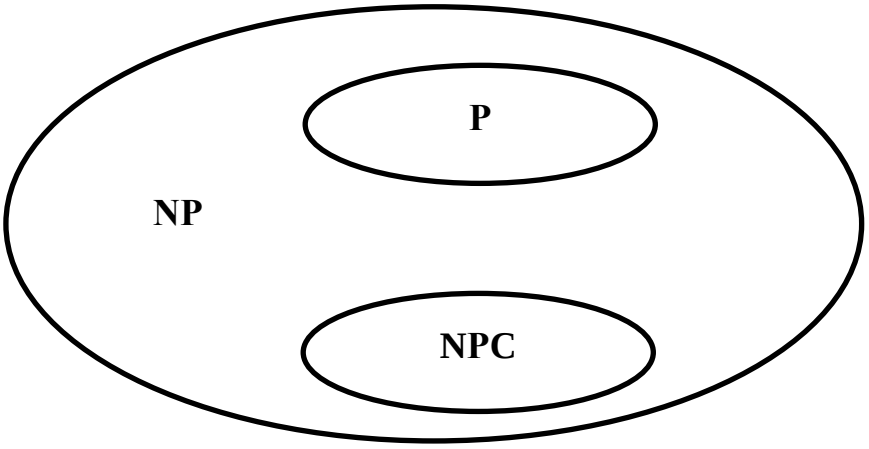


Рис. 6.1. – Умовне співвідношення між класами складності алгоритмів

Зазначеними класами підхід до класифікації складності алгоритмів не вичерпується. Існує більше класів складності [13]. Розглянуті класи відносяться до найвідоміших.

Контрольні питання

1. *Які задачі відносяться до класу NPC?*
2. *Чи можна для NP повної задачі знайти поліноміальний алгоритм рішення?*
3. *До якого класу відносяться задачі динамічного програмування?*
4. *Наведіть приклади NP-повних задач.*

Розділ 7
РЕКУРСИВНІ ФУНКЦІЇ І АЛГОРИТМИ

Рекурсія – це визначення об'єкта через звернення до самого себе. Таке загальне визначення відноситься не тільки до алгоритмів чи програмних кодів. Це визначення тісно пов’язане з рекурсивними функціями як окремого класу обчислюваних функцій. Обчислення цих функцій можливе лише із застосуванням комп’ютерів. В комп’ютерному програмуванні під рекурсією розуміють алгоритм або код програми в яких міститься звернення до самих себе. Термін рекурентні співвідношення (Recurrence relation) пов'язаний з американським науковим стилем і визначає математичне завдання функції за допомогою рекурсії [14].

Рекурсивні алгоритми можуть використовувати пряме звернення функції до самої себе або ***пряму*** рекурсію або звертатися до функції опосередковано, коли функція викликає інші функції зі свого тіла. У такому випадку це непряме звернення або ***непряма*** рекурсія.

Для застосування рекурсивних методів необхідно щоб задачі або об’єкти мали рекурсивні властивості. Це дозволяє більш складну задачу спростити шляхом заміни її на ряд простіших підзадач з дещо іншим набором параметрів.

Глибиною рекурсії називається максимальна кількість рекурсивних викликів процедури без повернень, що відбувається під час виконання програми.

Головною вимогою до рекурсивних процедур є переривання. Воно полягає в тому, що виклик рекурсивної процедури має виконуватися за умовою, яка на якомусь рівні рекурсії стане хибною і виконання процедури повинно закінчитися. Одним із класичних прикладів рекурсивного алгоритму є алгоритм розрахунку ряду чисел де наступне число є сумою попередніх (числа Фібоначчі). Співвідношення для розрахунку такої послідовності є наступним:

$$f_n = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ f_{n-1} + f_{n-2}, & n > 1 \end{cases} \qquad (7.1)$$

Програмна реалізація рекурсивного алгоритму мовою Python матиме наступний вигляд:

```
def F(n):
    print(n)#Друкування викликів функції
    if n in (1, 2):
        return 1
    return F(n - 1) + F(n - 2)
print(F(5))#Результат для 5-ти чисел
```

Якщо подати розрахунки цього числа у вигляді дерева, то виклик функції буде виглядати наступним чином.

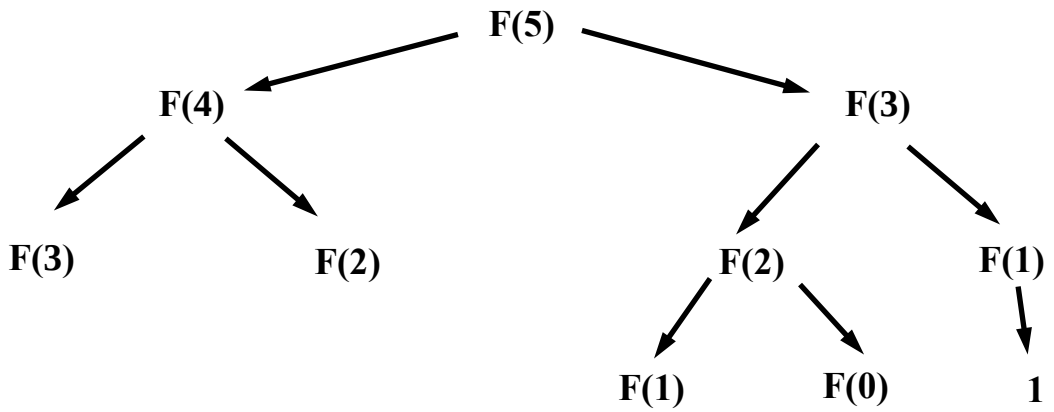


Рис. 7.1. Дерево викликів функції для розрахунку 5-го числа Фібоначчі

Виклики для цього дерева легко виявити запустивши код програмної реалізації. Результат буде наступним.

5 4 3 2 1 2 3 2 1 5

Зрозуміло, що зі збільшенням кількості чисел обсяг розрахунків буде суттєво зростати.

Це можна простежити самостійно задаючи для розрахунку інші числа шляхом зміни останнього рядка, наприклад, **print(F(10))#Результат для 10-ти чисел**. Очевидно, що трудомісткість такого рекурсивного алгоритму можна оцінювати використовуючи наведену деревоподібну структуру.

Обсягом рекурсії називають кількість вершин повного рекурсивного дерева без одиниці для конкретного набору параметрів.

Аналіз рекурсивних алгоритмів

Методи, які використовуються для аналізу алгоритмів, що були розглянуті в попередніх главах для рекурсивних процедур є непридатними. У випадку з рекурсивними процедурами розрахувати трудомісткість для найгіршого, найкращого та середнього випадків практично неможливо. Це не означає, що неможливо визначити нотації для таких алгоритмів або підрахувати їх часову складність. Крім наведеного аналізу за допомогою рекурсивного дерева, який не завжди можна здійснювати для аналізу рекурсивних алгоритмів використовуються наступні методи:

- метод ітерацій або метод підстановки;
- метод математичної індукції.

Метод ітерацій передбачає заміну рекурентної частини алгоритму нерекурентною (ітераційною). Заміна проводиться до того часу поки не вдасться визначити загальний принцип і проаналізувати його вже відомими методами. Наприклад, для пошуку елементу в масиві створюється таке перетворення.

$$T_n^{search} = 1 \times O(1) + O\left(T_{n-1}^{search}\right) = 2 \times O(1) + O\left(T_{n-2}^{search}\right) = 3 \times O(1) + O\left(T_{n-3}^{search}\right) + \ldots \quad (7.2)$$

Виходячи з цього наведену послідовність можна записати так

$$T_n^{search} = T_{n-1}^{search} + k \times O(1)$$

або

$$T_n^{search} = T_0^{search} n \times O(1) = O(n).$$

Метод математичної індукції полягає у "вгадуванні" рішення або висуванні гіпотези і подальшого доведення цієї гіпотези. Наприклад, для такої функції

$$\begin{cases} f(0) = 1 \\ f(n+1) = 2 \cdot f(n) \end{cases} \qquad (7.3)$$

можна зробити припущення, що $f(n) = 2^n$

У такому випадку для $f(n) = 2^n$ справедливим є $f(0) = 1$, а для $f(n+1) \Rightarrow 2 \cdot 2^n = 2^{n+1}$

Подібні доведення часто є досить трудомісткими.

Рекурсивні алгоритми відносять до ресурсоємних, остільки вони використовують стекову область пам'яті яка швидко заповнюється при великій кількості самовикликів рекурсивних функцій. Тому, під час практичної роботи слід ретельно підходити до використання цих алгоритмів, особливо в тих випадках де вони можуть бути ефективно замінені на ітераційні.

Контрольні питання

1. *В чому полягає відмінність між ітераційними та рекурсивними алгоритмами?*
2. *Як визначається глибина рекурсії?*
3. *Яка головна умова для використання рекурсивних процедур?*
4. *Якими методами здійснюється аналіз рекурсивних алгоритмів?*

Розділ 8
ШИФРУВАННЯ ДАНИХ І АЛГОРИТМИ. МОДУЛЬНА АРИФМЕТИКА. ХЕШУВАННЯ

Шифрування даних є одним із розділів теорії алгоритмів. Фактично шифрування це перетворення інформації певним чином. Таке перетворення виконувалося без використання комп'ютерів з давніх часів. Найбільш відомий алгоритм шифрування який практично використовувався називається шифр Цезаря. Фактично це зміщення алфавіту на певну позицію.

Таблиця 8.1

Шифрування шифром Цезаря

№(*x*)	Алфавіт	Зсунутий алфавіт	Текст	Зашифрований текст
0	А	Г		
1	Б	Г'		
2	В	Д	В	Д
3	Г	Е		
4	Г'	Є		
5	Д	Ж		
6	Е	З		
7	Є	И		
8	Ж	І		
9	З	Ї		
10	И	Й		
11	І	К	І	К
12	Ї	Л		
13	Й	М		
14	К	Н	К	Н
15	Л	О		
16	М	П		
17	Н	Р	Н	Р
18	О	С	О	С
19	П	Т		
20	Р	У		
21	С	Ф		
22	Т	Х		

Продовження табл. 8.1

Шифрування шифром Цезаря

№(x)	Алфавіт	Зсунутий алфавіт	Текст	Зашифрований текст
23	У	Ц		
24	Ф	Ч		
25	Х	Ш		
26	Ц	Щ		
27	Ч	Ь		
28	Ш	Ю		
29	Щ	Я		
30	Ь	А		
31	Ю	Б		
32	Я	В		

Використовуючи порядкові номери табл. 8.1 шифрування і розшифровування шифру Цезаря можна подати формулами:

$$y = (x + k)\ mod\ n$$

$$x = (y - k)\ mod\ n\ , \qquad (8.1)$$

де

x – порядковий номер символу тексту що шифрується;

y – порядковий номер символу шифрованого тексту;

n – кількість символів алфавіту(потужність алфавіту);

k – ключ (зсув на k позицій).

Не дивлячись на те, що зазначений алгоритм шифрування в наш час видається примітивним на його основі можна зробити певні висновки.

По-перше, із формул (8.1) видно, що для створення алгоритму цього шифру використовується ділення по модулю. Використання модульної арифметики необхідне у всіх алгоритмах шифрування. По-друге, особливістю такого алгоритму є наявність єдиного ключа. Ця особливість притаманна всім алгоритмам так званого симетричного шифрування.

Модульна арифметика або арифметика цілих чисел - це така категорія арифметики, яка використовує лише цілі числа.

Її ще називають арифметикою конгруентності або тактовою арифметикою. Вона активно використовується для шифрування інформації. Окрім шифрування модульна арифметика використовується також в економіці, соціальних науках, праві та в інших галузях де пропорційний поділ і розподіл мають важливе значення.

Простий приклад пояснює що називають діленням по-модулю.

$$\frac{13}{5} = 2(\text{ціле число})\ 3(\text{остача})$$

Щоб більш детально ознайомитися з використанням модульної арифметики розглянемо приклад алгоритму шифрування з використанням автоключового шифр. Для цього використаємо формули – для шифрування:

$$S_i = (V_i + k)\ mod\ n \qquad (8.2)$$

Для розшифровування:

$$V_i = (S_i + k)\ mod\ n, \qquad (8.3)$$

де

n – кількість символів алфавіту(потужність алфавіту);

k – ключ (певне число);

V_i –i-й символ тексту;

S_i – зашифрований i-й символ тексту.

Результат роботи автоключового шифру показані на рис 8.1. Літери вихідного тексту пронумеровані відповідно до номерів табл. 8.1. Ключ k=12. Другий рядок номерів зміщений праворуч на довжину ключа. Щоб зашифрувати, наприклад, літеру У необхідно виконати дію – (23+9) mod 33, де 33 це кількість літер алфавіту. Отриманий номер співставленні з літерою алфавіту табл 8.1 дає літеру Я. Розшифровування цієї літери виконується у зворотному порядку – (32-9) mod 33.

Шифрування

Вихідний текст	М	И	З	У	К	Р	А	Ї	Н	И
	16	10	9	23	14	20	0	12	17	10
Ключ →	12	16	10	9	23	14	20	0	12	17
	28	26	19	32	4	1	20	12	29	27
Шифр	Ш	Ц	П	Я	Г'	Б	Р	Ї	Щ	Ч

Розшифровування

	Ш	Ц	П	Я	Г'	Б	Р	Ї	Щ	Ч
	28	26	19	32	4	1	20	12	29	27
Ключ →	12	16	10	9	23	14	20	0	12	17
	16	10	9	23	14	20	0	12	17	10
Розшифрований текст	М	И	З	У	К	Р	А	Ї	Н	И

Рис 8.1. Приклад шифрування і розшифровування автоключовим алгоритмом

Подібні але більш досконалі алгоритми шифрування з одним(таємним або закритим) ключем сьогодні активно використовуються в інформаційних системах. До них відноситься алгоритм який має абсолютну криптографічну стійкість – шифр Вернама [15].

Прості числа та їх використання в алгоритмах шифрування

Модульна арифметика і прості числа складають основу алгоритмів шифрування в сучасних криптографічних системах. Зокрема в алгоритмах асиметричного шифрування ключове значення мають.

Функція Ейлера – $\varphi(x)$ яка показує кількість натуральних чисел, що не є більшими за число x і взаємно простих(таких, що не мають спільних дільників) з ним. Наприклад, для простого число 12 ця функція буде дорівнювати 4, що складає множину чисел – 3, 5, 7, 11.

Обернене число за модулем φ це таке число x для якого виконується тотожність [15]

$$a^{\varphi(x)} \equiv 1 \ (mod\ x), \qquad (8.4)$$

де

$\varphi(x)$ – кількість взаємно простих з *x* чисел від 1 до *x*.

Тобто, залишок від ділення за модулем *x* числа a повинен дорівнювати 1.

Асиметричні алгоритми шифрування мають два ключі – відкритий та закритий. Розглянемо приклад який пояснює роботу асиметричного алгоритму RSA[16].

Необхідно зашифрувати один символ алфавіту – літеру Р. В табл. 8.1 вона під номером 20.

Оберемо два простих числа – p=3 та q=11 та знайдемо їх добуток $p \cdot q = 33$. Отже, модуль дорівнює 20.

Знайдемо функцію Ейлера для числа 33 – $\varphi = (p-1) \cdot (q-1) = 20$. Тобто множина взаємно простих чисел і модулем 21 буде такою – $\{1, 2, 4, 5, 7, 8, 10, 13, 14, 16, 17, 19, 20, 23, 25, 26, 28, 29, 31, 32\}$. Оберемо просте число менше 12. Нехай це буде 3. Тоді відкритий ключ буде складатися із двох цифр – $\{3,\ 33\}$.

Щоб сформувати закритий ключ необхідно використовуючи прості числа із множини знайденої функції Ейлера і отримати число зворотне до 3 по модулю 20. Прості числа знайденої функції складають множину $\{1, 2, 5, 7,\ 13, 17, 19\}$. Виходячи із формули (8.4) це $a^{20} \equiv 1\ (mod\ 3)$, або $a \cdot 3\ mod\ = 1$. Із наявної множини це може бути тільки одне число менше 20 – 7, табл 8.2

Таблиця 8.2

Отримання числа для закритого ключа

№	Числа <20	Множник	Добуток	Дільник	Остача
1	1	3	3	20	3
2	2	3	6	20	6
3	5	3	15	20	15
4	7	3	21	20	1
5	13	3	39	20	19
6	17	3	51	20	11
7	19	3	57	20	17

Отже, закритий ключ також складається із двох цифр – $\{7,\ 33\}$.

Зашифруємо і розшифруємо за допомогою цих ключів літеру Р. Зашифроване значення за допомогою відкритого колюча дорівнює – $20^3 mod\ 33 = 14$.

Щоб розшифрувати цю літеру використаємо закритий ключ – $14^7 mod\ 33 = 20$. Номер літери Р – 20. Літера розшифрована.

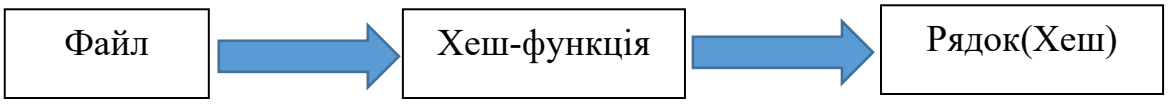
Подібним чином можна шифрувати і розшифровувати будь-яку інформацію. Наявність же відкритого(для шифрування) і закритого(для розшифровування) ключів дає практичну можливість обмежити кількість осіб які можуть розшифровувати інформацію до однієї особи. Таким чином забезпечується не тільки математична але і фізична криптостійкість. Зловмисникам практично неможливо «вирахувати» закритий ключ маючи відкритий. Для цього потрібно розкласти модуль великого числа, яке застосовують для шифрування на прості множники (факторизація). Навіть на для сучасних комп'ютерів така задача непосильна за прийнятний час.

Звичайно, щоб на практиці здійснювати асиметричне шифрування використовують великі прості числа. Тому, в сучасних алгоритмах шифрування є важливим знаходження великих простих чисел і робота з ними.

Найбільш простим алгоритмом, знаходження простих чисел є решето Ератосфена. В сучасним і найбільш ефективним тестом числа на простоту є тест AKS[17].

Алгоритми хешування

Хешування (Hashing) це процес надання відповідності масиву інформації шляхом запису рядка фіксованої довжини. Іншими словами якщо існує файл будь-якого розміру для нього можна записати унікальний відповідний рядок символів. Таке хешування здійснюється з використанням хеш-функцій.



Будь-яка зміна файлу вже не відповідатиме записаному рядку і будь-яка зміна рядка не відповідатиме файлу. Звідси виникають наступні завдання.

1. Створювати такий рядок, який би не відповідав іншому файлу тобто був унікальним. Подібна відповідність (коли один і той же рядок відповідає двом різним файлам) називається колізією другого роду. Коли ж два різних рядки відповідають одному і тому ж файлу, то тоді, така відповідність називається колізією першого роду.

2. Максимально утруднити або унеможливити знаходження відповідності рядок-файл сторонніми особами.

3. Забезпечити виконання операції хешування (створення рядка) за прийнятний час.

Виконання цих завдань забезпечить надійне зберігання послідовності даних без безпосереднього до них звертання адже набагато швидше знаходити потрібні файли через їх відповідники, а не безпосередньо звертатися до них. Використання хешування також забезпечує надійність зберігання даних. Будь-яке спотворення файлу призводитиме до невідповідності його хеш-рядку.

Створення рядка довжиною 2^n для файлу, що складається з множини інформаційних одиниць $a_0, a_1, \dots a_n$ в найпростішому випадку може бути здійснена шляхом ділення по модулю

$$a_0, a_1, \dots a_n \to (a_0 + a_1 + \cdots + a_n) mod 2^n \qquad (8.5)$$

Однак, у цьому випадку уникнути колізій не вдасться, бо навіть проста заміна доданків послідовності $a_0 + a_1 + \cdots + a_n$ призведе до створення того самого рядка. Щоб вийти із цієї неприємної ситуації більш продуктивним буде використати поліном

$$a_0, a_1, \dots a_n \to (a_0 + a_1 \cdot p + \cdots + a_n \cdot p^n) mod 2^n, \qquad (8.6)$$

де p – деяке число.

Звісно для створення хешів які використовуються на практиці використовують спеціальні алгоритми хешування, які забезпечують практичне уникнення колізій першого і другого роду.

Створення хешів дає можливість ефективно здійснювати пошук та використання інформації шляхом створення хеш таблиць у яких ця інформація ефективно і надійно обробляється, зокрема для збереження конфіденційної інформації такої як логіни і паролі користувачів.

Найбільш розповсюдженими алгоритмами хешування насьогодні є – SHA-256, SHA-512, KECCAK256 та інші. Алгоритми хешування постійно вдосконалюються з метою забезпечення надійного зберігання і обробки інформації.

Контрольні питання

1. *Які існують типи алгоритмів шифрування?*
2. *Чим відрізняється алгоритм симетричного і асиметричного шифрування?*
3. *Які завдання вирішуються хешуванням даних?*
4. *Де використовуються алгоритми хешування?*

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Горлова Т. М., Бобровнік К. Є., Лімaнська Н. В. Теорія алгоритмів: конспект лекцій для студентів напряму підготовки 6.050101 «Комп'ютерні науки» денної та заочної форм навчання. Київ: НУХТ, 2015. 95 с. URL: http://library.nuft.edu.ua/ebook/file/M51.21.pdf
2. Information processing - Documentation symbols an.d conventions for data, program and system flowcharts, program network charts and system resources charts: International Standard ISO 5807:1985. URL: https://cdn.standards.iteh.ai/samples/11955/1b7dd254a2a54fd7a89d616dc0570e18/ISO-5807-1985.pdf
3. Rajendra Kumar. Theory of Automata. Tata McGraw-Hill Education, 2010. 343 p.
4. Turing Machine Simulations: Classic and Quantum/ Hafsa Yazdani, Fatemah Al Zayer, Naya Nagy, Marius Nagy. *IJCEE*. 2014. Vol.6 (1). Pp. 49-53. URL: https://www.researchgate.net/publication/271303578_Turing_Machine_Simulations_Classic_and_Quantum
5. Anil Maheshwari, Michiel Smid. Introduction to Theory of Computation. Ottawa: School of Computer Science Carleton University, 2019. URL: https://cglab.ca/~michiel/TheoryOfComputation/TheoryOfComputation.pdf
6. Успенський В. А., Семенов А. Л. Алгоритмічні проблеми, що можна і не можна вирішити. *Квант.* 1985. № 7 С. 9-15.
7. Дороговцев А. Я. Математичний аналіз: підручник: У двох частинах. Ч.1. Київ: Либідь, 1993. 320 с.
8. Levitin Anany. Introduction to The Design and Analysis of Algorithms. 3rd Edition. New Jersey: Addison Wesley, 2012. 565 p.
9. Emperor Penguins Colony: a new metaheuristic algorithm for optimization/ Sasan Harifi, Madjid Khalilian, Javad Mohammadzadeh, Sadoullah Ebrahimnejad. *Evolutionary Intelligence*. 2019.Vol. 12. Pp.211–226. URL: https://www.researchgate.net/publication/331328734_Emperor_Penguins_Colony_a_new_metaheuristic_algorithm_for_optimization
10. Henderson Darrall, Jacobson Sheldon, Johnson Alan. The Theory and Practice of Simulated Annealing. *Handbook of Metaheuristics/*Ed. By Fred Glover, Gary A. Kochenberger. NY: Springer New York, 2003. Pp. 287-319. URL: https://link.springer.com/chapter/10.1007/0-306-48056-5_10
11. Introduction to Algorithms/ Ed. by Thomas H. Cormen, Charles E. Leiserson, Ronald L.Rivest, Clifford Stein. 2nd edition. The MIT Press, 2001. 1184 p. URL: https://athena.nitc.ac.in/summerschool/Files/clrs.pdf
12. Theory and Practice of Computation: Proceedings of the Workshop on Computation: Theory and Practice (WCTP 2019) (September 26-27, 2019, Manila, The Philippines). URL:https://www.routledge.com/Theory-and-Practice-of-Computation-Proceedings-of-the-Workshop-on-Computation/Nishizaki-Numao-Suarez-Caro/p/book/9780367545888

13. Матвієнко М. П. Математична логіка та теорія алгоритмів: навчальний посібник. Київ: Видавництво Ліра-К, 2015. 212 с.

14. Rogers Hartley. Theory of recursive functions and effective compatibility. The MIT Press, 1992. URL: https://www.academia.edu/7823095/Theory_of_Recursive_Function_and_Effective_Computability

15. Christof Paar, Jan Pelzl. Understanding Cryptography. A Textbook for Students and Practitioners/ Foreword by Bart Preneel. London-New York: Springer Heidelberg Dordrecht, 2010. 382 p. URL: https://swarm.cs.pub.ro/~mbarbulescu/cripto/Understanding%20Cryptography%20by%20Christof%20Paar%20.pdf

16. Шифрування і розшифрування повідомлень за схемою несиметричного шифрування RSA і Ель-Гамаля. URL: http://lib.kart.edu.ua/bitstream/123456789/1530/1/%D0%A1%D0%9A%D0%A1%2089.pdf

17 Agrawal M., Kayal N., Saxena N. PRIMES is in P. *Annals of Mathematics.* 2004. Vol. 160, Iss. 2. P. 781–793. URL: https://annals.math.princeton.edu/wp-content/uploads/annals-v160-n2-p12.pdf

Додатки

Додаток А. Умови та інструкції для виконання лабораторних робіт

Програмне забезпечення для виконання робіт

Для виконання лабораторних робіт використовуються програмні імітатори машини Поста та машини Тюрінга [] та редактор мови програмування С++ – **Dev-C++.** Цей редактор являє собою інтегроване середовище розробки для мов програмування С/С++. Воно розповсюджується на засадах вільного програмного забезпечення і знаходиться у вільному доступі в мережі Інтернет. Його завантаження може бути здійснене за посиланням https://sourceforge.net/projects/dev-cpp/

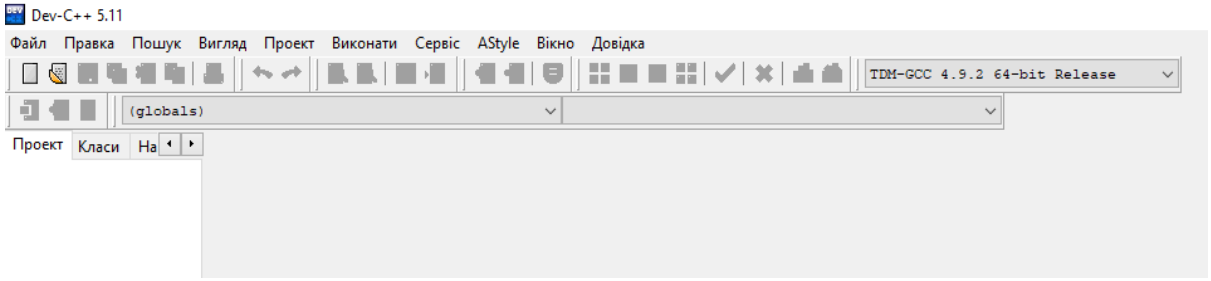


Рис. А.1. Фрагмент інтерфейсу Dev-C++

Для розміщення комп'ютерного коду і запуску програм в Dev-C++ необхідно створити вихідний файл натисненням комбінації кнопок CTRL+N або використати меню цієї програми, рис. * 2.

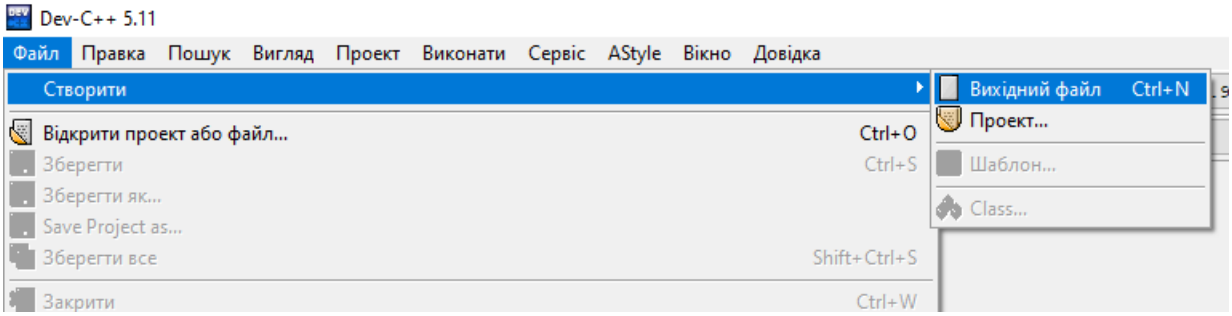


Рис. А.2. Створення вихідного файлу для розміщення коду

Після створення нового файлу слід розміщувати код в поле Untitiled1 надрукувавши його або скопіювавши. Після цього компіляція коду здійснюється функціональною кнопкою F9, а виконання кнопкою F10.

Під час виконання робіт всі програмні додатки на комп’ютері повинні бути закритими, включаючи браузери.

Зразок виконання лабораторної роботи
Експериментальне оцінювання часу роботи алгоритму сортування обміном (бульбашковий алгоритм)

Мета роботи

Вивчити особливості алгоритму сортування обміном

Необхідне обладнання

Комп'ютер з операційною системою Windows версії не нижче 7.

Встановлений редактор мови програмування **DEV-C++**.

Теоретична частина

Бульбашковий алгоритм або алгоритм сортування обміном відносять до одного з повільних алгоритмів сортування. Сутність цього алгоритму зводиться до пошуку мінімального числа в масиві і покрокового порівняння його з іншими елементами масиву та поступового переміщення цього елементу на початок масиву шляхом обміну з іншими елементами. Подібна процедура переміщення повторюється для всіх елементів масиву. Нижче наведена послідовність переміщення мінімального елемента масиву, що нагадує підняття бульбашки в рідині (звідси і назва алгоритму).

пошук мінімуму	4		4		4	порівняння	4	заміна	1
	7	порівняння	7	заміна	1		1		4
	1		1		7		7		7
	3		3		3		3		3

Порядок виконання роботи

1. Закрити всі сторонні програми на комп'ютері і запустити Dev-C++

2. Скопіювати код алгоритму до вікна редактора Dev-C++. Відкомпілювати код (кнопка F9).

3. Запустити програму на виконання кнопкою F10. Записати час виконання програми до табл. А.1 в колонку під назвою «Час сортування невідсортованого масиву, с».

4. Повторити пункт 3 змінюючи рядок коду **size = 1000; //Розмір масиву** та компілюючи і запускаючи код у відповідності зі значеннями табл. А.1. Заповнити колонку табл. А.1 – «Час сортування невідсортованого масиву, с».

5. Повторити пункти 3 та 4 змінюючи фрагменти коду. Для частково відсортованого масиву розкоментувати фрагмент

```
if (i>n/2)
        {
                arr[i] = i + 1001;//Генерація відсортованих чисел
        }
        else
        {
                arr[i] = 1 + rand() % 1001;//Генерація випадкового числа від 1 до 1001
        }
```

І закоментувати два рядки

```
//arr[i] = 1 + rand() % 1001;//Генерація випадкового числа від 1 до 1001
//arr[i] = i + 1001;//Генерація відсортованих чисел
```

Для відсортованого масиву розкоментувати рядок

```
arr[i] = i + 1001;//Генерація відсортованих чисел
```

а наступні рядки закоментувати

```
/*if (i>n/2)
        {
                arr[i] = i + 1001;//Генерація відсортованих чисел
        }
        else
        {
                arr[i] = 1 + rand() % 1001;//Генерація випадкового числа від 1 до 1001
        }
*/
//arr[i] = 1 + rand() % 1001;//Генерація випадкового числа від 1 до 1001
```

За результатами таблиці побудувати графіки зміни часу виконання програм і зробити висновки

Таблиця А.1

Зразок таблиці для проведення обчислень за алгоритмом сортування обміном

№	Кількість елементів масиву	Час сортування невідсортованого масиву, с	Час частково відсортованого масиву, с	Час сортування відсортованого масиву, с
1				
2				
3				
4				
5				
6				

Код бульбашкового алгоритму для копіювання до Dev C++

```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;
enum SortOrder//напрям сортування
{
   ASC,    //за зростанням
   DESC    //за спаданням
};

void swapElements(int &element1, int &element2)//обмін елементів масиву
{
   int tempVar = element1;
   element1 = element2;
   element2 = tempVar;
}

bool isSorted(int a, int b, SortOrder sortOrder)//перевірка правильності розташування елементів
{
   if (sortOrder == ASC)
   {
      return a <= b;
   }
   else
```

```
    {
        return a >= b;
    }
}

int *bubbleSort(int *array, int n, SortOrder sortOrder)//сортування бульбашкою
{
    bool swappedFlag = false;

    for (int i = 1; i < n; i++)
    {
        swappedFlag = false;
        for (int j = 0; j < n - i; j++)
        {
            if (!isSorted(array[j], array[j + 1], sortOrder))
            {
                swapElements(array[j], array[j + 1]);
                swappedFlag = true;
            }
        }
        if (!swappedFlag)//якщо обмінів не було, перериваємо цикл
        {
            break;
        }
    }
    return array;
}

int *fillArray(int *arr, int n)//заповнення масиву з клавіатури або генерація випадкових чисел
{
  srand(time(NULL));
    for (int i = 0; i < n; i++)
    {
 /*       if (i>n/2)
        {
                arr[i] = i + 1001;//Генерація відсортованих чисел
        }
        else
        {
                arr[i] = 1 + rand() % 1001;//Генерація випадкового числа від 1 до 1001
        }
```

```
*/
      arr[i] = 1 + rand() % 1001;//Генерація випадкового числа від 1 до
1001
      //arr[i] = i + 1001;//Генерація відсортованих чисел
   }
   return arr;
}

void printArray(int *arr, int n)//Вивід відсортованого масиву на екран
{
/*    for (int i = 0; i < n; i++)
   {
      cout << arr[i] << " ";
   }
*/
}
int main(int argc, char **argv)
{
   int *arr;
   int size;
   size = 1000; //Розмір масиву
   arr = new int[size];
   arr = fillArray(arr, size);
   arr = bubbleSort(arr, size, ASC);
   printArray(arr, size);
   delete arr;//видалення масиву

   return 0;
}
```

Результат виконання роботи

Таблиця А.2

Результати проведення обчислень за алгоритмом сортування обміном

№	Кількість елементів масиву	Час сортування невідсортованого масиву, с	Час частково відсортованого масиву, с	Час сортування відсортованого масиву, с
1	1000	0,022	0,021	0,021
2	5000	0,12	0,07	0,024
3	10000	0,43	0,23	0,027
4	20000	1,69	0,84	0,024
5	50000	10,41	5,26	0,02
6	100000	41,91	20,86	0,024

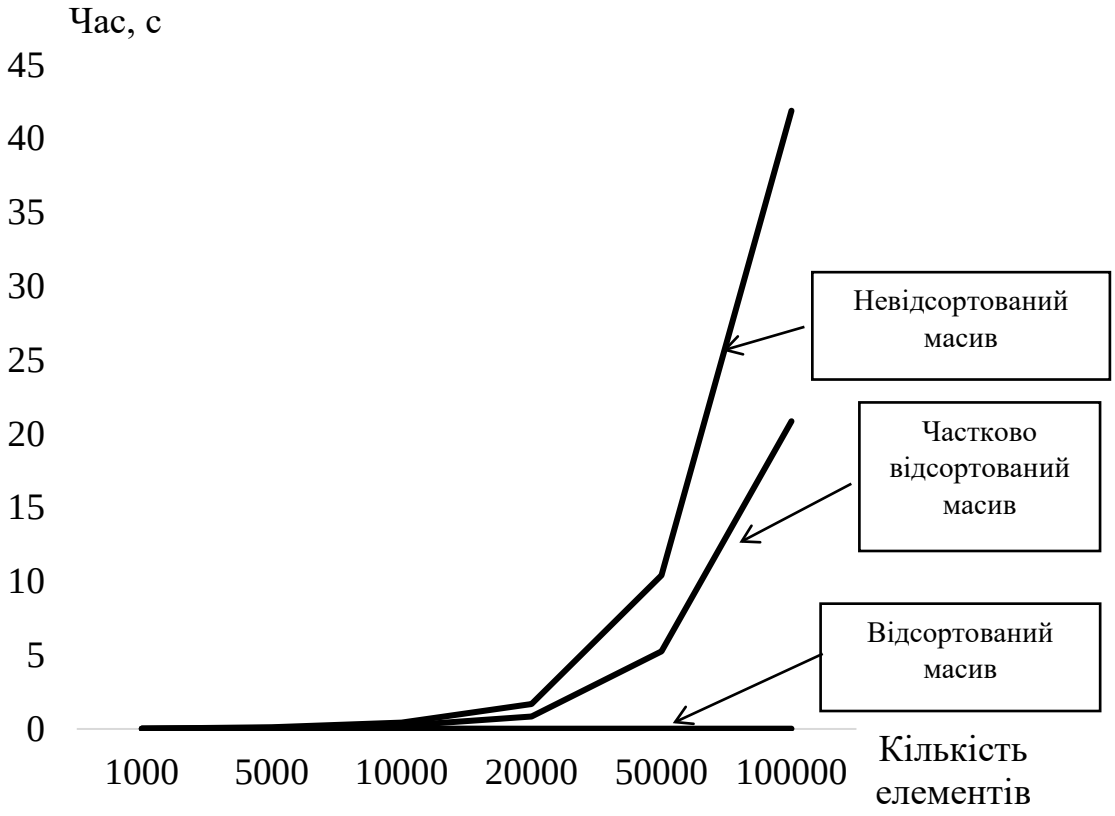


Рисунок А.1. Графіки залежності часу виконання алгоритму від кількості елементів масиву для відсортованого, частково відсортованого та невідсортованого масивів

Висновки

Судячи з результатів експерименту алгоритм сортування обміном відноситься до кількісно-параметричних алгоритмів за трудомісткістю. Два крайніх варіанти які були використані в роботі – з відсортованим і невідсортованим масивами, що час виконання алгоритму сортування обміном критично залежить від ступеню відсортованості вхідного масиву. Із збільшенням кількості елементів зростає час виконання алгоритму.

Як видно на графіку зростання часу виконання невідсортованого масиву(найгірший варіант) має вигляд наближений до параболи. Практичне застосування цього алгоритму в алгоритмах сортування з великою кількістю невідсортованих елементів або в алгоритмах зі значною частотою використання невідсортованих масивів є неефективним.

Додаток Б. Лабораторні роботи
Лабораторна робота № 1
ОЗНАЙОМЛЕННЯ З РОБОТОЮ МАШИНИ ПОСТА

Мета роботи

Вивчити склад і роботу машини Поста

Необхідне обладнання

Комп'ютер з операційною системою Windows версії не нижче 7.

Некомерційна комп'ютерна модель - імітатор машини Поста.

Теоретична частина

Машина поста являє собою математичну концепцію комп'ютерних обчислень. До складу машини водять:

- нескінченна стрічка з комірками;
- каретка для запису та зчитування команд.

Машина поста використовує унарну обчислювальну систему, що використовує мітки. Мітки та пробіли між ними складають алфавіт машини. Існує розширений варіант машини до складу якого входить двійкова система – 0 та 1 та пробіл.

Система команд яка умовно називається програма машини Поста містить п'ять команд:

1. записати мітку, перейти до і-го рядка програми - V;
2. записати стерти мітку, перейти до і-го рядка програми - X;
3. перемістится вліво, перейти до і-го рядка програми - ←;
4. перемістится вправо, перейти до і-го рядка програми - →;
5. зупинитися - !

Для машини Поста створено багато комп'ютерних симуляторів, що дозволяють імітувати її роботу. Вони дозволяють виконувати елементарні математичні операції над мітками в унарній обчислювальній системі. Зовнішній вигляд одного з таких симуляторів показаний на рис. * 1.

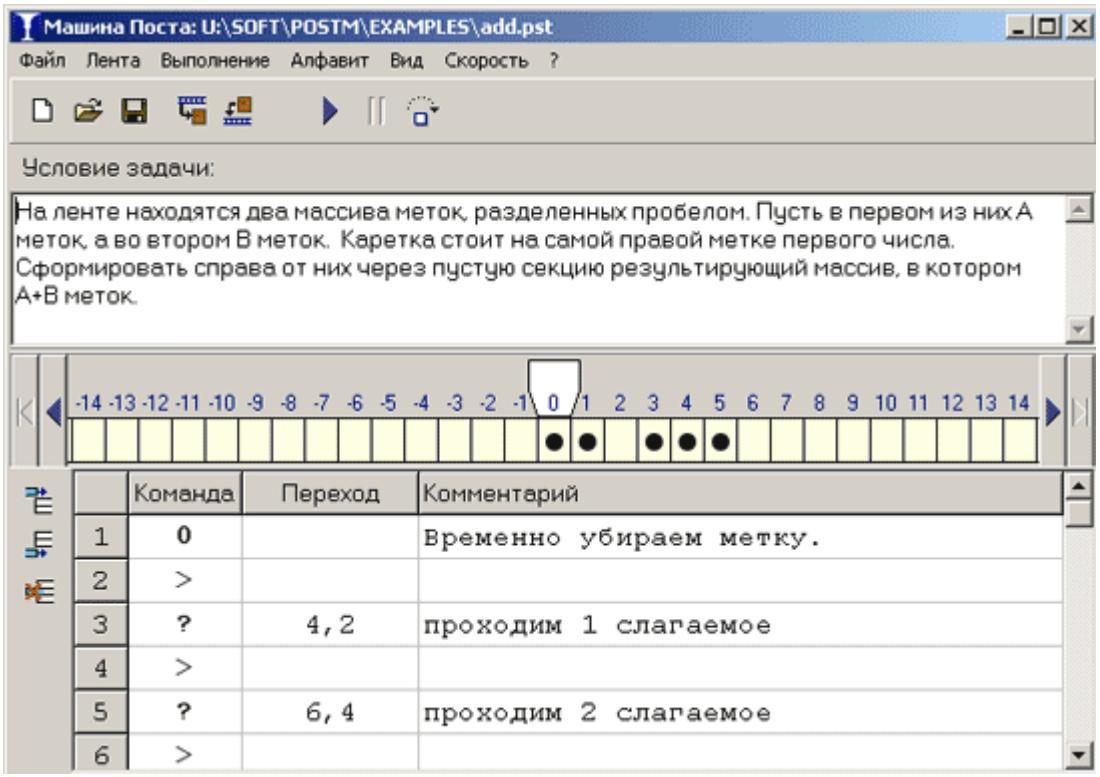


Рис. Б.1 – Зовнішній вигляд програми симулятора машини Поста

Приклад вирішення арифметичної задачі машиною Поста. Необхідно виконати операцію віднімання в унарній системі числення

	*	*	*		*	*	
					V		

Результат виконання повинен бути наступним

	*						
		V					

Система команд для вирішення цієї задачі наведена нижче.

1. ← — крок ліворуч
2. ? 1; 3 — якщо в комірці пусто перейти до 1 кроку, якщо ні — до 3-го
3. X — видалити мітку
4. → — крок праворуч
5. ? 4; 6 — якщо в комірці пусто перейти до 4 кроку, якщо ні — до 6-го
6. X — видалити мітку
7. → — крок праворуч
8. ? 9; 1 — якщо в комірці пусто перейти до 9 кроку, якщо ні — до 1-го
9. ! — кінець

Порядок виконання роботи

Виконати операцію над унарними числами, відповідно до варіанту, заданому в табл. * 1. Номер варіанта для кожного студента визначає викладач. Для виконання роботи використати симулятор машини Поста, що розміщений на сайті https://kpolyakov.spb.ru/prog/post.htm. Результат виконання надати викладачу для оцінювання.

Таблиця Б.1

Варіанти завдань для виконання

Номер варіанта	Завдання
1	Додати ***** та **. Каретка ліворуч.
2	Відняти від **** наступне ***. Каретка праворуч.
3	Зменшити ***** на ***. Каретка ліворуч.
4	Розділити **** на **. Каретка ліворуч.
5	Розділені 2-ма пробілами масиви з'єднати в один **** та ***. Каретка праворуч.
6	Витерти середню мітку послідовності *******. Каретка ліворуч.
7	Скопіювати послідовність ***** і зменшити її на *. Каретка ліворуч.
8	З'єднати розділені мітки в одну послідовність * * * * *. Каретка праворуч.
9	Скласти дві послідовності **** та ***** розділені 2 пробілами та розділити їх на 3. Каретка ліворуч.
10	Скласти 3 послідовності розділені 2 та 1 пробілами **** *** ***. Каретка ліворуч.
11	Перемножити дві послідовності *** та **. Каретка ліворуч.

Лабораторна робота № 2
ОЗНАЙОМЛЕННЯ З РОБОТОЮ МАШИНИ ТЮРІНГА

Мета роботи

Вивчити склад і роботу машини Тюрінга

Необхідне обладнання

Комп'ютер з операційною системою Windows версії не нижче 7.

Некомерційна комп'ютерна модель - імітатор машини Тюрінга.

Теоретична частина

Машина Тюрінга являє собою математичну концепцію комп'ютера і так само як і машина Поста містить нескінченну стрічку та каретку, але на відміну від машини Поста вона має алфавіт символів та програмований перелік станів. На основі цієї машини Алан Тюрінг сформулював інтуїтивне поняття алгоритму. Існує певна кількість варіацій машини Тюрінга, що відрізняються від первинного варіанту кількістю кареток (їх може бути декілька), кількістю стрічок, обмеження стрічки тощо.

Порядок виконання роботи

Виконати операції над послідовностями символів алфавіту описаними в табл. Б.2 відповідно до варіанту. Номер варіанта кожному студентові визначає викладач. Для виконання роботи використати симулятор машини Тюрінга, що розміщений на сайті https://kpolyakov.spb.ru/prog/turing.htm. Результат виконання надати викладачу для оцінювання.

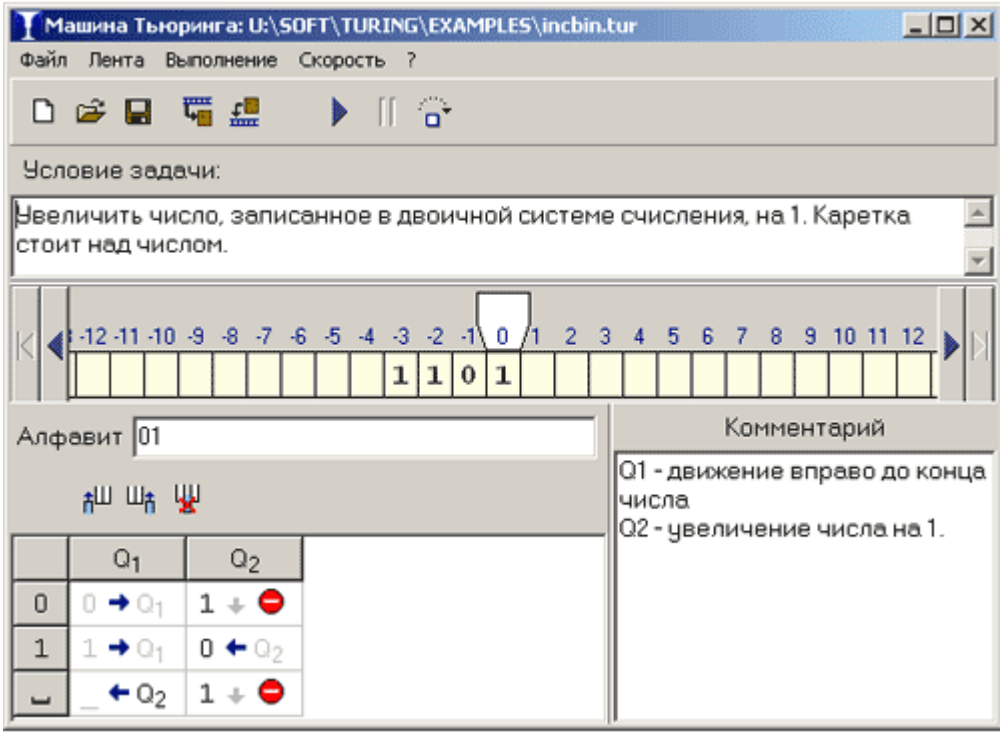


Рис. Б.2. Зовнішній вигляд програми симулятора машини Тюрінга

Таблиця Б.2

Варіанти завдань для виконання

Номер варіанта	Завдання
1	Впорядкувати послідовність ‘бабаббаа’ Спочатку літери а потім ‘б’.
2	Реверсувати 0 на 1 в рядку ‘01101010100 11’. Каретка ліворуч.
3	Змінити місцями дві послідовності ‘vvvvvv mmmm’.
4	Зменшити число в двійковій системі 11101 на 1.
5	Видалити із послідовності ‘farhad’ другий та четвертий символи.
6	Збільшити число 345 на 1.
7	Подвоїти слово ‘чотири’ поставивши між словами знак +
8	Обернути всі парні символи рядка ‘апроуек’ на 0. Каретка праворуч.
9	Переставити останній символ на початок послідовності – ‘fdfffrd’.
10	Видалити всі символи ‘а’ із послідовності ‘assallama’.
11	Розставити коми між символами в послідовності ‘dddggghhh’.

Лабораторна робота № 3
ЗДІЙСНЕННЯ ПООПЕРАЦІЙНОГО АНАЛІЗУ ЧАСОВОЇ СКЛАДНОСТІ АЛГОРИТМУ

Мета роботи

Навчитися експериментально визначати швидкість виконання математичних операцій та здійснення подальшого пооперацiйного аналізу алгоритмів.

Необхідне обладнання

Комп'ютер з операційною системою Windows версії не нижче 7.

Встановлений редактор мови програмування DEV C++.

Теоретична частина

Одним із підходів теорії алгоритмів є оцінка часу роботи алгоритму на конкретному процесорі. Така оцінка може здійснюватися декількома методами. Такими методами, зокрема є:

- пооперацiйний аналіз;
- метод Гіббсона;
- метод прямого визначення середнього часу.

Ці методи вимагають знань щодо часу виконання ряду операцій, які прийнято вважати «елементарними». До таких операцій відносять – операції додавання, віднімання, множення, ділення та ряд інших, таких, наприклад, як логічні операції або операції порівняння. Крім цього при створенні кодів програмісти використовують ряд спеціальних функцій, таких як піднесення до ступеню або вилучення кореня. Для здійснення ефективного оцінювання часу роботи того чи іншого алгоритму важливо знати час роботи кожної елементарної операції на конкретному комп'ютері. Практично, найменший час виконання процесором мають логічні операції, операції присвоювання та порівняння. Далі йдуть операції додавання, віднімання, множення, ділення. Найбільш тривалими виявляються математичні операції, що часто вимагають застосування спеціальних підпрограм і їх не називають елементарними.

До них відносяться операції вилучення кореня та піднесення до ступеня. Лабораторна робота дає можливість експериментально підтвердити це твердження.

Перевірити на практиці таке твердження в сучасних умовах досить непросто, остільки сучасні процесори мають високу потужність і виконують ряд обслуговуючих програм, тому обчислювальний ресурс витрачається нерівномірно. В цій роботі пропонується пооперaційний аналіз часу виконання програм. З цією метою пропонується один і той самий програмний код мовою С++ з різними елементарними операціями, що виконуються фіксоване число повторів у циклі. Блок-схема алгоритму цього коду подана на рис. Б.3. Після визначення експериментальним шляхом часу виконання операцій комп'ютером необхідно здійснити розрахунок часу виконання програми у відповідності до варіанту поданому в табл. Б.4 шляхом пооперaційного аналізу – підрахунку часу всіх операцій які виконуються в циклі над заданою у варіанті функцією.

Для реалізації цього коду і здійснення пооперaційного аналізу необхідно:

1. Закрити всі програми, які виконуються комп'ютером (браузери, офісні програми тощо. Перевірити чи закриті ці програми можна через Диспетчер завдань, що викликається натисканням правої кнопки мишки на кнопці «Пуск», рис. Б.4.
2. Запустити редактор мови С++ - DEV C++. Його виконання можна побачити у вікні Диспетчера завдань, рис. Б.5.
3. Вставити у вікно редактора скопійований програмний код алгоритму.
4. Скомпілювати програмний код(кнопка F9) та запустити його (кнопка F10). Дочекатися виконання програми.
5. Записати час виконання програми і розділити його на кількість повторів циклу *n*, що зазначені в коді.

6. Повторити пункти 4 та 5 змінюючи в коді рядок з елементарною операцією, для операції додавання B=A+5, віднімання B=A-5, множення – B=A*5, ділення – B=A/5, ділення по модулю B=A%5; піднесення до ступеня, квадрат, куб – B=pow(A,2), B=pow(A,3), B=A*A, B=A*A*A, вилучення квадратного кореня B=sqrt(A), B=pow(A, 0.5), вилучення кубічного кореня B=pow(A,1/3), B=cbrt(A).

7. Занести час виконання кожної операції в табл. Б.3.

Розрахувати час виконання програми за варіантом поданим в табл. Б.4.

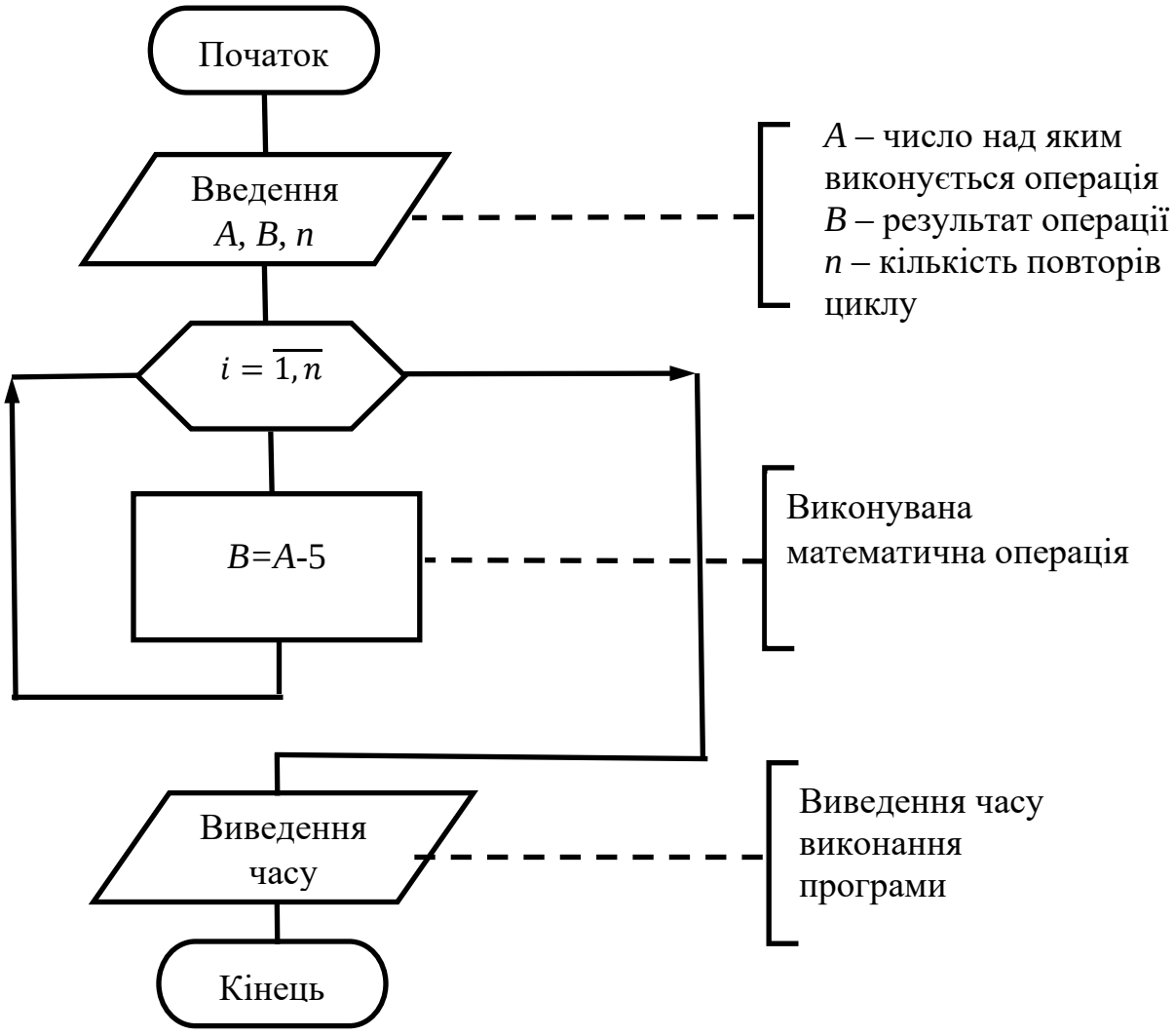


Рис Б.3. Блок-схема алгоритму коду

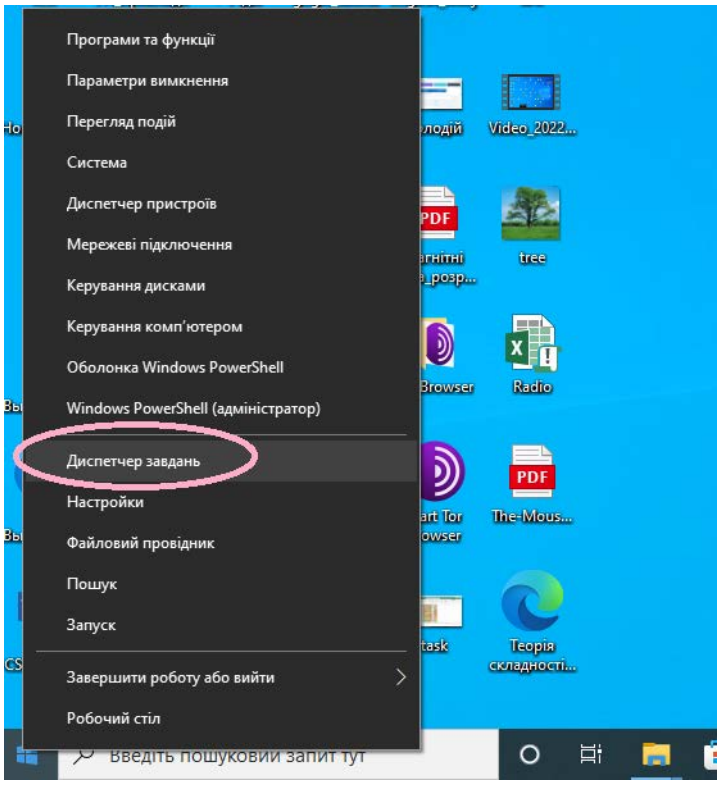


Рис. Б.4. Вікно для завантаження Диспетчера завдань

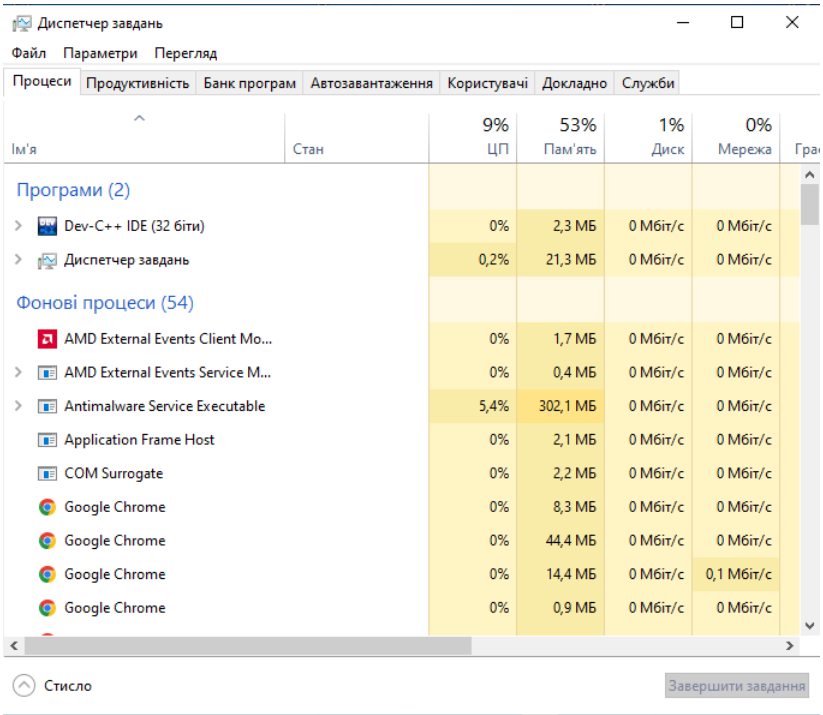


Рис. Б.5. Відкритий диспетчер завдань

Код алгоритму

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <math.h>//Бібліотека спеціальних математичних функцій
using namespace std;
int main()
{
  setlocale(LC_ALL, "ukr");
  int A=10,B;                    //Ініціалізація змінних A та B
  long long int n=4000000000;        //Ініціалізація змінної n
  srand(time(NULL));
  for (long long int i = 0; i < n; i++)//Цикл від 0 до n
  {
        B = A-5; //Операція віднімання.
                 //Замість цього рядка треба ставити
                 //рядки з іншими операціями.
        }        //Кінець циклу
  cout << endl;//Виведення часу
  return 0;
}
```

Таблиця Б.3

Час виконання математичних операцій

№	Операція	Час виконання, мкс
1	Додавання	
2	Віднімання	
3	Ділення	
4	Ділення по модулю	
5	Множення	
6	Квадрат через множення – B=A*A	
7	Квадрат через функцію – B=pow(A,2)	
8	Куб через множення – B=A*A*A	
9	Куб через функцію – B=pow(A,3)	
10	Корінь квадратний через ступінь – B=pow(A,0.5)	
11	Корінь квадратний через функцію – B=sqrt(A)	
12	Корінь кубічний через ступінь – B=pow(A,1/3)	
13	Корінь кубічний через функцію – B=cbrt(A)	

Таблиця Б.4

Варіанти для розрахунку часу виконання програми шляхом пооперaційного аналізу

№	Розрахункове співвідношення	Кількість циклів виконання	Розрахований час виконання, с
1	$y = x^2 + \sqrt[2]{41 \cdot x}/5$	13500	
2	$y = x^3 + 3 \cdot x^2 + 2 \cdot x - 5$	21000	
3	$y = \sqrt[3]{x^2} + x^4$	18000	
4	$z = x^3 + 2 \cdot y^3$	30000	
5	$y = 2 \cdot x^2 - 4 \cdot x + 5$	12000	
6	$y = \frac{x^3}{\sqrt{2 \cdot x}}$	14000	
7	$y = \sqrt[4]{x^2}$	20000	
8	$y = \frac{x^5}{13}$	24000	
9	$y = \sqrt[2]{x} + \sqrt[3]{x}$	12800	
10	$y = 5 \cdot x^3 + 4 \cdot x$	14000	
11	$y = 5 \cdot x^2 + \sqrt{x}$	16000	
12	$y = (x^3 - 7 \cdot x^2) mod 3$	29000	

Лабораторна робота №4
АЛГОРИТМИ КІЛЬКІСНО - ЗАЛЕЖНІ ЗА ТРУДОМІСТКІСТЮ

Мета роботи

Вивчити роботу кількісно-залежних за трудомісткістю алгоритмів на прикладі операцій з масивами

Необхідне обладнання

Комп'ютер з операційною системою Windows версії не нижче 7.

Встановлений редактор мови програмування DEV C++.

Теоретична частина

Кількісно залежні з трудомісткістю алгоритми час виконання залежить від розміру даних на вході і не залежить від конкретних значень. Одним із прикладів таких алгоритмів є алгоритм множення в масивах. Блок-схема такого алгоритму в якому реалізується множення чисел у двомірному масиві, показаний на рис. Б.6. Результатом такого множення є заповнений двомірний масив в якому розміщені значення чисел із таблиці множення. Оскільки час виконання операції множення надзвичайно короткий, то для його відстеження використовується додатковий зовнішній цикл $i = \overline{0,10}$ кількість ітерацій якого можна змінювати.

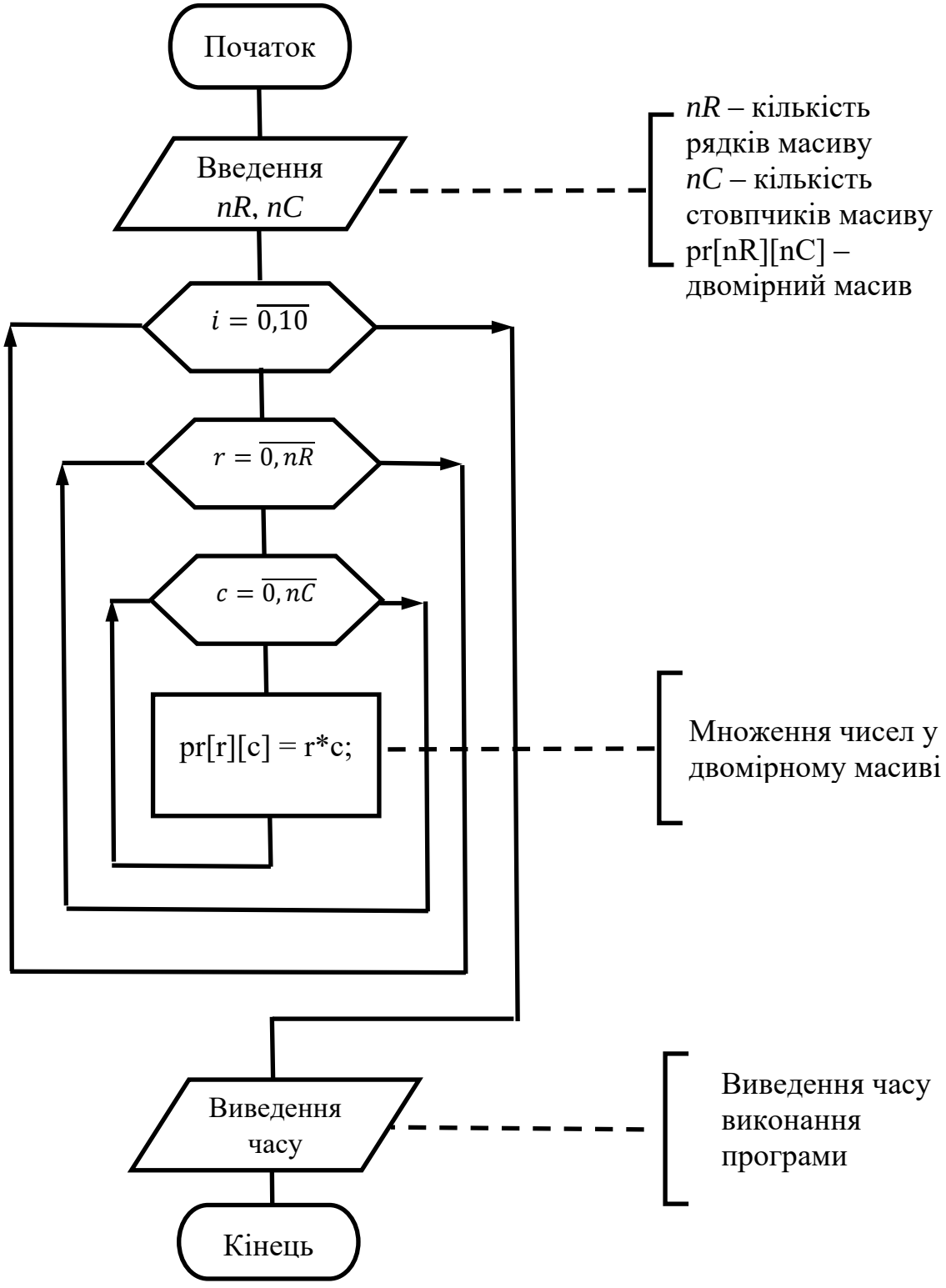


Рис. Б.6. Блок-схема алгоритму множення чисел у двомірному масиві

Код програми множення чисел у двомірному масиві

```
#include <iostream>

int main()
{
   // Оголошуємо масив розмірності nRXnC
   const int nR = 101;
   const int nC = 101;
   int pr[nR][nC] = { 0 };

 for (int i=0; i<10; i++)//Зовнішній цикл
 {

   // Створюємо таблицю множення
   for (int r = 0; r < nR; ++r)
      for (int c = 0; c < nC; ++c)
         pr[r][c] = r*c;//Множення чисел і розміщення їх у масиві
         //pr[r][c] = 0 * 0;
}
   // Виводимо таблицю множення - закоментовано
/*    for (int r = 1; r < nR; ++r)
   {
      for (int c = 1; c < nC; ++c)
         std::cout << pr[r][c] << "\t";

      std::cout << '\n';
   }
*/
   return 0;
}
```

Порядок виконання роботи

Закрити всі сторонні програми на комп’ютері (браузери, зайві офісні програми які не використовуються тощо).

Відкрити редактор DEV C++.

Створити вихідний файл – Ctrl+N.

Скопіювати і вставити код програми множення чисел у вікно редактора.

Розкоментувати блок коду

```
/*    for (int r = 1; r < nR; ++r)
    {
        for (int c = 1; c < nC; ++c)
            std::cout << pr[r][c] << "\t";

        std::cout << '\n';
    }
*/
```

Перевірити коректність роботи програми спочатку скомпілювавши її – F9, потім запустивши F10. Результат повинен відповідати рис. Б.7.

```
3007    3104    3201    3298    3395    3492    3589    3686    3783    3880    3977    4074    4171    4268    4365
4462    4559    4656    4753    4850    4947    5044    5141    5238    5335    5432    5529    5626    5723    5820
5917    6014    6111    6208    6305    6402    6499    6596    6693    6790    6887    6984    7081    7178    7275
7372    7469    7566    7663    7760    7857    7954    8051    8148    8245    8342    8439    8536    8633    8730
8827    8924    9021    9118    9215    9312    9409    9506    9603    9700
98      196     294     392     490     588     686     784     882     980     1078    1176    1274    1372    1470
1568    1666    1764    1862    1960    2058    2156    2254    2352    2450    2548    2646    2744    2842    2940
3038    3136    3234    3332    3430    3528    3626    3724    3822    3920    4018    4116    4214    4312    4410
4508    4606    4704    4802    4900    4998    5096    5194    5292    5390    5488    5586    5684    5782    5880
5978    6076    6174    6272    6370    6468    6566    6664    6762    6860    6958    7056    7154    7252    7350
7448    7546    7644    7742    7840    7938    8036    8134    8232    8330    8428    8526    8624    8722    8820
8918    9016    9114    9212    9310    9408    9506    9604    9702    9800
99      198     297     396     495     594     693     792     891     990     1089    1188    1287    1386    1485
1584    1683    1782    1881    1980    2079    2178    2277    2376    2475    2574    2673    2772    2871    2970
3069    3168    3267    3366    3465    3564    3663    3762    3861    3960    4059    4158    4257    4356    4455
4554    4653    4752    4851    4950    5049    5148    5247    5346    5445    5544    5643    5742    5841    5940
6039    6138    6237    6336    6435    6534    6633    6732    6831    6930    7029    7128    7227    7326    7425
7524    7623    7722    7821    7920    8019    8118    8217    8316    8415    8514    8613    8712    8811    8910
9009    9108    9207    9306    9405    9504    9603    9702    9801    9900
100     200     300     400     500     600     700     800     900     1000    1100    1200    1300    1400    1500
1600    1700    1800    1900    2000    2100    2200    2300    2400    2500    2600    2700    2800    2900    3000
3100    3200    3300    3400    3500    3600    3700    3800    3900    4000    4100    4200    4300    4400    4500
4600    4700    4800    4900    5000    5100    5200    5300    5400    5500    5600    5700    5800    5900    6000
6100    6200    6300    6400    6500    6600    6700    6800    6900    7000    7100    7200    7300    7400    7500
7600    7700    7800    7900    8000    8100    8200    8300    8400    8500    8600    8700    8800    8900    9000
9100    9200    9300    9400    9500    9600    9700    9800    9900    10000

--------------------------------
Process exited after 1.191 seconds with return value 0
Press any key to continue . . .
```

Рис. Б.7. Результат перевірки.

Закоментувати блок коду

```
for (int r = 1; r < nR; ++r)
{
    for (int c = 1; c < nC; ++c)
        std::cout << pr[r][c] << "\t";

    std::cout << '\n';
}
```

Скомпілювати і запустити програму ще раз. Час виконання занести в табл. Б.5. Змінити рядок коду for (int i=0; i<10; i++)//Зовнішній цикл збільшивши кількість повторів в циклі в 10 разів → for (int i=0; i<100; i++)//Зовнішній цикл.

Повторно скомпілювати і запустити програму. Знов занести час до табл. Б.5.

Відповідно до кількості повторів циклу зазначених в табл. Б.5 збільшувати кількість повторів в циклі і повторювати процедуру компіляції і запуску програми заносячи значення часу у відповідні рядки табл. Б.5.

Змінити рядки коду програми

```
pr[r][c] = r*c;//Множення чисел і розміщення їх у масиві
//pr[r][c] = 0 * 0;
```

на наступні

```
//pr[r][c] = r*c;//Множення чисел і розміщення їх у масиві
pr[r][c] = 0 * 0;
```

Повторити процедуру компілювання і запуску змінюючи кількість повторів в циклі відповідно до табл. Б.5. Після заповнення таблиці побудувати два графіки залежності між кількістю повторів циклу і часом виконання у двох випадках. Зробити висновки.

Таблиця Б.5

Результат виконання коду програми множення чисел у двомірному масиві

№	Кількість повторів циклу	Час виконання програми для фрагменту коду pr[r][c] = r*c;	Час виконання програми для фрагменту коду pr[r][c] = 0*0;
1	10		
2	100		
3	1000		
4	10000		
5	100000		
6	1000000		

Лабораторна робота №5
АЛГОРИТМИ ПАРАМЕТРИЧНІ ЗА ТРУДОМІСТКІСТЮ

Мета роботи

Вивчити роботу параметричних за трудомісткістю алгоритмів на прикладі алгоритму розрахунку функції косинуса, арккосинуса та тангенса із заданою точністю

Необхідне обладнання

Комп'ютер з операційною системою Windows версії не нижче 7.

Встановлений редактор мови програмування DEV C++.

Теоретична частина

В параметрично-залежних за трудомісткістю алгоритмах трудомісткість визначена конкретним значенням оброблюваних слів пам'яті. Одним із прикладів параметрично-залежних алгоритмів є розрахунок математичних функцій з використанням ряду Тейлора. Ряд Тейлора дозволяє подавати математичні функції у вигляді суми доданків. Теоретично така сума повинна бути нескінченною. Формула ряду Тейлора має вигляд:

$$\sum_{n=0}^{\infty} \frac{f^{(n)} \cdot (a)}{n!} \cdot (x-a)^n \qquad (Б.1)$$

Наприклад, для функції синуса ряд Тейлора буде записаний наступним чином:

$$sinx = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \cdots = \sum_{n=0}^{\infty} (-1)^n \cdot \frac{x^{2 \cdot n+1}}{(2 \cdot n+1)!} \qquad (Б.2)$$

Наявність відповідної суми доданків дозволяє обчислювати функції із заданою точністю. Від цієї точності залежить час виконання алгоритму.

Блок-схема алгоритму розрахунку функції косинуса з використанням ряду Тейлора заданої точності показана на рис. Б.8.

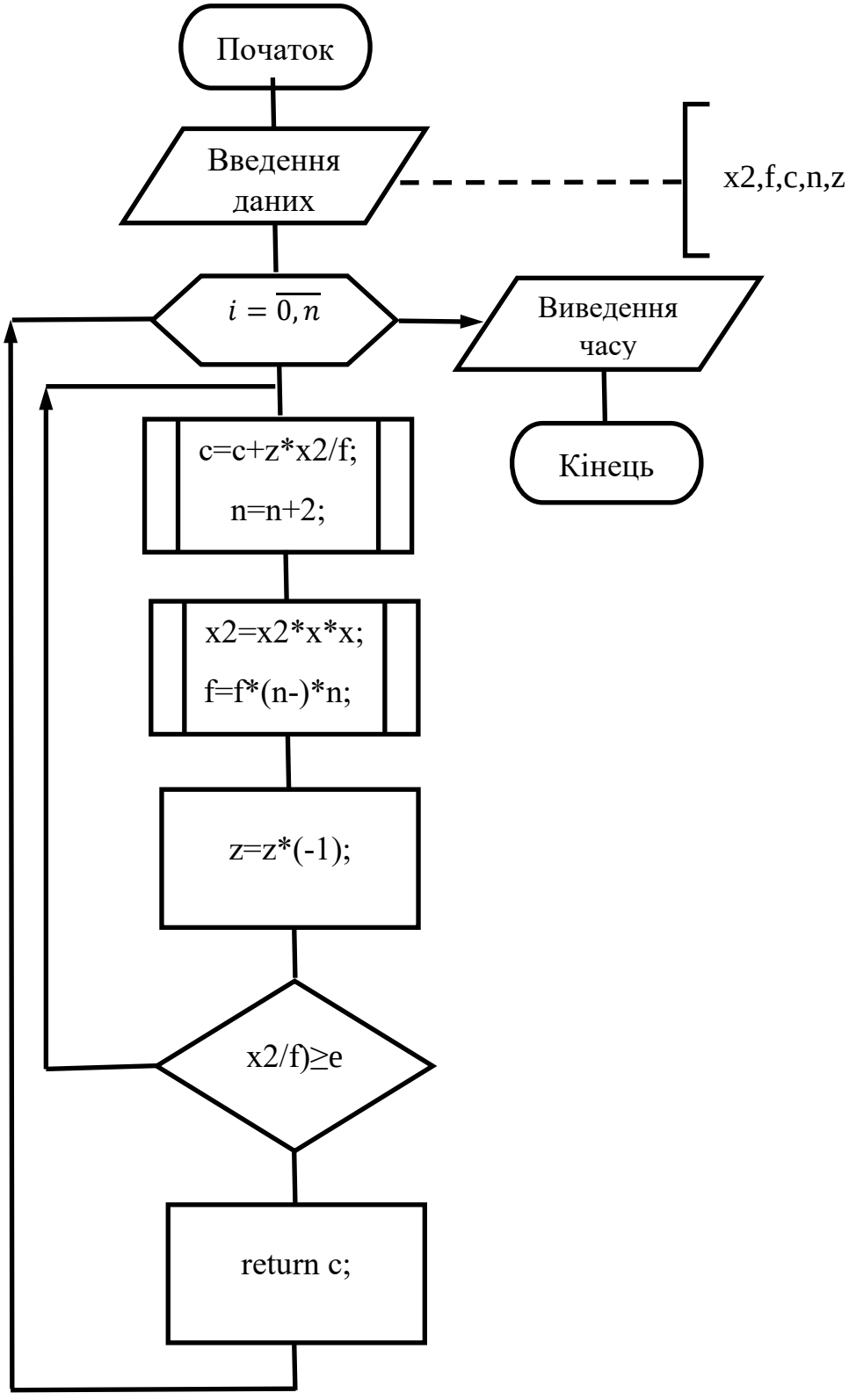


Рис. Б.8. Блок - схема алгоритму розрахунку функції косинуса з використанням ряду Тейлора

Порядок виконання роботи

1. Закрити всі програми, які виконуються комп'ютером (браузери, офісні програми тощо).

2. Запустити редактор мови C++ - DEV C++.

3. Вставити у вікно редактора скопійований програмний код для функції *cos x*.

4. Задати точність обчислення (вказана в коді) у відповідно до табл. Б.6 і провести обчислення часу виконання програми. Результат занести в табл. Б.6.

5. Повторити пункти 3 та 4 для всіх значень точності та функцій *arccos x* та *tan x*.

6. Побудувати гістограми для часу виконання алгоритмів обчислення функцій.

7. Зробити висновки.

Код програми обчислення функції cos x із заданою точністю

```
#include <iostream>
#include <cmath>
using namespace std;
double cos (double x, double e) {

   double x2,c,n,f,z;
   x2= x*x;
   f= 2;
   c= 1;
   n= 2;
   z= -1;
   while ((x2/f)>=e) {
      c=c+z*x2/f;
      n=n+2;
      x2=x2*x*x;
      f=f*(n-1)*n;
      z=z*(-1);
   }
   //cout<<x<<endl;
   return c;
}
```

```
int main() {
   double x,e;
   //cin>>x;
   //cin>>e;
   x = 5;
   e = 0.1;//точність обчислення
for (int j=0; j<10; j++)
{

for (int i=0; i<10000000; i++)
{
   cos(x,e);
}
}
 cout<<cos(x,e)<<endl;
   return 0;
}
```

Код програми для обчислення функції *arccos x* із заданою точністю

```
#include <iostream>
#include <cmath>
#define PI 3.1415926
using namespace std;

double f(double e, double x){


  int i = 0;
  double p = x;
  double s = x;
  while (p > e){
        p*=(x*x*(2*i + 1)*(2*i + 1))/(2*(i+1)*(2*i + 3));
        s+=p;
        i++;
  }
  return s;
}

int main() {
  double e;
  double x;
//        cin >> e >> x;
e=0.0000001; //точність обчислення
x=0.5;
```

```
for (int k=0; k<100; k++)
{

for (int j=0; j<10000000; j++)
{
PI/2 - f(e, x);
}
}
cout << PI/2 - f(e, x) << endl;
return 0;
}
```

Код програми для обчислення функції *tan x* із заданою точністю

```
#include <iostream>
#include <cmath>
using namespace std;

double tangent(double x, double accuracy) {
  int i = 1;
  double cos = 1, sin = x, intermediateValueCos = 1, intermediateValueSin =
x;
  x = fmod(x, M_PI);
  while (fabs(intermediateValueCos *= - x * x / (2 * i * (2 * i - 1))) > accuracy
        && fabs(intermediateValueSin *= - x * x / (2 * i * (2 * i + 1))) >
accuracy) {
        cos += intermediateValueCos;
        sin += intermediateValueSin;
        i++;
  }
  return sin / cos;
}

int main() {
  double x, accuracy;
  x = 200; accuracy = 0.0001;//точність обчислення

    for (int i=0; i <100000000; i++)
    {
        tangent(x, accuracy);
  }
  cout << tangent(x, accuracy);
  return 0;
}
```

Код програми для обчислення функції *tan x* із заданою точністю

Таблиця Б.6

Результати виконання кодів програм обчислення функцій з використанням рядів Тейлора

№	Точність обчислення - E	Час виконання, с		
		cos x	*arccos x*	*tan x*
1	0,1			
2	0,0001			
3	0,0000001			
4	0,000000001			

Лабораторна робота №6
АЛГОРИТМИ КІЛЬКІСНО – ПАРАМЕТРИЧНІ ЗА ТРУДОМІСТКІСТЮ

Мета роботи

Вивчити роботу кількісно-параметричних за трудомісткістю алгоритмів на прикладі алгоритму швидкого сортування

Необхідне обладнання

Комп'ютер з операційною системою Windows версії не нижче 7.

Встановлений редактор мови програмування DEV C++.

Теоретична частина

В порядково - залежних за трудомісткістю алгоритмах час виконання програмного коду залежить головним чином від порядку розташування елементів на вході. Ці алгоритми відносять до параметрично залежних. В параметрично залежних за трудомісткістю алгоритмах ця трудомісткість визначається не розмірністю входу, а конкретними значеннями оброблюваних вхідних даних.

До таких алгоритмів зокрема належать окремі алгоритми пошуку максимальних або мінімальних елементів масиву, деякі алгоритми сортування та ряд інших.

Слід зазначити, що в чистому вигляді порядково - залежні алгоритми зустрічаються не часто. На практиці вхідні дані варіюються за розміром і більшість алгоритмів можна віднести до кількісно-параметричних. Тобто залежність спостерігається як від кількості вхідних даних так і від порядку їх розташування. Розглядуваний приклад алгоритму швидкого сортування наочно демонструє як працюють такі алгоритми. Ідея алгоритму полягає в переставлянні елементів масиву таким чином, щоб його можна було розділити на дві частини і кожний елемент з першої частини був не більший за будь-який елемент з другої. Впорядкування кожної частини здійснюється шляхом рекурсії.

Алгоритм швидкого сортування працює швидше за інші алгоритми сортування але лише в тих випадках, коли порядок розташування елементів вхідного масиву є неоднорідним. Тобто кількість однакових елементів є незначною.

В противному випадку швидкість сортування суттєво знижується. В найкращому випадку алгоритм виконує $O(n\ log\ n)$ операцій. В найгіршому – $O(n^2)$.

Порядок виконання роботи

1. Виконати всі умови для коректного виконання роботи(вимкнути всі програми, які не стосуються лабораторної роботи).
2. Скопіювати код алгоритму до редактора Dev C++ та скомпілювати його.
3. Запустити програму кнопкою F10. Після виконання записати час виконання в табл. Б.7 – перший рядок.
4. Закоментувати рядок з кращим випадком і розкоментувати з гіршим. Відкомпілювати і запустити програму. Час виконання записати в другий рядок табл. Б.7.
5. Повторити пункти 3 та 4 розкоментовуючи та закоментовуючи рядки. Кількість елементів в масиві при цьому зменшити до 22. Заповнити табл. Б.7. Зробити висновки.

Код алгоритму швидкого сортування

```
#include <iostream>
using namespace std ;

/* Функція швидкого сортування */
void QuickSort (int Array[], /* масив для сортування */
          unsigned int N, /* розмір масиву */
          int L, /* ліва межа сортування */
          int R) /* права межа сортування */
{
   int iter = L ,
      jter = R ;

   int middle = (R+L)/2 ;

   int x = Array [middle] ;
   int w ;

   do
   {
      while (Array[iter]<x)
      { iter ++ ; }

      while (x<Array[jter])
      { jter -- ; }

      if (iter<=jter)
      {
        w = Array [iter];
        Array [iter] = Array [jter] ;
        Array [jter] = w ;

        iter ++ ;
        jter -- ;
      }
   }
   while (iter<jter) ;

   if (L<jter)
   { QuickSort (Array, N, L, jter) ; }

   if (iter<R)
   { QuickSort (Array, N, iter, R) ; }
```

```
}

int main (int argc, char** argv) /* головна функція програми */
{
   /* масив даних */
//    int Array [] = {0, 1, 2, 3, 4, 5, 6, 22, 8, 9, 10, 11, 0, 13, 14, 1, 16, 17, 1, 19, 2, 21, 22, 23, 2, 25, 0, 27, 22, 29, 22, 31, 32, 1, 34, 40, 36, 2, 38, 3, 40, 42, 42, 43, 44, 22} ;  //гірший випадок
     int Array [] = {0, 2, 1, 4, 30, 5, 6, 7, 8, 29, 10, 28, 12, 13, 14, 16, 15, 17, 18, 19, 20, 21, 23, 22, 24, 25, 26, 27, 11, 9, 3, 31, 32, 33, 34, 35, 36, 37, 38, 40, 39, 41, 42, 43, 44, 45} ; //кращий випадок

   int N = sizeof (Array) / sizeof (Array[0]) ; /* дізнаємося довжину масиву */

   int iter ; /* бігунки-ітератори для перебору масиву */

   /* виводимо масив на екран для наглядності */
   for (iter=0; iter<N; iter++)
   { cout << Array[iter] << " " ; }
   cout << endl ;

   for (int i=0; i<100000000; i++)
   {

   QuickSort (Array, N, 0, N-1) ;

   }
   /* виводимо масив на екран для наглядності */
   for (iter=0; iter<N; iter++)
   { cout << Array[iter] << " " ; }
   cout << endl ;

   return 0 ; /* Вихід з програми */
}
```

Таблиця Б.7

Час виконання алгоритму швидкого сортування для різної кількості елементів та відмінного порядку їх розташування в масиві

№	Опис вхідного масиву	Час виконання, с
1	45 елементів без повторів однакових значень – кращий випадок	
2	45 елементів з повторами однакових значень – гірший випадок	
3	22 елементи без повторів однакових значень – кращий випадок	
4	22 елементи з повторами однакових значень – гірший випадок	

Лабораторна робота №7
ОЦІНЮВАННЯ ЧАСУ ВИКОНАННЯ АЛГОРИТМУ НАЇВНОГО МНОЖЕННЯ ТА АЛГОРИТМУ КАРАЦУБИ

Мета роботи

Здійснити порівняння швидкодії роботи комп'ютерних програм на основі алгоритму наївного множення та алгоритму Карацуби

Необхідне обладнання

Комп'ютер з операційною системою Windows версії не нижче 7.

Встановлений редактор мови програмування DEV C++.

Теоретична частина

Наївне множення відомий зі школи метод який ще має назву множення у стовпчик. Довгий час вважалося, що єдиним методом множення може бути тільки цей метод, нотація Ландау для якого становить $O(n^2)$. Однак в 60-х роках XX сторіччя математик Карацуба запропонував метод множення чисел для якого нотація Ландау складала $O(n^{1,58})$. Його роботи дали поштовх для дослідження методів швидкого множення з використанням різних математичних прийомів. Насьогодні існує декілька методів швидкого множення які використовуються в різних задачах, коли виникає необхідність множення великих чисел. В цій роботі експериментальним шляхом робиться перевірка швидкості виконання програм, які ґрунтуються на алгоритмі наївного множення та множення Карацуби. Це дасть змогу зрозуміти сутність нотації Ландау.

Порядок виконання роботи

1. Відкрити програму Dev C++, створити вихідний файл і скопіювати у вікно редактора код програми для алгоритму наївного множення

Код програми для алгоритму наївного множення

```
#include <iostream>
#include <string>
#define OVERFLOW 2
#define ROW b_len
#define COL a_len+b_len+OVERFLOW
using namespace std;
int getCarry(int num) {
   int carry = 0;
   if(num>=10) {
      while(num!=0) {
         carry = num %10;
         num = num/10;
      }
   }
   else carry = 0;
   return carry;
}

int num(char a) {
   return int(a)-48;
}

string mult(string a, string b) {
      string ret;
      int a_len = a.length();
      int b_len = b.length();
      int mat[ROW][COL];
      for(int i =0; i<ROW; ++i) {
         for(int j=0; j<COL; ++j) {
            mat[i][j] = 0;

         }
      }

      int carry=0, n,x=a_len-1,y=b_len-1;
      for(int i=0; i<ROW; ++i) {
         x=a_len-1;
         carry = 0;
         for(int j=(COL-1)-i; j>=0; --j) {
            if((x>=0)&&(y>=0))  {
               n = (num(a[x])*num(b[y]))+carry;
               mat[i][j] = n%10;
```

```
            carry = getCarry(n);
          }
          else if((x>=-1)&&(y>=-1)) mat[i][j] = carry;
          x=x-1;
        }
        y=y-1;
      }

      carry = 0;
      int sum_arr[COL];
      for(int i =0; i<COL; ++i) sum_arr[i] = 0;
      for(int i=0; i<ROW; ++i) {
        for(int j=COL-1; j>=0; --j) {
          sum_arr[j] += (mat[i][j]);
        }
      }
      int temp;
      for(int i=COL-1; i>=0; --i) {
        sum_arr[i] += carry;
        temp = sum_arr[i];
        sum_arr[i] = sum_arr[i]%10;
        carry = getCarry(temp);
      }

      for(int i=0; i<COL; ++i) {
        ret.push_back(char(sum_arr[i]+48));
      }

      while(ret[0]=='0'){
        ret = ret.substr(1,ret.length()-1);
      }
      return ret;
}

void printhuge(string a) {
  // cout<<"\n";
  /* for(string::iterator i = a.begin(); i!=a.end(); ++i) {
      cout<<*i;
   }
   */
}

int main() {
```

```
    string a,b;
    a = "12345";
    b = "67890";
    //cin>>a>>b;
    for (int i=0; i<9302500;i++)//Тут встановлювати кількість повторів
    {
      printhuge(mult(a,b));
    }
    return 0;

}
```

2. В програмному коді встановити кількість повторів циклу у рядку **for (int i=0; i<9302500;i++)** відповідно до значень першого рядка таблиці Б.8 (стовпчик «Кількість повторів циклу»).

3. Відкомпілювати кнопкою F9 та запустити кнопкою F10 програмний код. Час виконання програми записати в таблицю Б.8 у стовпчик з назвою «Наївне множення».

4. Повторити дії пунктів 2 та 3 для всіх рядків таблиці Б.8 заповнивши стовпчик «Наївне множення» значеннями часу виконання програми.

5. Відкрити нове вікно програми Dev C++, створити вихідний файл і скопіювати у вікно редактора код програми для алгоритму Карацуби.

Код програми для алгоритму Карацуби

```
#include <stdio.h>
#include <math.h>
#include <string>
#include <iostream>
#include <cstdlib>
#include <ctime>

// Get size of the numbers
int getSize(long num)
{
   int count = 0;
   while (num > 0)
   {
      count++;
      num /= 10;
   }
   return count;
}
long karatsuba(long X, long Y)
{
   // Base Case
   if (X < 10 && Y < 10)
      return X * Y;

   // determine the size of X and Y

   int size = fmax(getSize(X), getSize(Y));

   // Split X and Y
   int n = (int)ceil(size / 2.0);
   long p = (long)pow(10, n);
   long a = (long)floor(X / (double)p);
   long b = X % p;
   long c = (long)floor(Y / (double)p);
   long d = Y % p;

   // Recur until base case
   long ac = karatsuba(a, c);
   long bd = karatsuba(b, d);
   long e = karatsuba(a + b, c + d) - ac - bd;
```

```
    // return the equation
    return (long)(pow(10 * 1L, 2 * n) * ac + pow(10 * 1L, n) * e + bd);
}

int main()
{
    int a,b;
    srand(time(NULL));
    for (int i=0; i<9302500;i++)
    {
      a = 12345;
    b = 67890;
      //a = 20000 + rand() % 50001;//випадкове число від 20000 до 50001
      //b = 20000 + rand() % 50001;//випадкове число від 20000 до 50001
      //a = 4449;
      //b = 5559;
      karatsuba(a, b);
    }
    std::cout << karatsuba(a, b);
}
```

6. Виконати дії які описані в пунктах 2 та 3 для алгоритму Карацуби, заповнивши стовпчик таблиці Б.8 з назвою “Алгоритм Карацуби”.

7. Побудувати графік і зробити висновки.

Таблиця Б.8

Вихідні дані для роботи над алгоритмами

№	Кількість повторів циклу	Наївне множення	Алгоритм Карацуби
1	1000		
2	2000		
3	5000		
4	10000		
5	20000		
6	50000		
7	100000		
8	200000		
9	500000		
10	1000000		
11	5000000		
12	9302500		

Лабораторна робота №8
НЕДЕТЕРМІНОВАНІ АЛГОРИТМИ. ОЦІНЮВАННЯ ЧАСУ ВИКОНАННЯ ПРОГРАМ НА ОСНОВІ АЛГОРИТМУ ЕВКЛІДА

Мета роботи

Дослідити особливості роботи недетермінованих алгоритмів на через порівняння швидкодії роботи комп'ютерних програм на основі алгоритму Евкліда.

Необхідне обладнання

Комп'ютер з операційною системою Windows версії не нижче 7.

Встановлений редактор мови програмування DEV C++.

Теоретична частина

Алгоритм Евкліда – метод знаходження найбільшого спільного дільника двох цілих чисел. Найбільший спільний дільник це число, що ділить обидва задані числа без остачі. Цей алгоритм являє собою один із найстаріших відомих алгоритмів який використовується до нашого часу. Реалізація цього алгоритму можлива декількома методами. В роботі використані метод віднімання та метод ділення по модулю. Блок-схеми цього алгоритму для зазначених методів наведені на рис. Б.9 та рис. Б.10. Подібність методів та практично ідентична структура реалізації алгоритму цими методами дозволяє порівнювати швидкості виконання алгоритмів, що побудовані на різних підходах до вирішення однієї і тієї ж задачі.

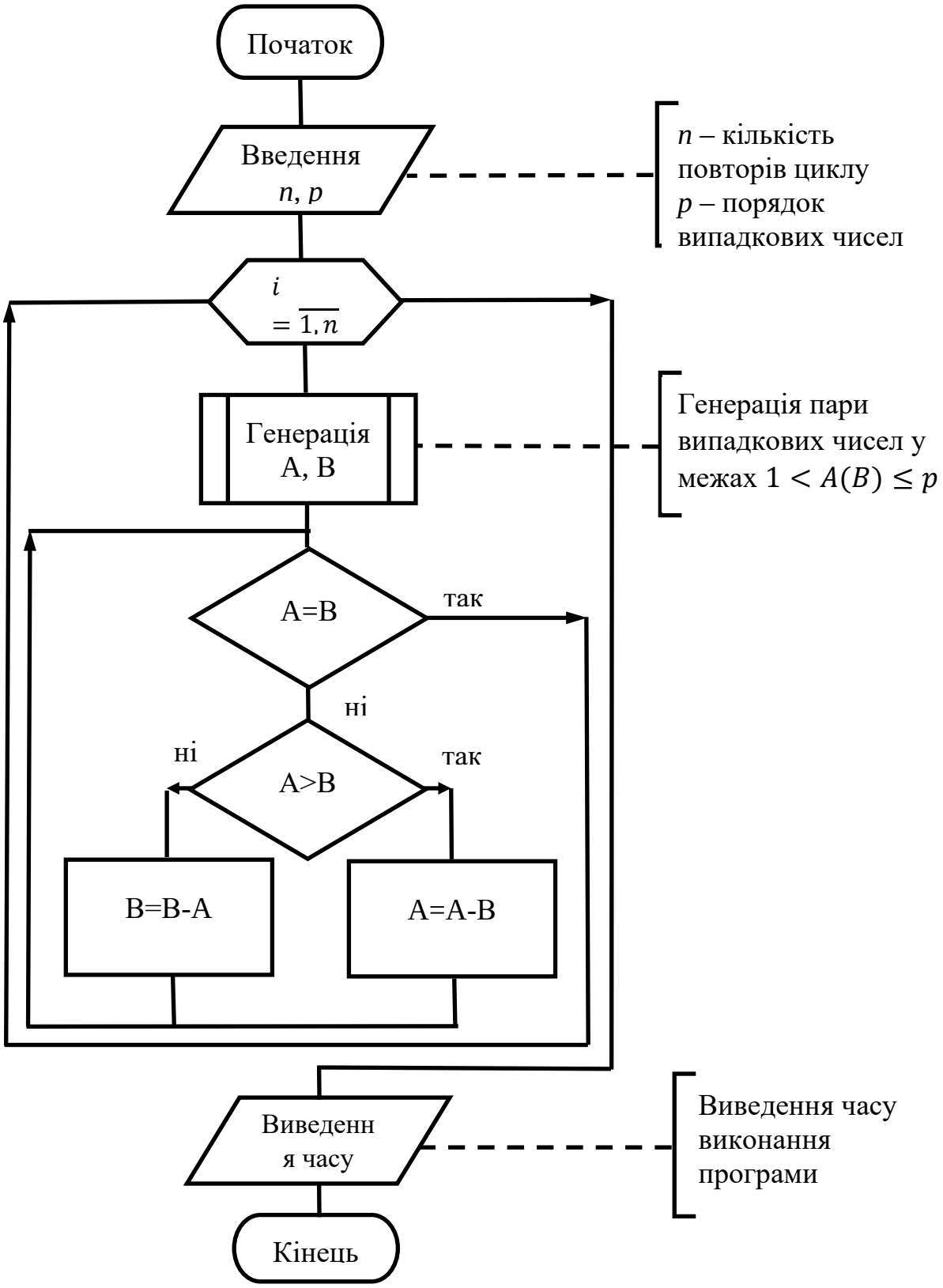


Рис. Б.9. Блок - схема алгоритму для визначення часу виконання алгоритму Евкліда що використовує операцію віднімання

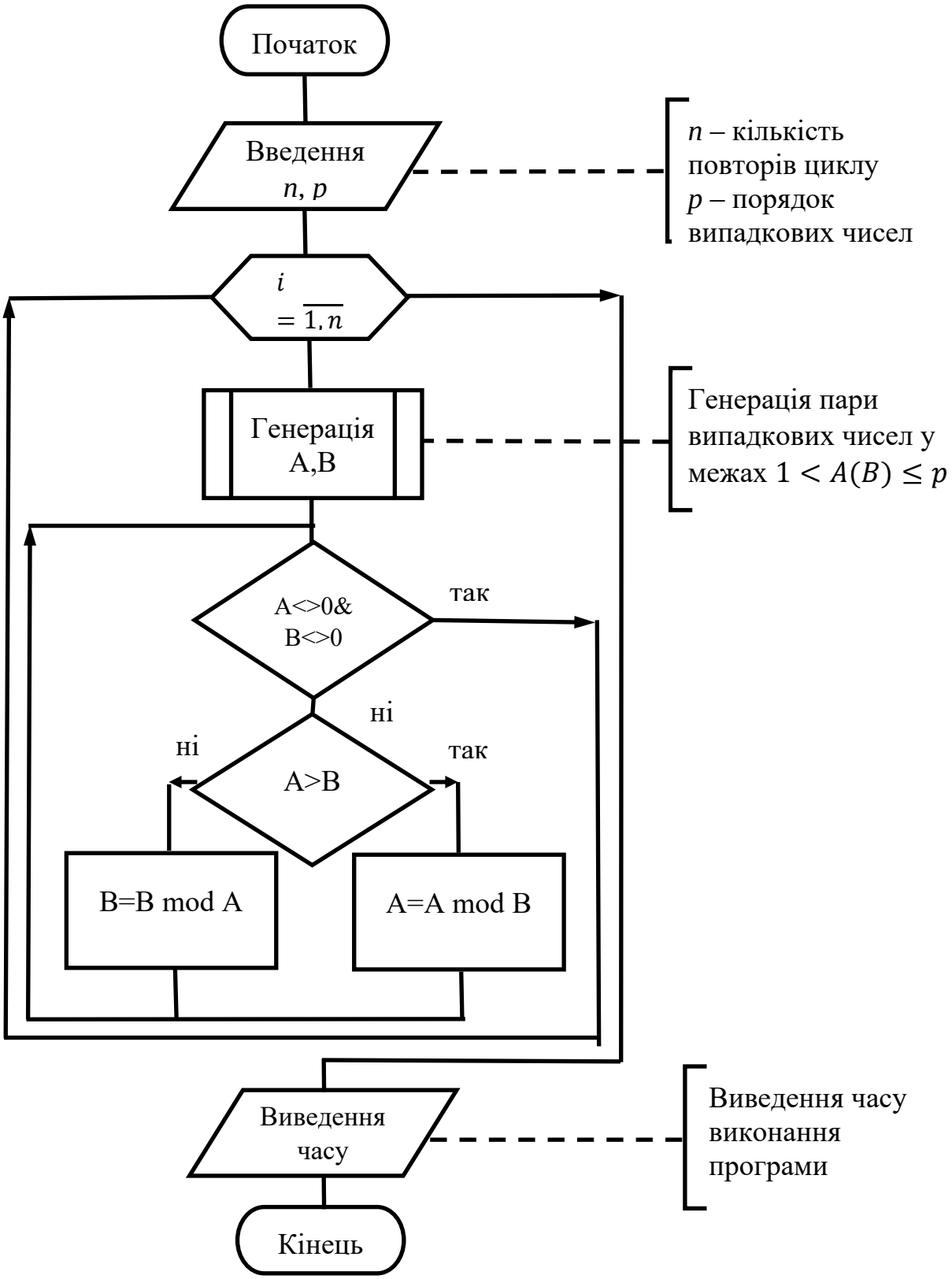


Рис. Б. 10. Блок - схема алгоритму для визначення часу виконання алгоритму Евкліда що використовує операцію ділення по модулю

Таблиця Б.9

Значення розрядності випадкових чисел та кількостей повторів циклів для визначення швидкостей виконання алгоритмів Евкліда

№	Кількість повторів в циклі	Розрядність випадкових чисел	Час виконання, с	
			Алгоритм Евкліда-віднімання	Алгоритм Евкліда-ділення по модулю
1	10	100		
2	10	1000		
3	10	10000		
4	10	100000		
5	10	1000000		
6	100	100		
7	100	1000		
8	100	10000		
9	100	100000		
10	100	1000000		
11	1000	100		
12	1000	1000		
13	1000	10000		
14	1000	100000		
15	1000	1000000		
16	10000	100		
17	10000	1000		
18	10000	10000		
19	10000	100000		
20	10000	1000000		
21	100000	100		
22	100000	1000		
23	100000	10000		
24	100000	100000		
25	100000	1000000		
26	1000000	100		
27	1000000	1000		
28	1000000	10000		
29	1000000	100000		
30	1000000	1000000		
31	10000000	100		
32	10000000	1000		
33	10000000	10000		
34	10000000	100000		
35	10000000	1000000		

Порядок виконання роботи

1. Відкрити програму Dev C++, створити вихідний файл і скопіювати у вікно редактора код програми для алгоритму Евкліда, що використовує операцію віднімання.

Код програми для алгоритму Евкліда з операцією віднімання

```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;
int main()
{
  setlocale(LC_ALL, "ukr");
  int A, B;
  srand(time(NULL));
  for (int i = 0; i < 1000; i++)//Кількість повторів циклу
  {
          //Генерація 2-х випадкових чисел
          A = 1+rand() % 1000;          // Перше число - A
          B = 1+rand() % 1000;          // Друге число - B
          //       Алгоритм Евкліда
                  while (A != B)
      {
          if (A>B) { A = A-B; }  else {B = B-A; }//Використання віднімання
        }
  }
  cout << endl;
  return 0;
}
```

2. В програмному коді встановити кількість повторів циклу – 10, розрядність випадкових чисел – 100, що відповідає першому рядку таблиці Б.9. 2. Скомпілювати програму кнопкою F9. Запустити програму кнопкою F10. Розрядність випадкових чисел треба змінювати як для числа A так і для числа B.

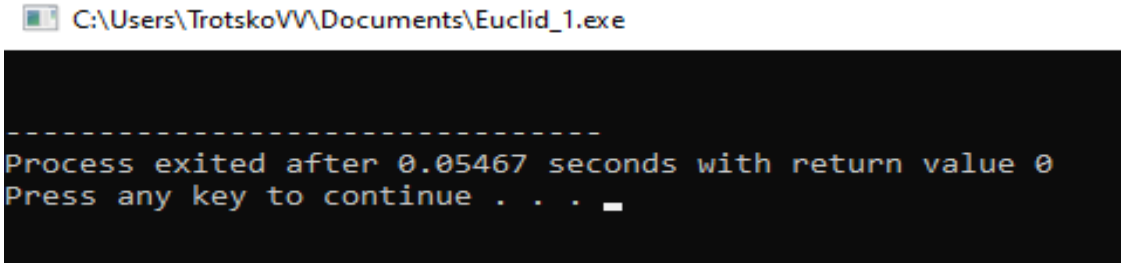


Рис. Б.11. Зовнішній вигляд вікна після виконання програмного коду з інформацією про час виконання програми.

3. Після виконання програми записати час в четверту колонку таблиці Б.9 під назвою «Алгоритм Евкліда-віднімання». Повторити виконання програми, змінюючи кількість повторів циклу і розрядність випадкових чисел у відповідності із значеннями записаними в рядках табл. Б.9. Для кожної зміни коду виконувати компіляцію кнопкою F9 і здійснювати виконання коду кнопкою F10. Значення часу виконання програми записувати в четверту колонку таблиці Б.9.

4. Повторити пункти 2 і 3 для коду програми, що використовує операцію ділення по модулю. Значення часу виконання цього коду записувати в п’яту колонку таблиці Б.9.

Код програми для алгоритму Евкліда з операцією ділення по модулю

```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;
int main()
{
  setlocale(LC_ALL, "ukr");
  int A, B;
  srand(time(NULL));
  for (int i = 0; i < 10; i++)
  {
        // Генерація 2-х випадкових чисел
        A = 1+rand() % 100;          // Перше число
        B = 1+rand() % 100;          // Друге число
```

```
        //      Алгоритм Евкліда
        while ((A != 0) && (B!=0))
             {
               if (A>B) { A = A%B;}  else { B = B%A; }//Використання
ділення
        }
       A=A+B;
 }
 cout << endl;
 return 0;
}
```

5. Після заповнення четвертної і п’ятої колонок таблиці Б.9 побудувати графік залежності часу виконання програми від кількості повторів циклу та графік залежності часу виконання програми від розрядності випадкових чисел.

6. Зробити висновки в яких пояснити причину існування різниці в швидкості виконання алгоритму, що реалізований двома різними методами.

Лабораторна робота №9
ВИВЧЕННЯ ОСОБЛИВОСТЕЙ ЕВРИСТИЧНИХ АЛГОРИТМІВ НА ПРИКЛАДІ ЗАДАЧІ КОМІВОЯЖЕРА

Мета роботи

Ознайомитися з особливостями роботи евристичних алгоритмів та вивчити окремі практичні аспекти їх роботи.

Необхідне обладнання

Комп’ютер з операційною системою Windows версії не нижче 7.

Встановлений редактор мови програмування DEV C++.

В лабораторній роботі використаний програмний код в якому об’єднані три методи. Схематично використання цих методів показане на рис. Б.11.

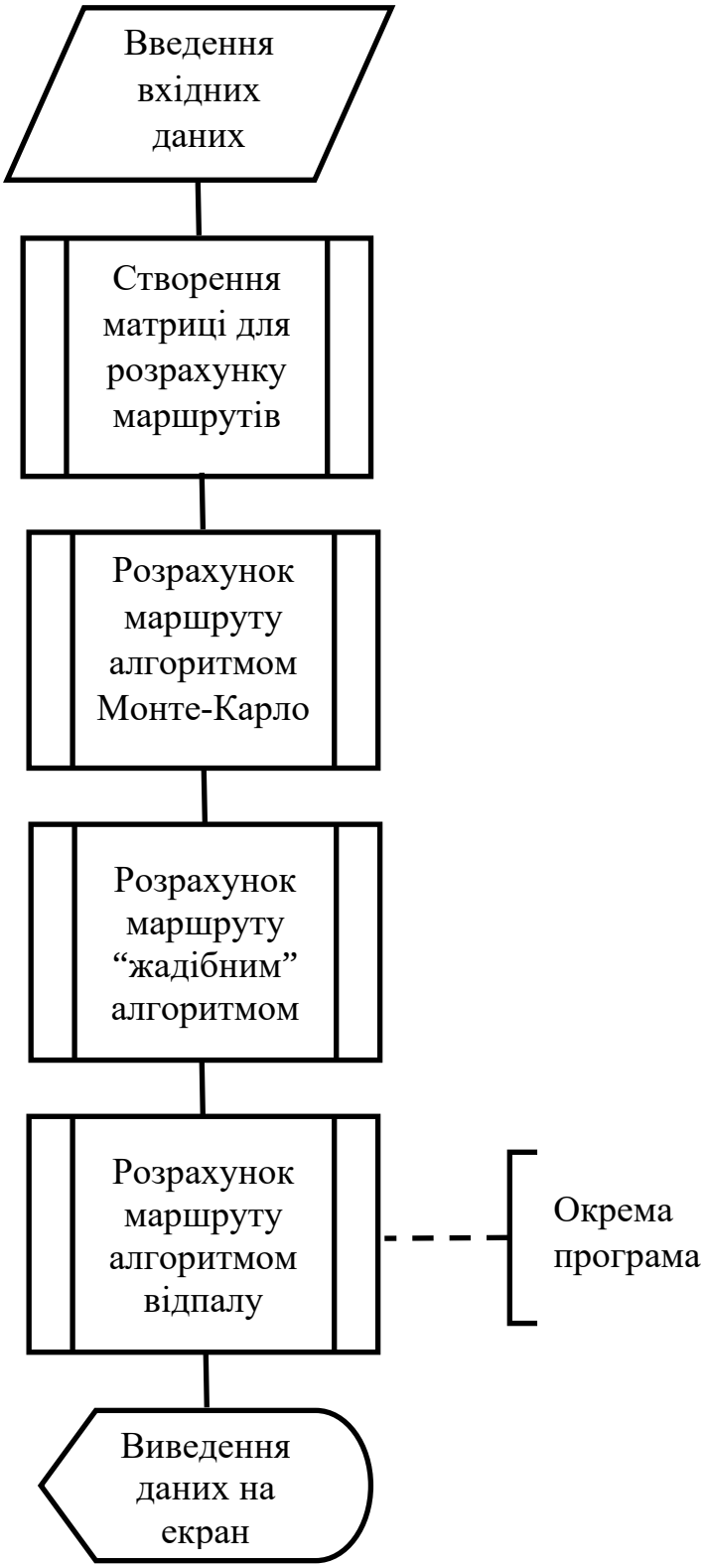


Рис. Б.11 – Послідовність застосування евристичних методів в лабораторній роботі

Порядок виконання роботи

1. Закрити всі сторонні програми на комп'ютері (браузери, зайві офісні програми які не використовуються тощо).
2. Відкрити редактор DEV C++.
3. Створити вихідний файл – Ctrl+N.
4. Скопіювати і вставити код програми MonteCarlo+Greedy.cpp у вікно редактора.
5. Встановити кількість міст у відповідності з першим рядком табл. Б.10. Для цього змінити рядки коду.

```
int n = 10; //Кількість міст
int res = 9999999;//Шукана відстань

const unsigned int DIM1 = 10;
const unsigned int DIM2 = 10;
const unsigned int DIM3 = 10+1;
```

на наступні

```
int n = 5; //Кількість міст
int res = 9999999;//Шукана відстань

const unsigned int DIM1 = 5;
const unsigned int DIM2 = 5;
const unsigned int DIM3 = 5+1;
```

6. Відкомпілювати – F9 та запустити – F10 програму. Записати значення довжини маршруту для "жадібного" алгоритму та алгоритму Монте-Карло у відповідні клітинки табл. Б.10. Змінити кількість ітерацій в коді в наступному рядку – for (int mar=0; mar<10000; mar++) з 10000 на 50000.
7. Повторно відкомпілювати та запустити програму. Отримане значення довжини записати у відповідну клітинку табл. Б.10.
8. Повторити пункти 4-6 до повного заповнення відповідних клітинок табл. Б.10.
9. Повторити пункти 4-6 та заповнити таблицю Б.10 для алгоритму імітації відпалу.

10. Побудувати гістограму для значень табл. Б.10 та зробити висновки.

Таблиця Б.10

Результати обчислень роботи алгоритмів

№	Розмірність матриці маршрутів, n	Довжина маршруту “жадібний” алгоритм	Довжина маршруту алгоритм Монте-Карло,	Довжина маршруту алгоритм імітації відпалу
1	5			
2	10			
3	20			
4	30			
5	40			
6	50			
7	100			

Код програми для алгоритмів Монте-Карло та жадібного

```
#include <iostream>//-1
#include <ctime>
#include <iomanip>
#include <stdlib.h>
using namespace std;
//Код для створення зворотносиметричної матриці відстаней до міст
                    int n = 10; //Кількість міст

const unsigned int DIM1 = 10;
const unsigned int DIM2 = 10;
const unsigned int DIM3 = 10+1;

int ary[DIM1][DIM2];//масив для матриці відстаней. перший елемент -
номер рядка, другий - номер стовпчика
int ary1[DIM1][DIM2];//копія матриці відстаней
int aary[DIM3];//масив для маршруту
int aary1[DIM3];//масив для комбінованого маршруту
int aary2[DIM3];//масив для комбінованого маршруту

int main() {
  setlocale(0, "");

  int m,nm,oz,oz1;
```

```
   srand(time(NULL));//Забезпечує неповторюваність випадковості
   for (int i = 0; i < DIM1; i++) {
      for (int j = 0; j < DIM2; j++) {
         if (i!=j)
                  {
                        /*if (i>=5)    {*/ary[i][j] = 1 + rand() %
100;//}//генерація випадкового числа
                  }
                  else { ary[i][j] = 0;//нулі на діагоналі
                  }
      }
   }
   for (int i = 0; i < DIM1; i++) {
      for (int j = 0; j < DIM2; j++) {
                   ary[i][j] = ary[j][i];
      }
   }

//Копіювання матриці ary до матриці ary1 та ary2
   for (int i = 0; i < DIM1; i++) {
      for (int j = 0; j < DIM2; j++) {
                   ary1[i][j] = ary[i][j];
      }
   }

//Зворотносиметрична матриця відстаней створена
//Реалізація алгоритму Монте-Карло

  cout << endl;//Новий рядок
  cout << 0 << "алгоритм Монте-Карло";
  cout << endl;//Новий рядок

  int *work_array = new int [n];
  int *work_array1 = new int [n];
  int res = 9999999;//Шукана відстань
  for (int mar=0; mar<10000; mar++)//Цикл за кількістю випадкових
маршрутів
  {

   for (int i = -1; i <= n; i++)     { work_array[i] = 0; }//Заповнення масиву
із маршрутами нулями

  oz=0; int j=0;
  while(oz==0)
```

```
 {
        m= 0+rand() % n;
          if    (m!=0)
          {
                int ozz=0;
       for (int p=1; p<n; p++)
       {
        if      (work_array[p]==m) {ozz=1;}
       }
       if (ozz==0){j++;work_array[j]=m;}
       if (j==n-1) {oz=1;}
    }
 }

 int dov = 0;int k,l;

 for (int i = 0; i < n; i++)
 {
        k=work_array[i];
 l=work_array[i+1];
 dov = dov+ary[k][l];
 }
  if (res>dov)
 {
    res=dov;
     for (int i = 0; i < n; i++)
     {
            work_array1[i] = work_array[i];
     }

 }
}
cout << endl;//Новий рядок
cout << res <<"";//Результат алгоритму Монте-Карло
//Реалізація жадібного алгоритму
 cout << endl;//Новий рядок
 cout << endl;//Новий рядок
 cout << 0 << "жадібний алгоритм";
 cout << endl;//Новий рядок

for (int i = 0; i < n; i++) {aary[i] = 0;}//заповнення масиву для маршруту

      cout << endl;//Новий рядок
int nn=0;
```

```
for (int j = 0; j < n-1; j++)
{

int nom=nn;int p=nn;
int min=99999;

for (int i = 0; i < n; i++)
{
  if ((min>=ary[nn][i])&&(ary[nn][i]>0))    { min=ary[nn][i];nom=i;}
}
nn=nom;

for (int i = 0; i < n; i++) {ary[i][p] =0;}

aary[j+1]=nn;

}

cout << endl;//Новий рядок
//Розрахунок довжини маршруту
int dov = 0;int k,l;
       for (int i = 0; i < n; i++) {
           k=aary[i];
//         cout << setw(4) << aary[i];//Результат жадібного алгоритму
           l=aary[i+1];
                  dov = dov+ary1[k][l];
       }
cout << endl;//Новий рядок
cout << setw(4) << dov;//Результат жадібного алгоритму
//cout << setw(4) << "greedy";

//delete [] ary;

//Комбінований алгоритм
  cout << endl;//Новий рядок
          //Виведення матриці на екран
 cout << endl;//Новий рядок

  return 0;
}
```

Код програми для алгоритму імітації відпалу

```
#include <cmath>
#include <iostream>//-1
#include <ctime>
#include <iomanip>
#include <stdlib.h>
using namespace std;

int n = 10;
const unsigned int DIM1 = 10;
const unsigned int DIM2 = 10;
const unsigned int DIM3 = 10+1;

int ary[DIM1][DIM2];
int ary1[DIM1][DIM2];
int work_array[DIM1];
int work_array1[DIM1];
int work_array2[DIM1];

int main() {
  setlocale(0, "");

  int nm,oz,oz1;
  cout << endl;
  cout << 0 << "Алгоритм відпалу";
  cout << endl;
  cout << endl;

    srand(time(NULL));
    for (int i = 0; i < DIM1; i++) {
       for (int j = 0; j < DIM2; j++) {
          if (i!=j)
                  {
                          ary[i][j] = 1   + rand() % 100;
                  }
                  else { ary[i][j] = 0;
                  }
       }
    }
    for (int i = 0; i < DIM1; i++) {
       for (int j = 0; j < DIM2; j++) {
                   ary[i][j] = ary[j][i];
       }
```

```
  }

  for (int i = 0; i < DIM1; i++) {
     for (int j = 0; j < DIM2; j++) {
                ary1[i][j] = ary[i][j];
     }
  }


 for (int i = 0; i <= n; i++)        { work_array[i] = 0; }

 oz=0; int j=0, m;
 while(oz==0)
 {
   m= 0+rand() % n;
          if    (m!=0)
          {
                int ozz=0;
       for (int p=1; p<n; p++)
       {
         if       (work_array[p]==m) {ozz=1;}
       }
       if (ozz==0){j++;work_array[j]=m;}
       if (j==n-1) {oz=1;}
     }
 }

for (int ii = 0; ii < 500000; ii++)
{

 for (int i = 0; i <= n; i++)        { work_array1[i] = work_array[i]; }

 oz=0; int m1=0;
 while(oz==0)
 {
        m = 0+rand() % n;
        m1 = 0+rand() % n;
        if ((m!=m1)&&(m!=0)&&(m1!=0)) {oz=1;}
  }
 for (int i = -1; i <= n; i++)
 {
  if(m==work_array1[i]) {work_array1[i] = m1; } else
  if(m1==work_array1[i]) {work_array1[i] = m; }
 }
```

```
int dov = 0; int dov1 = 0; int k,l,k1,l1;
for (int i = 0; i < n; i++)
{
        k=work_array[i];
        k1=work_array1[i];
l=work_array[i+1];
l1 =work_array1[i+1];
dov = dov+ary[k][l];
dov1 = dov1+ary[k1][l1];
}

int ooz=0;
if (dov<=dov1)
{
 for (int i = 0; i <= n; i++)      { work_array1[i] = work_array[i]; }
 }
 else
 {
 int ddov=dov1;
  int ozzz=0; float vv=100;
  while(ozzz==0)//
  {
    float eexp, dif, jj;
    dif = dov-dov1;
    if (dif>0)
    {
    vv=vv/2;
    eexp = float(100)*exp((float)-dif/float((vv)));
    jj = 1+rand() % 100;
  if (eexp>jj)
  {
     oz=0;
     while(oz==0)
     {
            m = 0+rand() % n;
            m1 = 0+rand() % n;
            if ((m!=m1)&&(m!=0)&&(m1!=0)) {oz=1;}
     }
     for (int i = 0; i <= n; i++)
     {
       if(m==work_array[i]) {work_array[i] = m1; } else
       if(m1==work_array[i]) {work_array[i] = m; }
     }
     dov1 = 0;
```

```
      for (int i = 0; i < n; i++)
      {
            k=work_array[i];
       l=work_array[i+1];
       dov1 = dov1+ary[k][l];
      }
         if (dov1<ddov)
         {
               ooz=1;
               for (int i = 0; i < n; i++)  {work_array2[i]=work_array[i];}
         }
   }
   else {ozzz=1;}
     } else {ozzz=1;}
   }
  }
  if (ooz==1) {for (int i = 0; i < n; i++)  {work_array[i]=work_array2[i];}}
  else {for (int i = 0; i < n; i++) {work_array[i]=work_array1[i];}}
}

 int res = 0; int k,l;
 for (int i = 0; i < n; i++)
 {
       k=work_array[i];
 l=work_array[i+1];
 res = res+ary[k][l];
 }
  cout << endl;// Кінець рядка
  cout<<"Результат: "<<res<<endl;

return 0;

}
```

ТЕОРІЯ АЛГОРИТМІВ

Навчально - методичний посібник

м. Київ, Україна, 27 жовтня 2022 р.

Науковий редактор:
Технічний редактор:

Підписано до друку 20.12.2021 р. Формат 60х84/16. Папір офсетний.
Друк офсетний. Гарнітура Times New Roman.
Ум. друк. арк. 19,8. Наклад 300 прим.
Зам. 232

Університет економіки та права «КРОК»
Свідоцтво про внесення суб'єкта видавничої справи
до Державного реєстру ДК № 613 від 25.09.2001 р.

Надруковано департаментом поліграфії
Університет економіки та права «КРОК»
місто Київ, вулиця Табірна, 30-32
тел.: (044) 455-69-80
e-mail: polygrafi a.krok@gmail.com