МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ОДЕСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ імені І. І. МЕЧНИКОВА
ФАКУЛЬТЕТ МАТЕМАТИКИ, ФІЗИКИ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

ТЕОРІЯ АЛГОРИТМІВ

методичні вказівки до лабораторних робіт студентам
факультету математики, фізики та інформаційних технологій
першого (бакалаврського) рівня освіти,
спеціальності 122 «Комп'ютерні науки»

ОДЕСА
ОНУ
2023

УДК 004.421(076.5)
Т338

Укладачі:
Г. В. Коренкова, кандидат фізико-математичних наук, доцент кафедри комп'ютерних систем та технологій;
Л. Я. Мартинович, старший викладач кафедри комп'ютерних систем та технологій**;**
О. М. Зуй, викладач кафедри комп'ютерних систем та технологій**.**

Рецензенти:
А. Л. Рачинська, кандидат фізико-математичних наук, доцент кафедри механіки, автоматизації та інформаційних технологій Одеського національного університету імені І. І. Мечникова;
В. Ф. Ложечніков, кандидат технічних наук, доцент кафедри комп'ютеризованих систем та програмних технологій Національний університет «Одеська Політехніка».

Рекомендовано Вченою радою факультету МФІТ
Одеського національного університету імені І. І. Мечникова
Протокол № 3 від 23 грудня 2022 р.

Т338 **Теорія** алгоритмів : метод. вказівки до лаб. робіт студентів факультету математики, фізики та інформаційних технологій першого (бакалавр.) рівня освіти, спеціальності 122 «Комп'ютерні науки» / уклад.: Г. В. Коренкова, Л. Я. Мартинович, О. М. Зуй. – Одеса : Одес. нац. ун-т ім. І. І. Мечникова, 2023. – 71 с.

Пропоновані методичні вказівки стануть у нагоді при виконанні лабораторних робіт з дисципліни базової підготовки за спеціальністю «Теорії алгоритмів», яка викладається студентам першого (бакалаврського) рівня освіти, спеціальності 122 «Комп'ютерні науки» факультету математики, фізики та інформаційних технологій Одеського національного університету імені І. І. Мечникова.

УДК 004.421(076.5)

ЗМІСТ

ВСТУП

Дані методичні вказівки до лабораторних занять підготовлені відповідно до програми курсу «Теорія алгоритмів». Навчальна дисципліна відноситься до базової підготовки студентів 2 курсу спеціальності «122- Комп’ютерні науки».

Предметом вивчення навчальної дисципліни «Теорія алгоритмів» є опис різноманітних підходів до розробки алгоритмів.

Метою викладання цієї дисципліни є отримання студентами знань з області побудови та аналізу алгоритмів щодо вирішення різноманітних практичних задач.

Завдання дисципліни “Теорія алгоритмів” - надати студентам знання в сфері реалізації задач обробки інформації за допомогою комп'ютерної техніки. Такі знання майбутній спеціаліст зможе застосовувати як при подальшому навчанні, так і після отримання вищої освіти у своїй професійній діяльності.

Лабораторні роботи з дисципліни виконуються з метою закріплення та поглиблення теоретичних та практичних знань та вмінь, набутих у процесі засвоєння всього навчального матеріалу дисципліни. Кожна лабораторна робота містить теоретичні відомості, приклади, контрольні запитання та завдання для самостійної роботи студентів.

Метою даних методичних вказівок є закріплення лекційного матеріалу та вироблення у студентів навичок застосування алгоритмів розв'язання стандартних завдань. До таких відносяться задачі сортування, пошуку, а також аналізу складності побудови алгоритмів. Крім того, розглядаються завдання перетворення дискретної інформації машинами Тюрінга та Поста.

Порядок виконання робіт та оформлення звіту

Лабораторні роботи містять теоретичні відомості, розібрані приклади, контрольні запитання та завдання для самостійної роботи. Вони призначені для закріплення та поглиблення теоретичних та практичних знань та вмінь, набутих у процесі засвоєння всього навчального матеріалу дисципліни. Кожна лабораторна робота виконується як мінімум одну пару (2 академічні години).

Перед виконанням самостійного завдання студенти мають ознайомитись з теоретичним матеріалом та відповісти на контрольні запитання.

При оформленні звіту з лабораторної роботи в нього обов’язково треба включати номер і назву роботи, її мету, всі завдання, передбачені у ході цієї роботи. Результати вирішення завдань необхідно наводити в повній мірі, ілюструючи достатньою кількістю різних варіантів вхідних даних та результатів роботи програм. Там, де це передбачено завданням, потрібно зробити порівняння. В кінці звіту з кожної лабораторної роботи зробити висновки.

Лабораторна робота №1
Машина Тюрінга

Мета роботи: ознайомитись зі способами запису алгоритмів за допомогою машини Тюрінга.

Теоретичні відомості

Машина Тюрінга - це один із способів запису алгоритму. Машина Тюрінга — це алгоритм, записом якого є функціональна таблиця або функціональна діаграма, а правило виконання — опис його структури.

Машина Тюрінга складається з нескінченної в обидва боки стрічки та каретки, яка переміщується відносно стрічки (Рис. 1.1). Стрічка розділена на комірки, заповнені символами із зовнішнього алфавіту. Кожна комірка може містити тільки одну літеру. У кожний момент часу машина перебуває в одному із станів, що позначені літерами внутрішнього алфавіту, й оглядає тільки одну комірку стрічки.

За кожен крок каретка може виконати такі дії:

1. Зчитати символ з комірки.
2. Записати символ у комірку.
3. Зміститися або вліво (Л), або вправо (П), або залишитись оглядати дану комірку.

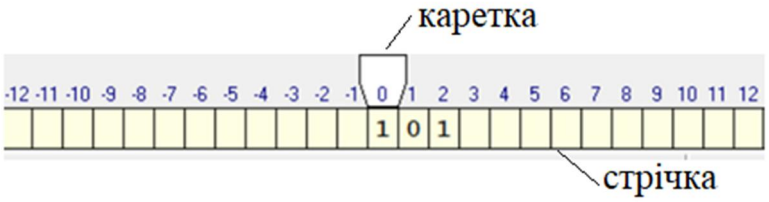


Рис.1.1 Машина Тюрінга.

Щоб задати конкретну машину Тюрінга, потрібно описати для неї такі складові:

1. Зовнішній алфавіт ($A = \{a_0, a_1, \dots, a_n\}$), який складається з кінцевої множини. Елементами цієї множини є літери (символи). Одна з літер цього алфавіту повинна бути порожнім символом (a_0).
2. Внутрішній алфавіт ($Q = \{q_0, q_1, \dots, q_n\}$). Кінцева множина станів каретки. Один із станів має бути початковим (q_1). Ще один із станів має бути кінцевим– стан зупинки (q_0).
3. Таблиця переходів. Опис поведінки каретки в залежності від стану та символу, який було зчитано.

Машиною Тюрінга називається часткове відображення

$$\{q_0, q_1, \dots, q_{n-1}\} \times \{a_0, a_1, \dots, a_n\} \to \{q_0, q_1, \dots, q_{n-1}\} \times \{\Lambda, \Pi\} \times \{a_0, a_1, \dots, a_n\},$$

де Λ, П – відповідно «вліво», «вправо»;

$\{q_0, q_1, \dots, q_{n-1}\}$ - безліч станів машини.

Кожна команда для машини Тюрінга є комбінацією трьох складових: вказівок, який символ записати в комірку, куди пересунути каретку і в який стан перейти. Команда $q_i a_j \to q_k a_i X$ (де X – напрям зсуву каретки) означає, що машина перебуває в стані q_i й оглядає комірку, у якій записана буква a_j та переходить у стан q_k. У комірці, що оглядається, видаляється літера a_j та заноситься туди літера a_i.

Програма машини Тюрінга записується у вигляді таблиці переходів. В першому стовпчику цієї таблиці знаходяться усі можливі внутрішні стани, а перший рядок – усі можливі символи заданого зовнішнього алфавіту. Вміст таблиці – це команди для машини Тюрінга. Символ в комірці, який зчитує каретка і її внутрішній стан визначають яку команду потрібно виконати. Команда, яку потрібно виконати на наступному кроці визначається перетином символів внутрішнього і зовнішнього алфавітів в таблиці.

Можливі зупинки машини Тюрінга:

1) результативна зупинка - у ході виконання програми машина дійде до виконання команди зупинки;

2) машина ніколи не зупиняється, відбувається зациклювання.

3) виникнення помилки: даного символу немає в алфавіті, не вказано команду, стан q_i відсутній.

Приклад 1.1: Сконструювати машину Тюрінга з зовнішнім алфавітом A={a,b,c}, яка замінює символ в комірці на a після першого входження c.

Таблиця переходів наведено на рисунку 1.2.

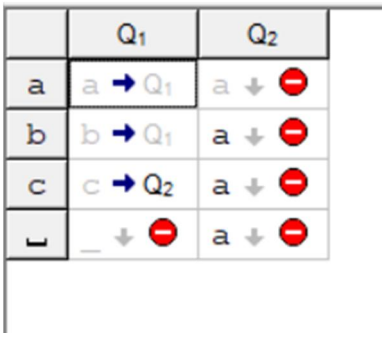


Рис. 1.2 Приклад програми машини Тюрінга.

Пояснення розв’язання задачі:

Якщо перший символ є a чи b, то каретка залишається в стані q_1 і рухається праворуч. Якщо ж зустрічається c, то машина переходить в стан q_2 та замінює будь-які символи, що знаходяться в наступній комірці, на a. При виявленні порожньої комірки, машина зупиняється.

Приклад 1.2: Нехай зовнішнім алфавітом машини Тюрінга є набір символів {0,1,2,3,4,5,6,7,8,9,+}. Знайти суму натуральних чисел в десятковій системі числення (Рис. 1.3). Каретка знаходиться над останнім символом.

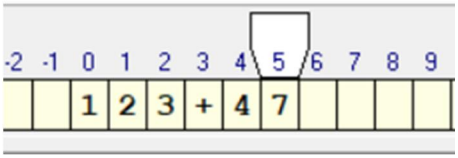


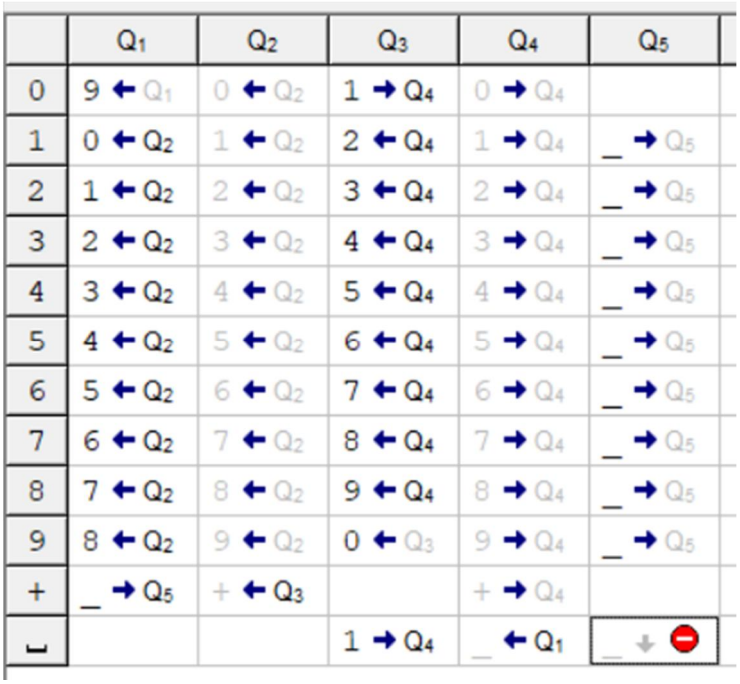
Рис.1.3. Стрічка машини Тюрінга

Таблиця переходів наведена на рисунку 1.4.

Пояснення розв’язання задачі:

В першому стані q_1 машина віднімає одиницю з другого числа та каретка рухається ліворуч, переходячи в стан q_2. В другому стані каретка буде рухатись ліворуч не змінюючи символи, які находяться в комірках, доки в стрічці не зустріне символ «+». Каретка зміщується на останню цифру першого числа і

машина переходить у стан q_3. В цьому стані машина додає одиницю до першого числа та переходить у стан q_4. В четвертому стані каретка рухається праворуч поки не зустріне порожню комірку. Машина переходить в стан q_1 і всі кроки повторюються. В стані q_5 машина стирає зайві символи.



	Q1	Q2	Q3	Q4	Q5
0	9 ← Q1	0 ← Q2	1 → Q4	0 → Q4	
1	0 ← Q2	1 ← Q2	2 ← Q4	1 → Q4	_ → Q5
2	1 ← Q2	2 ← Q2	3 ← Q4	2 → Q4	_ → Q5
3	2 ← Q2	3 ← Q2	4 ← Q4	3 → Q4	_ → Q5
4	3 ← Q2	4 ← Q2	5 ← Q4	4 → Q4	_ → Q5
5	4 ← Q2	5 ← Q2	6 ← Q4	5 → Q4	_ → Q5
6	5 ← Q2	6 ← Q2	7 ← Q4	6 → Q4	_ → Q5
7	6 ← Q2	7 ← Q2	8 ← Q4	7 → Q4	_ → Q5
8	7 ← Q2	8 ← Q2	9 ← Q4	8 → Q4	_ → Q5
9	8 ← Q2	9 ← Q2	0 ← Q3	9 → Q4	_ → Q5
+	_ → Q5	+ ← Q3		+ → Q4	
␣			1 → Q4	_ ← Q1	_

Рис. 1.4. Програма машини Тюрінга знаходження суми двох чисел в десятковій системі числення.

При розв’язанні задачі P алгоритмом A розглядають такі характеристики:

1) кількість кроків $T_a(x)$, яких необхідно зробити алгоритму A для отримання результату під час використання вхідних даних x. Величина $T(A,n) = \max_{|x|=n} T_a(x)$ (максимум береться за всіма вхідними даними обсягу n) називається часовою складністю алгоритм A.

2) обсяг пам'яті $M_a(x)$, необхідний для зберігання всіх вхідних та проміжних даних у процесі виконання алгоритму під час використання вхідних даних x. Величина $M(A,\ n) = max_{|x|=n} M_a(x)$ (максимум береться за всіма вхідними даними обсягу n) називається ємнісною складністю алгоритму A.

Для визначення часової складності алгоритму замість загального числа кроків алгоритму можна використовувати кількість операцій певного виду.

Аналогічно можна визначити середні величини часової та ємнісної складності алгоритму. Складність задачі P – це складність найкращого алгоритму, відомого для її вирішення, тобто (мінімум $S(P,n) = \min_A T(A,n)$ береться за всіма алгоритмами для завдання P).

Контрольні питання

1. Дайте визначення машини Тюрінга.
2. З чого складається машина Тюрінга?
3. Що таке зовнішній та внутрішній алфавіти?
4. Якими командами визначається програма машини Тюрінга?

Порядок виконання лабораторної роботи

1. Скласти програму машини Тюрінга для розв’язання задач.
2. Перевірити функціонування машини, написавши алгоритм обробки різних вхідних послідовностей.
3. Визначити необхідну пам'ять (число комірок на стрічці) та часову складність (число тактів роботи) для машини Тюрінга.

Задачі

1. A = {a, b, c}. Приписати ліворуч до непорожнього слова P його перший символ.
2. A = {a, b, c}. Замінити на a кожен другий символ у слові P
3. A = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}. Додати до числа P 2.
4. A = {a, b, c}. Приписати праворуч до слова P символи abc (P → Pabc).
5. A = {a, b, c}. Замінити всі входження символу a у слові P на символ c.
6. A = {a, b, c, 1, 0}. Визначити, чи входить символ a у слово P. Якщо входить, то вивести 1, а якщо ні, то нуль.

7. A={a,b,c}. Для не пустого слова P визначити, чи є його першим символом a. Якщо є, то залишити тільки цей символ замість слова, а якщо ні, то слово замінити порожнім.

8. A = {a, b, =}. Подвоїти кожен символ слова P (наприклад: bab → bbaabb).

9. A={0, 1, 2, 3, 4, 5, 6 ,7, 8, 9, +}. Знайти суму двох чисел, записаних в десятковій системі числення.

10. A={0, 1, 2, 3, 4, 5, 6 ,7, 8, 9, -}. Знайти різницю двох чисел, записаних в десятковій системі числення. Перше число більше за друге.

11. A={0, 1, 2, 3, 4, 5, 6 ,7, 8, 9, -}. Знайти модуль різниці двох чисел, записаних в десятковій системі числення.

12. A={0, 1, 2, 3, 4, 5, 6 ,7, 8, 9}. Помножити число на 2.

13. A={0, 1, 2, 3, 4, 5, 6 ,7, 8, 9}. Визначити є число парним або непарним. Якщо число є парним, то замість нього вивести одиницю, а якщо непарне, то нуль.

14. A={1, 0}.У слові P видалити всі нулі поміж усіх одиниць. (Наприклад, 1001110 → 1111.)

15. A = {a, b}. В непорожньому слові P поміняти місцями перший та останній символи.

16. A={1}. Число записане в унарній системі числення помножити на 2.

17. A={0, 1, 2, 3, 4, 5, 6 ,7, 8, 9}. Число записати в оберненому порядку.

18. A={0, 1, 2, 3, 4, 5, 6 ,7, 8, 9}. Перевести число в унарній системі в десяткову систему числення.

19. A = {a, b, c, 1}. В стрічці записані слова літерами a, b, c, які розділені порожнюю коміркою. Підрахувати кількість слів у стрічці.

20. A = {a, b, c}. Якщо P – слово парної довжини (0, 2, 4, …), видати відповідь a, інакше – порожнє слово.

Лабораторна робота №2
Машина Поста

Мета роботи: вивчення формального визначення поняття алгоритму.

Теоретичні відомості

Абстрактна машина Поста є нескінченною стрічкою, розділеною на однакові комірки, кожна з яких може бути або порожньою, або заповненою міткою, і каретки, яка може переміщатися вздовж стрічки на одну клітинку вправо або вліво (рис. 2.1). Коли вона нерухома, то знаходиться навпроти однієї комірки. Каретка може наносити в комірку стрічки мітку, якщо цієї мітки там раніше не було, видаляти мітку, якщо вона там була, або перевіряти наявність мітки у комірці.

Інформація про заповнені мітками комірки стрічки характеризує стан стрічки, який може змінюватися в процесі роботи машини. У кожний момент часу каретка («-») знаходиться над однією з комірок стрічки і оглядає її. Інформація про розташування каретки разом зі станом стрічки характеризує стан машини.

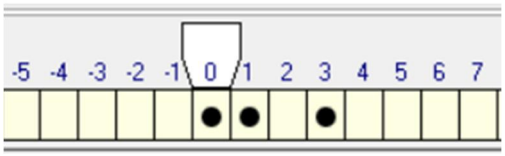


Рис. 2.1. Машина Поста

Машина Поста може виконувати такі елементарні операції:

1) Помітити комірку, якщо вона порожня.

2) Стерти мітку в комірці, якщо вона є.

3) Перемістится вліво на 1 комірку.

4) Перемістится вправо на 1 комірку.

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

6) Зупинитися.

Програма являє собою нумеровану послідовність інструкцій, причому переходи в інструкції проводяться на вказані в ній номери інших інструкцій. У команди «стоп» посилань нема.

Для роботи машини необхідно задати програму та її початковий стан (тобто стан стрічки і позицію каретки). Кареткою управляє програма, що складається з рядків команд.

Програмою для машини Поста називається непорожній список послідовно пронумерованих команд наступної структури: n K m, де n - порядковий номер команди, K − дія, яка виконується кареткою, m - номер наступної команди, яку необхідно виконати.

Графічне представлення команд машини Поста наведено в таблиці 1.

Таблиця 1.

Графічне представлення команд машини Поста

Команда	Опис
>	Крок вправо
<	Крок вліво
0	Стерти мітку
1	Внести мітку
? a: b	Переглянути комірку: якщо в комірці знаходиться 0, тоді перейти на команду з номером a, інакше на команду з номером b.
.	Зупинка

Зупинка програми машини Поста може мати три варіанти:

1) зупинка за командою «стоп». Така зупинка називається результативною і вказує на коректність алгоритму (програми);

2) зупинка під час виконання неприпустимої команди. У цьому випадку зупинка називається безрезультативною;

3) машина не зупиняється ніколи. У цьому та попередньому випадку ми маємо справу з некоректним алгоритмом (програмою).

Приклад 2.1. На стрічці задано масив міток. Збільшити довжину масиву на 2 мітки. Каретка знаходиться або ліворуч від масиву, або над одним із осередків самого масиву.

Програма представлена на рис. 2.2.

	Команда	Переход
1	?	2,3
2	>	1
3	>	4
4	?	5,3
5	1	6
6	>	7
7	1	8
8	.	

Рис. 2.2. Програма машини Поста, яка збільшує довжину масиву на 2 мітки.

Пояснення до розв’язання задачі:

Команди 1 і 2 - пересуваємо каретку до масиву. Команди 3 і 4 - пересуваємо каретку до кінця масиву. Команди 5-7 - ставимо 2 мітки в кінці масиву.

Вважається, що машина Поста працює в унарній системі числення. Число k представляється на стрічці машини k+1 мітками, які йдуть підряд (одна мітка означає цифру «0»). Між двома числами робиться інтервал як мінімум з однієї порожньої комірки на стрічці. Наприклад, числа 3 та 2 на стрічці машини Поста виглядає наступним чином рис. 2.3.

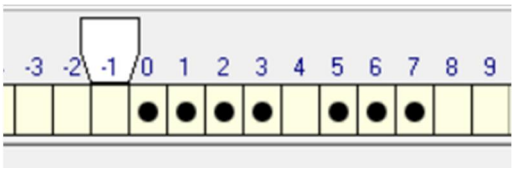


Рис. 2.3. Запис чисел 3 і 2 на стрічці машини Поста

Приклад 2.2. Додати два числа в унарній системі числення. Каретка знаходиться над крайньою лівою міткою першого масиву.

Програма знаходження суми двох чисел в унарній системі числення представлена на рис. 2.4.

	Команда	Переход
1	0	2
2	>	3
3	?	4,2
4	1	5
5	>	6
6	?	7,5
7	<	8
8	0	9
9	.	

Рис. 2.4. Програма знаходження суми двох чисел в унарній системі числення.

Пояснення до розв’язання задачі:

Команда 1 стирає першу мітку в масиві. Друга та третя команди переміщують каретку до порожньої комірки. В четвертій команді в порожній комірці ставиться мітка. Команди 5, 6 переміщують каретку в кінець другого масиву міток. Команди 7, 8 стирають зайву мітку.

Контрольні питання

1. Що таке машина Поста?
2. Які складові абстрактної машини Поста?
3. Як примітивні операції в машині Поста?
4. Що таке початковий стан каретки?
5. Що таке стан стрічки?
6. Що таке програма для машини Поста?
7. Яка зупинка називається безрезультативною?

Порядок виконання лабораторної роботи

1. Ознайомитись з принципами функціонування машини Поста.
2. Скласти програму машини Поста для розв’язання задач.
3. Перевірити функціонування машини, написавши алгоритм обробки різних вхідних послідовностей.

Задачі

1. Додати до масиву міток одиницю, якщо каретка знаходиться над крайньою правою міткою.

2. Додати до масиву міток одиницю, якщо каретка знаходиться над довільною міткою масиву.

3. Додати до масиву міток одиницю, якщо каретка знаходиться над довільною коміркою праворуч від масиву.

4. Додати до масиву міток одиницю, якщо каретка знаходиться над довільною коміркою ліворуч від масиву.

5. Зменшити ціле число на 2. Каретка знаходиться над крайньою лівою міткою.

6. Нехай задано вихідний стан каретки і потрібно на порожній стрічці написати дві мітки: одну в секцію під кареткою, другу праворуч від неї.

7. На стрічці є кілька міток (загальна кількість міток не менше 1). Між мітками множини можуть бути проміжки, довжина яких становить одна комірка. Заповнити всі порожні комірки мітками. Каретка знаходиться над лівою крайньою міткою.

8. Стерти єдину мітку в стрічці. Каретка знаходиться на деякій відстані лівіше за неї. Необхідно підвести каретку до мітки, стерти її і зупинити каретку ліворуч від цієї комірки.

9. На стрічці є масив із міток. Каретка оглядає крайню ліву мітку. Праворуч від даного масиву на відстані в k комірок знаходиться ще одна мітка. Скласти для машини Поста програму, що пересуває даний масив до даної комірки. Потрібно поєднати їх в один масив. Каретка знаходиться над крайньою лівою міткою першого масиву.

10. Обчислити різницю двох масивів. Каретка розташовується над лівою коміркою правого масиву.

11. Скласти кілька натуральних чисел. Числа відокремлені один від одного пробілами. Каретка знаходиться праворуч від першого числа.

12. Скласти програму машини Поста для додавання двох чисел, записаних на довільній відстані одне від одного.

13. Ціле число N записане на стрічці як послідовність міток. Обчислити функцію, яка дорівнює 0 (залишає порожню стрічку), якщо число N парне, і дорівнює 1, якщо N - непарне. Каретка стоїть над першою зліва міткою.

14. На стрічці задано масив. Подвоїти масив. Каретка розташовується над першою коміркою масиву.

15. Задано масив міток. Каретка розташовується над довільною коміркою масиву, але не над крайніми мітками. Стерти всі мітки, крім крайніх.

16. Скласти програму знаходження різниці двох додатних цілих чисел що знаходяться на стрічці машини Поста. Ліве число більше за праве та розділені порожньою коміркою. Каретка знаходиться над першою міткою лівого числа.

17. На стрічці машини Поста розташований масив з непарної кількості міток. Скласти програму, яка знаходить середню мітку масиву та її стирає.

18. . На стрічці машини Поста розташовані N міток, які розділені порожніми комірками. Стиснути масив так, щоб усі N міток займали N підряд комірок.

19. На стрічці знаходиться два масиви, розділені порожньою коміркою. З’ясувати, чи мають вони однаковий розмір. Каретка знаходиться над крайньою лівою міткою першого масиву.

20. На стрічці задано масив. Обчислити залишок від ділення довжини заданого масиву на 3. Каретка розташовується над першою коміркою масиву.

Лабораторна робота №3

Побудова нормальних алгоритмів Маркова

Мета роботи: отримати практичні навички запису алгоритмів з використанням нормальних алгоритмів Маркова.

Теоретичні відомості

Для визначення нормальних алгоритмів Маркова вводиться довільний алфавіт - кінцева множина символів, за допомогою яких описується алгоритм і дані. До алфавіту також включається порожній символ (λ). Під словом потрібно розуміти послідовність непустих знаків алфавіту або порожній знак, який позначає порожнє слово.

Кожен нормальний алгоритм Маркова визначається кінцевою упорядкованою множиною пар слів алфавіту. Ці пари утворюють підстановки. У парі слів підстановки перше слово є не порожнім, а друге слово може бути порожнім символом. В підстановці слова розділяються стрілкою.

Основна операція під час роботи алгоритмів Маркова – це заміна символів в словах у певному алфавіті. Ця операція полягає у здійснені в строго певному порядку замін певних послідовностей символів. Кожен крок роботи алгоритму може бути описаний таким чином:

- У впорядкованій послідовності підстановок шукаємо найпершу підстановку, ліве слово якої входить до рядка даних.
- У рядку даних шукаємо найперше (ліве) входження лівого слова знайденої підстановки.
- Це входження у рядку даних замінюємо на праве слово знайденої підстановки (перетворення даних).

Кроки роботи алгоритму можуть виконуватись до тих пір, поки не виникне ситуація, що жодна підстановка не може виконатись або процес підстановок не може зупинитись.

Алгоритми Маркова виконуються без будь-якого пристрою, що здійснює рух і має внутрішню пам'ять. Наразі є можливість оперувати лише стрічковими

знаками. Сама стрічка в цьому випадку не поділяється на осередки, а має гнучку основу, що дозволяє їй розтягуватися і стискатися, виходячи з того, чи збільшується в слові число символів чи зменшується.

Приклад 3.1. Побудувати нормальний алгоритм, який в будь-якому слові, яке задане в алфавіті A = {a, b}, дописати символ “a” до кінця слова P.

Алгоритм розв’язання задачі наведено на рисунку 3.1.

	Образец		Замена
1	#a	→	a#
2	#b	→	b#
3	#	→	a.
4		→	#

Рис. 3.1. Набір підстановок.

Пояснення до розв’язання задачі:

Для того щоб приписати “a” праворуч, треба спочатку помітити кінець слова. Для цього потрібно ввести додатковий символ, якій не входить до основного алфавіту. Його потрібно перемістити в кінець слова P, а потім замінити на a.

Приклад 3.2. Побудувати нормальний алгоритм, який в будь-якому слові, яке задане в алфавіті A = {a, b}, замінює всі символи “a” на символи “b” та всі символи “b” на “a”.

Початковий стан стрічки зображено на рисунку 3.2.

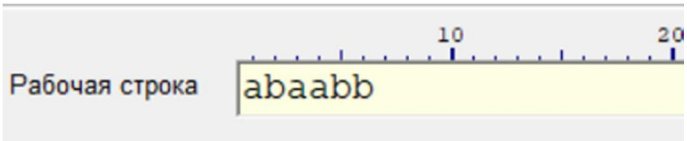


Рис.3.2. Початковий стан стрічки.

Алгоритм розв’язання задачі наведено на рисунку 3.3.

	Образец		Замена
1	#a	→	b#
2	#b	→	a#
3	#	→	.
4		→	#

Рис. 3.3. Набір підстановок.

Пояснення до розв’язання задачі:

На початку роботи алгоритму до вихідного слова зліва дописується додатковий символ #, який не входить до алфавіту цього слова. Таке приписування забезпечується правилом підстановки, яке застосовується до будь-якого слова. Але оскільки воно стоїть останнім, то застосовується лише один раз, коли інші правила ще не застосовні. Далі символ # рухається в кінець слова із заміною літери, що стоїть за ним. Коли # виявляється в кінці слова, застосовується підстановка, яка видаляє символ # та зупиняє виконання алгоритму.

Контрольні питання

1. Як функціонує нормальний алгоритм Маркова?
2. Яка основна операція під час роботи алгоритму Маркова?
3. У яких випадках нормальний алгоритм Маркова закінчує роботу та зупиняється?
4. У якому разі марківська підстановка вважається непридатною до певного слова?

Порядок виконання лабораторної роботи

1. Ознайомитись з принципами побудови нормального алгоритму Маркова.
2. Скласти нормальний алгоритм Маркова
3. Перевірити функціонування алгоритму для різних вхідних послідовностей.

Задачі

1. В слові P в алфавіті A={a, b, c} видалити його перший символ.
2. В алфавіті A={a, b, c} непорожнє слово замінити на порожнє.
3. Вважаючи слово P в алфавіті A={ \| } записом додатнього числа в унарній системі числення, зменшити це число на 1.
4. Вважаючи слово P в алфавіті A={ \| } записом додатнього числа в унарній системі числення, збільшити це число на 2.

5. Реалізувати алгоритм, що виконує перестановку в слові P в алфавіті A = {a, b} таким чином, щоб спочатку стояли всі букви a, а потім b.

6. Визначити чи входить в слово P в алфавіті A={a, b} літера a. Якщо входить, то замінити слово на a, якщо ні, то вивести порожньо слово.

7. В слові P в алфавіті A={a, b} вставити за першим символом літеру a.

8. Дописати справа до слова P в алфавіті A={a, b, c} слово abc.

9. Видалити з непорожнього слова в алфавіті A={a, b} останній символ.

10. Подвоїти кожен символ в слові. Слово задано в алфавіті A={a, b}.

11. В непорожньому слові залишити тільки останній символ. Слово задано в алфавіті A={a, b}.

12. Перенести перший символ непорожнього слова в кінець. Слово задано в алфавіті A={a, b}.

13. Перенести останній символ непорожнього слова в початок. Слово задано в алфавіті A={a, b}.

14. Перевірити чи співпадають у непорожньому слові P перший та останній символи. Якщо співпадають, то видалити обидва ці символи, а інакше слово не міняти. Слово задано в алфавіті A={a, b}.

15. Побудувати нормальний алгоритм додавання двох чисел в двійниковій системі числення.

16. Побудувати нормальний алгоритм віднімання двох чисел в двійниковій системі числення.

17. В слові P в алфавіті A={a, b} остання входження літери a замінити на aa.

18. В алфавіті A слові P замінити перше входження комбінації символів ba, якщо вона є, на символ a. Слово задано в алфавіті A={a, b}.

19. З непорожнього слова непарної довжини видалити середній символ. Слово задано в алфавіті A={a, b}.

20. З непорожнього слова P парної довжини видалити праву половину. Слово задано в алфавіті A={a, b}.

Лабораторна робота №4
Аналіз складності алгоритму

Мета: вивчення методів оцінки алгоритмів і визначення часової і просторової складності типових алгоритмів.

Теоретичні відомості

Складність алгоритму– це кількісна характеристика, що відображує споживані алгоритмом ресурси під час свого виконання.

Можна виділити такі основні складові складності алгоритму:

логічна складність – кількість людино-місяців, витрачених на створення алгоритму;

статистична складність – довжина опису алгоритмів (кількість операторів);

часова складність описує час потрібний для виконання алгоритму. Вона зазвичай визначається шляхом підрахунку кількості елементарних операцій, виконуваних алгоритмом, при цьому вважають, що кожна елементарна операція виконується за фіксовану кількість часу;

просторова складність – кількість умовних одиниць пам’яті (обсяг пам'яті), необхідних для роботи алгоритму.

На часову складність алгоритму значно впливає обсяг вхідних даних. Так, якщо при обробці невеликого обсягу даних різниця між роботою двох різних алгоритмів здається несуттєвою, то збільшення цього обсягу відчутно позначиться на часі виконання кожного з них. Але часова складність часто залежить не тільки від обсягу та значень даних, а також порядку їх надходження. Розглядають поняття часової складності в трьох випадках: гіршому, найкращому і середньому.

Найгірший випадок – найбільша кількість операцій, яка виконується алгоритмом A для вирішення конкретних проблем розмірністю N.

Кращий випадок – найменша кількість операцій, яка виконується алгоритмом A для вирішення конкретних проблем розмірністю N.

Середній випадок – середня кількість операцій, яка виконується алгоритмом A для вирішення конкретних проблем розмірністю N

У загальному випадку складність алгоритму можна оцінити за порядком величини. Цей метод можна застосовувати як до часової, так і до просторової складності.

Часова оцінка складності

Для визначення трудомісткості алгоритму зазвичай використовують наступний набір операцій, складність яких вважають рівною одиниці:

- просте присвоювання: a = b;
- арифметичні операції: -, +, *, /;
- операції порівняння: <,>, ==, <=,> =, <>;
- логічні операції: or, and, not.

За допомогою цих операцій трудомісткість основних алгоритмічних конструкцій можна представити наступним чином.

1. **Лінійна конструкція** з k послідовних блоків.

Трудомісткість конструкції є сума трудоміскостей блоків, які йдуть один за одним:

$$F(n) = f_1 + f_2 + \cdots + f_k,$$

де f_k – трудомісткість k –го блоку.

2. **Конструкція розгалуження**.

if (умова)

{Оператори}

else

{Оператори}

Трудомісткість такої конструкції вимагає аналізу ймовірності виконання операторів, що стоять після «*if*» (p) і «*else*» (1-p) і визначається як:

$$F(n) = f_{if} \cdot p + f_{else} \cdot (1 - p),$$

де f_{if} і f_{else} – трудомісткість відповідних гілок.

3. **Циклічна конструкція**.

for (i=0; i<n; i++)

{тіло циклу}

Трудомісткість такої конструкції визначається так:

$$F(n) = 1 + 3 \cdot n + n \cdot f_{\text{цикл}},$$

де $f_{\text{цикл}}$ – складність тіла циклу, n – число його повторень, $3 \cdot n$ – трудомісткість зміни параметру та перевірки умови виконання циклу.

Складність лінійної ділянки і розгалуження, як правило, не залежить від обсягу даних і вважається рівною деякій константі. При цьому алгоритми, що містять цикли і рекурсії, є більш складними. Величина F зростає зі збільшенням вкладеності циклів, а також при багаторазовому виклику методів обробки даних.

Приклад 4.1: Знайти суму елементів квадратної матриці

```
int Sum_Matrix(int n, int** a)
  {int i, j;
   int s = 0;
   for (i = 0; i < n; i++)
      {for (j = 0; j < n; j++)
          s += a[i][j];}
   return s;}
```

Очевидно, що в цьому алгоритмі розмір вхідних даних треба оцінювати по кількості елементів в масиві, тобто числом n. Алгоритм містить циклічну конструкцію, тому трудомісткість його можна розрахувати таким чином:

$$F = 1 + f_{\text{цикл по } i} = 1 + 1 + 3 \cdot n + n \cdot f_{\text{цикл по } j} =$$

$$= 1 + 1 + 3n + n\big(1 + 3n + n(2 + 2 + 1 + 1)\big) = 2 + 3n + n(1 + 3n + 6n) =$$

$$= 9n^2 + 4n + 2.$$

Асимптотичні оцінки складності

Підрахунок кількості операцій дозволяє порівняти ефективність алгоритмів. Однак, аналогічний результат можна отримати більш простим шляхом. Аналіз проводять з розрахунком на досить великий обсяг

оброблюваних даних (n → ∞), тому ключове значення має швидкість росту функції складності, а не точна кількість операцій.

В оцінці алгоритмів використовуються спеціальні асимптотичні позначення, що задають такі класи функцій:

- $f(n) = O(g(n))$ – означає, що функція $f(n)$ обмежена згори функцією $c \cdot g(n)$. Тобто існує така константа c, для якої $f(n) \leq c \cdot g(n)$ при достатньо великих n;
- $f(n) = \Omega(g(n))$ - означає, що функція $f(n)$ обмежена знизу функцією $c \cdot g(n)$. Тобто існує така константа c, для якої $f(n) \geq c \cdot g(n)$ для всіх великих $n \geq n_0$;
- $f(n) = \Theta(g(n))$ - означає, що функція $f(n)$ обмежена згори функцією $c_1 \cdot g(n)$, а знизу функцією $c_2 \cdot g(n)$ для всіх великих $n \geq n_0$. Тобто існують такі константи c_1 та c_2, для яких $f(n) \leq c_1 \cdot g(n)$ та $f(n) \geq c_2 \cdot g(n)$.

Правила оцінки складності:

1. $O(k \cdot f) = O(f), \Omega(k \cdot f) = \Omega(f), \Theta(k \cdot f) = \Theta(f)$;
2. $O(f \cdot g) = O(f) \cdot O(g), \Omega(f \cdot g) = \Omega(f) \cdot \Omega(g), \Theta(f \cdot g) = \Theta(f) \cdot \Theta(g)$;
3. $O(f + g) = Max(f, g), \Omega(f + g) = Max(f, g)$,
   $\Theta(f + g) = Max(f, g)$.

Складність алгоритмів визначається для великих обсягів даних, тобто при n → ∞. У зв'язку з цим, при порівнянні трудомісткості двох алгоритмів можна розглянути границю відношення функцій їх складності: $\lim_{n\to\infty} \frac{f(n)}{g(n)}$. Залежно від значення границі можливо зробити висновок щодо швидкостей зростання функцій:

- границя дорівнює константі і не дорівнює нулю, значить функції зростають з однією швидкістю:

$$f(n) = \Theta(g(n));$$

- границя дорівнює нулю, отже g(n) зростає швидше ніж f(n):

$$f(n) = O(g(n));$$

- границя дорівнює нескінченності, то g(n) зростає повільніше ніж f(n):

$$f(n) = \Omega\,(g(n)).$$

Якщо функції досить складні, то замість границі відношення функцій можна аналізувати границю відношення їх похідних (згідно з правилом Лопіталя): $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f\prime(n)}{g\prime(n)}$. Правило Лопіталя можна використовувати якщо функції мають такі властивості:

- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{0}{0}$ або $\frac{\infty}{\infty}$;
- f(n) і g(n) - диференційовані;
- $g'(n) \neq 0$ при $n \to \infty$;
- $\exists \lim_{n\to\infty} \frac{f\prime(n)}{g\prime(n)}$.

Приклад 4.2: Порівняти ефективність алгоритмів з оцінкою складності $O(2^n)$ та $O(n^3)$.

Знайдемо границю відношення функцій з використанням правила Лопіталя

$$\lim_{n\to\infty} \frac{2^n}{n^3} = \lim_{n\to\infty} \frac{2^n \ln 2}{3n^2} = \lim_{n\to\infty} \frac{2^n (\ln 2)^2}{6n} = \lim_{n\to\infty} \frac{2^n (\ln 2)^3}{6} = \infty.$$

Це свідчить про те, що 2^n має більш високу швидкість зростання.

Процес базового аналізу алгоритмів породжує невелику кількість класів функцій, яких достатньо практично для всіх алгоритмів, що розглядаються. Ці класи наведені в порядку зростання складності:

- Функції – константи, $f(n) = 1$ - для алгоритмів, складність яких не залежить від кількості вхідних даних.
- Логарифмічні функції, $f(n) = \log n$ - для програм, в яких велика задача ділиться на дрібні, які вирішуються окремо. Для алгоритмів на кшталт двійкового пошуку.
- Лінійні функції $f(n) = n$ - для одновимірних масивів і циклів, які виконуються n раз.

- Суперлінійні функції $f(n) = n \cdot lg\, n$ – для алгоритмів швидкого сортування та сортування злиттям.
- Квадратична функція $f(n) = n^2$ - в алгоритмах перегляду всіх пар елементів в універсальній множині з n елементів. Виникають в алгоритмах сортування вставками та вибору.
- Кубічна функція, $f(n) = n^3$ – виникає при переліку всіх тріад елементів в універсальній множині з n елементів.
- Показникові функції $f(n) = c^n\ (c > 1)$ – ці функції виникають при переліку усіх підмножин множини з n елементів.
- Факторіальні функції, $f(n) = n!$ – визначають всі перестановки n елементів.

Оцінка просторової (ємнісної) складності

Така оцінка виконується аналогічно з часовою. При цьому якщо для реалізації алгоритму потрібна додаткова пам'ять для однієї, двох або трьох змінних, то ємнісна складність алгоритму буде константною, тобто O (1). Якщо потрібна додаткова пам'ять пропорційна n, то просторова складність дорівнює O(n) і т.д.

Для всіх рекурсивних алгоритмів також важливо поняття ємнісної складності. При кожному виклику метод запитує невеликий об’єм пам'яті, але цей об'єм може значно збільшуватися в процесі рекурсивних викликів.

Контрольні питання

1. Чим характеризується складність алгоритму?
2. У чому полягає асимптотичний аналіз алгоритмів?
3. Які асимптотичні позначення ви знаєте?
4. У яких відомих вам алгоритмів складність є константною, а у яких - лінійна?

Порядок виконання лабораторної роботи

1. Ознайомитись з теоретичною частиною роботи.

2. Функція $f(n)$ є членом однієї з множин Θ (g(n)), O (g(n)), Ω (g(n)). Визначте, членом якої множини є функція $g(n)$:

Варіант	Функції
1.	$f(n) = \log n^2, g(n) = \log n + 5$
2.	$f(n) = \sqrt{n}, g(n) = \log n^2$
3.	$f(n) = \log n^2, f(n) = \log n$
4.	$f(n) = n, g(n) = log^2 n$
5.	$f(n) = n \log n + n, g(n) = \log n$
6.	$f(n) = 10; g(n) = 10n^2$
7.	$f(n) = 2^n, g(n) = 10n^2$
8.	$f(n) = 2^n, g(n) = 3^n$
9.	$f(n) = n + 2\sqrt{n}, g(n) = n^2$
10.	$f(n) = n + \log n, g(n) = \sqrt{n}$

3. Впорядкувати функції в порядку зростання: $n, n - n^3 + 7n^5, n^2 + \lg n, n^3$.

4. Довести, що $n^2 = O(2^n)$.

5. Визначити трудоміскість алгоритму пошуку середнього арифметичного елементів одномірного масиву.

Лабораторна робота №5
Експериментальний метод оцінки складності алгоритму

Мета: експериментально визначати складність алгоритмів.

Теоретичні відомості

Експериментальний метод заснований на вимірі часу виконання алгоритму при деякому наборі даних (тесті). При цьому використовуються стандартні засоби мови програмування, що дозволяють визначити системний час комп'ютера. Наприклад, для середовища C++ можна використовувати std::chrono::steady_clock.

Для оцінки трудомісткості деякого алгоритму достатньо зафіксувати час перед його виконанням та наприкінці, наприклад, за допомогою наступного фрагменту:

```
#include <chrono>
using namespace std::chrono;
int main()
{ auto start = steady_clock::now();
// Фрагмент програми, що реалізує
// Досліджуваний алгоритм
auto finish = steady_clock::now();
std::cout<<duration_cast<milliseconds>(finish - start).count();
```

Швидкодія сучасних процесорів з тактовою частотою кілька Гігагерц має порядок кількох десятків мільярдів операцій на секунду. При цьому час виконання простих алгоритмів може бути дуже малим, а вимір його пропонованим методом матиме велику похибку. Для її зменшення здійснюють багаторазове виконання досліджуваного фрагмента всередині

Контрольні питання

1. Як оцінюється складність експериментальним методом?
2. Чи збігаються результати експериментальної та верхньої оцінок і, якщо ні, то скільки вони відрізняються?

Порядок виконання лабораторної роботи

1. Скласти програму, яка формує одновимірний масив випадкових чисел. Знайти суму парних чисел та суму непарних чисел масиву.
2. Теоретично розрахувати часову складність алгоритму та побудувати графік залежності функції складності від обсягу вхідних даних.
3. Оцінити експериментальним способом час виконання алгоритму.
4. Вимірювання необхідно повторити п'ять разів для різного обсягу вихідних даних Кількість повторень алгоритму в кожному вимірі має бути однаковоюм.
5. Побудувати графік залежності часу виконання від обсягу вхідних даних.

Лабораторна робота №6
Розробка рекурсивних алгоритмів

Мета: розробка програм, що реалізують різні рекурсивні алгоритми та оцінка їх часової та просторової складності.

Теоретичні відомості

Під рекурсією розуміють спосіб завдання функції через саму себе. У програмуванні під рекурсивною процедурою (функцією) розуміють спосіб звернення процедури (функції) до себе.

Рекурсія – це виклик методу з самого себе безпосередньо (проста рекурсія) чи через інші методи (складна чи непряма рекурсія).

Серед широкого класу задач зручно представляти з використанням рекурсивних процедур (функцій) ті завдання, які зводяться до завдань того ж типу, але меншої розмірності.

Кількість вкладених викликів функції чи процедури називається глибиною рекурсії.

Загальна методика аналізу рекурсії містить три етапи:

1. Параметризація задачі, що полягає у виділенні різних елементів, від яких залежить розв’язання, зокрема, розмірності розв'язуваної задачі. Після кожного рекурсивного виклику розмірність має зменшуватися.

2. Пошук тривіального випадку та його вирішення. Як правило, це ключовий етап у рекурсії, розмірність задачі при цьому часто дорівнює 0 або 1.

3. Декомпозиція загального випадку, що має на меті звести завдання до одного або кількох завдань того ж типу, але меншої розмірності.

Розглянемо кілька прикладів простої рекурсії, коли метод викликає сам себе.

Приклад 6.1. Обчислити факторіал числа n із використанням рекурсії.

Факторіал числа n (позначається n!) - Добуток всіх натуральних чисел від 1 до n включно, тобто $n! = 1 \cdot 2 \cdot \ldots \cdot n$.

Наприклад, $5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 4! \cdot 5$. У загальному випадку формулу для знаходження факторіалу можна записати таким чином:

$$n! = \begin{cases} (n-1)! \cdot n, \text{якщо } n > 1 \\ 1, \text{якщо } n = 1. \end{cases}$$

Функцію можна реалізувати таким чином:

```
unsigned long long int Factorial(int n)
{
if (n==1) return 1;
else
return n*Factorial(n-1);}
```

Якщо функція А викликає функцію В, а та у свою чергу викликає А, це називається складною рекурсією. При цьому виявляється, що описана перша процедура повинна викликати ще не описану. Щоб це було можливо, потрібно використовувати опис функції В до використання.

Приклад 6.2. Обчислити $\frac{x^n}{n!}$

```
int pow(int,int);
double calc(int x,int n)
{
return (double)pow(x,n)/n;
}
int pow(int x, int n)
{if (n==1) return x;
return x*calc(x,n-1);}
```

Метод підстановки в аналізі рекурсівних алгоритмів

Основний метод побудови рекурсивних алгоритмів – метод декомпозиції. Ідея методу полягає у поділі завдання на частини меншої розмірності, отримання рішення для отриманих частин та об'єднання рішень.

В загальному вигляді, якщо відбувається поділ завдання на b підзадач, яке призводить до необхідності рішення a підзадач розмірністю n/b, то загальний вигляд функції трудомісткості має вигляд:

$$f(n) = a \cdot f(n/b) + d(n) + U(n),$$

де:

a – кількість підзадач,

n/b – їх розмір,

d(n) – трудомісткість алгоритму поділу задачі на підзадачі,

U(n) – трудомісткість алгоритму поєднання отриманих рішень.

Для обчислення значення такого виразу використовують метод підстановки. Він полягає в послідовній заміні рекурентної частини у виразі для отримання нових виразів. Заміна проводиться до тих пір, поки не вдасться визначити загальний принцип і записати його у вигляді нерекурентної формули. Для доказу цього припущення використовують метод математичної індукції.

Приклад 6.3. Визначити максимум в масиві за допомогою рекурсії

```
int max(int*a,int n, int k, int m)
{
if(k==n) return m;
if(m<a[k])
m=a[k];
m=max(a, n, k+1, m);}
```

Проаналізуємо складність даного алгоритму.

$$f_n = O(1) + f_{n-1} = 2 \cdot O(1) + f_{n-2} = 3 \cdot O(1) + f_{n-3}$$

Будемо вважати, що $f_n = f_{n-k} + k \cdot O(1)$, але тоді $f_n = f_0 + n \cdot O(1) = O(n)$.

Доведемо коректність припущення, зробленого з оцінки трудомісткості функції пошуку ($f_n = (n+1) \cdot O(1)$) з використанням методу математичної індукції:

1. $f_1 = 2 \cdot O(1)$ вірно за умовою;

2. Припустимо вірність $f_n = (n + 1) \cdot O(1)$;

3. Треба довести, що $f_{n+1} = \big((n + 1) + 1\big) \cdot O(1) = (n + 2) \cdot O(1)$;

Підставимо n+1 у рекурентне співвідношення: $f_{n+1} = O(1) + f_n$.

В правій частині виразу можна замінити на основі індуктивної гіпотези: $f_{n+1} = O(1) + (n + 1) \cdot O(1) = (n + 2) \cdot O(1)$.

Твердження доведено.

Контрольні питання

1. Що таке рекурсія?
2. Чим відрізняється проста рекурсія від складної?
3. Чим можна замінити рекурсію?
4. Як можна обчислити факторіал без використання рекурсивного алгоритму?
5. Як можна оцінити ємнісну складність рекурсивного алгоритму?

Порядок виконання лабораторної роботи

1. Розробити програми з функціями, наведеними в теоретичній частині.
2. Розробити програму алгоритму Евкліда з використанням рекурсії.
3. Отримати експериментальну та теоретичну оцінку часу виконання алгоритму та максимальне n знаходження факторіалу.
4. Побудувати ряд чисел Фібоначі до заданого значення x з використанням рекурсії.

Лабораторна робота №7
Напівстатична структура даних стек

Мета роботи: вивчити алгоритми роботи зі структурою даних стек.

Теоретичні відомості

Структура даних – це множина елементів даних і відношення між ними. При цьому під елементами даних розуміють як прості дані так і структуру даних. Під відношеннями між даними розуміють функціональні зв'язки з-поміж них і покажчики на те, де перебувають ці дані.

Структури даних класифікуються:

1. За мінливістю структури у часі або у процесі виконання програми:

- статичні структури – це структури, які змінюються після остаточного виконання програми (списки, масиви, рядки, вектори);

- напівстатичні структури (стеки, деки, черги);

- динамічні структури – це структури, в яких відбувається повна зміна під час виконання програми.

2. За зв'язаністю даних у структурі:

- якщо дані у структурі пов'язані дуже слабко, такі структури називаються непов'язаними (вектор, масив, рядки, стеки);

- якщо дані у структурі пов'язані, такі структури називаються пов'язаними (пов'язані списки)

3. За впорядкованістю структури:

- лінійні (вектори, масиви, стеки, деки, записи);

- нелінійні (багатозв'язні списки, деревоподібні структури, графи).

Стек (Stack) – структура типу LIFO (Last In, First Out) - останнім увійшов, першим вийде. У структурі виконуються наступні умови:

- новий елемент приєднується завжди тільки з одного боку послідовності;

- доступ до елемента здійснюється завжди з того боку послідовності, до якого приєднується елемент;

- елемент зберігається в послідовності до моменту його виклику.

До основних операцій зі стеком (Таблиця 7.1) відносяться:

-формування стека (додавання елемента в стек);

-обробка елементів стека (перегляд, пошук, видалення);

-звільнення пам'яті, зайнятої стеком.

Шаблон stack міститься в заголовному файлі <stack>. Створення стеку відбувається на основі зазначеного шаблону:

stack<Type>StackName,

де

Type - тип даних елемента для збереження у стеку;

StackName –ім’я стеку.

Таблиця 7.1

Операції зі стеком

Функції	**Опис**	**Приклад реалізації**
empty	Перевіряє, чи порожній стек.	StackName.empty( )
push	Додає елемент у верхню частину стеку.	StackName.push(10)
pop	Видаляє елемент із верхньої частини стеку. Для застосування функції- стек має бути непорожнім.	StackName.pop( )
size	Повертає кількість елементів у контейнері стек.	StackName.size( )
top	Повертає посилання на елемент у верхній частині стек	StackName.top( )

Стек може бути реалізований на базі статичного масиву за допомогою функцій:

1. Створення структури стеку

```
struct Stack
{ int A [n];
int last; };
```

2. Перевірка стеку на наявність елементів

```
bool empty(Stack &s)
```

```
{ return (s.last==0); }
```

3. Додавання елементу до стеку

```
void push(Stack &s, int x)
{ if (s.last==n)
{ cout<<"Stack Overflow"; return;}
s.A[s.last++]=x; }
```

4. Видалення елементу зі стеку

```
int pop(Stack&s)
{ return s.A[--s.last];}
```

5. Виведення верхнього елементу стеку

```
int top(Stack &s)
{ int i=s.last-1; return s.A[i];}
```

6. Визначення розміру стеку

```
int size(Stack &s)
{ return (s.last); }
```

Контрольні питання

1. Що таке стек?
2. Які основні функції по роботі зі стеком?
3. Як позначається операція додавання елемента в стек?
4. Як позначається операція видалення верхнього елемента зі стека?

Порядок виконання лабораторної роботи

1. Реалізувати стек та операції з ним з використанням шаблону stack.
2. Реалізуйте стек та методи роботи з ним на базі статичного масиву.
3. Розв’язати задачу з індивідуального завдання.

Індивідуальні завдання

1. Створити стек цілих чисел. Ввести 6 елементів. В стек потрапляють тільки від'ємні числа. Перевірити розмір стеку.

2. Створити стек цілих чисел. Ввести 6 елементів. Видалити 2 елемента зі стеку. Перевірити розмір стеку.

3. Створити стек символів. Ввести еталонний символ. Вводити символи в стек до зустрічі еталонного елементу. Вивести розмір стеку.

4. Створити два стека цілих чисел. Ввести елементи в стеки таким чином, що від'ємні числа потрапляють у перший стек, а додатні – у другий. Вивести розміри стеків.

5. Створити стек символів. Ввести елементи таким чином, щоб вони потрапляли по черзі в стеки. Вивести розмір стеків.

6. Створити стек. Ввести елементи в стек. Ввести кількість елементів для видалення зі стеку і видалити їх. Вивести розмір стеку.

7. Створити стек цілих чисел. Елементи ввести в стек. Якщо елемент вершини стеку співпадає з елементом, що вводиться, то видалити його. Вивести розмір стеку.

8. Створити два символьних стека. Ввести елементи стеку. Великі літери потрапляють до першого стеку, а маленькі – до другого. Вивести розміри стеків.

9. Створити стек цілих чисел. Заповнити стек елементами. Якщо елемент вершини стеку є від'ємне число, то видалити його. Вивести розмір стеку.

10. Створити стек цілих чисел. Заповнити стек елементами. Якщо елемент вершини стеку є від'ємне число, то додати ще два елемента. Вивести розмір стеку.

Лабораторна робота №8
Базова структура даних. Черга

Мета роботи: вивчити алгоритми роботи зі структурою даних черга.

Теоретичні відомості

Черга (Queue**)** – структура типу FIFO (First In, First Out — «перший прийшов — перший пішов»).

У структурі виконуються наступні умови:

- новий елемент приєднується завжди з одного і того самого боку структури;

- доступ до елементів або їх вилучення завжди здійснюється з іншого боку структури.

Розрізняють такі види черг:

- звичайна черга;
- кільцева черга. У такій черзі елемент, що виходить із початку черги, буде поміщений у її кінець;
- чергу з пріоритетами. У такій черзі елементи розміщуються за пріоритетами (ваговими коефіцієнтами). Першим із черги виходить елемент із найвищим пріоритетом.

Типовими операціями для черги є:

- визначення кількості елементів у черзі;
- додавання елемента в кінець черги;
- вибірка елемента з початку черги;
- перевірка чи порожня черга; результатом є значення «істина», якщо черга порожня, і «хибно» у протилежному випадку.

Черга може бути реалізована за допомогою структури queue. Для роботи із чергою необхідно підключити заголовок #include <queue>. Функції для роботи з чергою наведені у таблиці 8.1. Складність операції додавання елементів у чергу та їх вилучення – O(1). Створення черги відбувається на основі queue шаблону:

queue <Type> QueueName,

де

Type - тип даних елемента для збереження у черзі;

QueueName –ім’я черги.

Таблиця 8.1

Операції зі чергою

Функції	**Опис**	**Приклад реалізації**
empty	Перевіряє, чи порожня черга. Повертає значення true, якщо черга порожня; false – черга непорожня.	QueueName.empty( )
pop	Видаляє елемент з передньої частини черги. Щоб застосувати цю функцію, черга має бути непорожньою.	QueueName.pop( )
push	Додає елемент до задньої частини черги.	QueueName.push(10)
size	Повертає кількість елементів у черзі.	QueueName.size( )
back	Повертає посилання на останній доданий елемент до кінця черги	QueueName.back( )
front	Повертає посилання на перший елемент у черзі	QueueName.front( )

Звичайна черга може бути реалізована на базі масиву за допомогою функцій:

1. Створення структури черги

```
struct queue
{int q[N];
int start;
int finish;
queue()
{start = 0;
finish = 0;}
```

2. Функція додавання елементу в кінець черги

```
void push(int n)
{q[finish] = n;
finish++;}
```

3. Функція видалення елементу з черги

```
int pop()
{int a = q [start];
start++;
return a;}
```

4. Функція визначення розміру черги

```
int size()
{return finish - start;}
```

5. Функція видалення елементів із черги

```
void clear()
{finish = 0;
start = 0;}
```

Для перегляду елементів в початку черги можна використати функцію top(), а для елемента кінця черги - back():

```
int front(){ return a[start]; }
int back(){ return a[finish -1]; }
```

Для реалізації кільцевої черги можна скористатися наступним алгоритмом: оголосити два покажчики start (перший елемент черги) та finish (останній елемент черги). При ініціалізації кругової черги їм надають значення -1. Після додавання першого елементу start приймає значення 0, а значення finish збільшуємо на 1. Коли покажчик опиниться наприкінці черги, він має перемістится на її початок. Додаємо новий елемент на ту позицію, куди вказує finish.

Для перевірки чи заповнена черга можна скористатися одним із наступних співвідношень:

1. start = 0 && finish == N - 1
2. start = finish + 1.

Контрольні питання

1. Яка структура називається чергою?
2. Який елемент називають «головою» черги?
3. Який елемент називають «хвостом» черги?
4. Які види черги розрізняють?

Порядок виконання лабораторної роботи

1. Реалізувати чергу та операції з ним з використанням шаблону queue.
2. Реалізуйте чергу та методи роботи з нею на базі масиву.
3. Реалізувати кільцеву чергу на базі масиву.

Лабораторна робота №9
Базова структура даних. Дек

Мета роботи: вивчити алгоритми роботи зі структурою даних дек.

Теоретичні відомості

Дек (Deque)- структура даних, в якій додавання нових елементів і видалення існуючих проводиться з обох кінців. Ця структура підтримує як FIFO, і LIFO, тому у ньому можна реалізувати як стек, і чергу.

До основних операцій зі структурою дек відносяться:

- Додавання елемента на початок та кінець.
- Видалення елементу з початку та кінця.
- Звільнити структуру
- Перевірка на порожність дека.

Дек можна реалізувати за допомогою шаблону deque. Цей шаблон міститься у заголовочному файлі #include <deque>. Для генерації деку використовують наступний синтаксис:

```
deque< Type > DequeName,
```

де

Type - тип даних елемента для збереження у деці;

DequeName –ім’я деку.

Функцію deque можна ініціалізувати в такий спосіб:

```
deque < int> DequeName ={1, 3, 7, 0, 5, 8};
deque<int> DequeName {1,3,7,0,5,8};
```

Таблиця 9.1

Основні операції із деком

Функції	Опис	Приклад реалізації
back	Повертає посилання на останній елемент деку.	DequeName.back( )
begin	Повертає ітератор, що звертається до першого елемента в контейнері деку.	DequeName.begin( )
clear	Стирає всі елементи в деку.	DequeName.clear( )
empty	Повертає true, якщо дек не містить елементів, і false якщо він містить один або кілька елементів.	DequeName.empty( )
end	Повертає ітератор, який звертається до місця, наступного за останнім елементом у контейнері дек	DequeName.end( )
front	Повертає посилання на перший елемент деку.	DequeName.front( )
erase	Видаляє елемент або діапазон елементів із зазначених позицій у деку.	DequeName.erase(iterator first, iterator last)
max_size	Повертає максимальну довжину деку.	DequeName.max_size( )
pop_back	Видаляє елемент у кінці деку.	DequeName.pop_back( )
pop_front	Видаляє елемент на початку деку.	DequeName.pop_front( )
push_back	Додає елемент у кінець деку.	DequeName.push_back(1)
push_front	Додає елемент на початок деку.	DequeName

Дек може бути реалізований на базі масиву.

1. Створення деку.

```
struct deque
{int d[N];
int  s = 0, start = 0, end = 0;}
```

2. Функція додавання елементу на початок деку:

```
void push_front(int n)
{if (s == N) { cout << "Full\n"; }
else if {(s == 0) { d[start] = n; s++; }
else { if (start == 0) { start = N - 1; d[start] = n; s++; }
else { start--; d[start] = n; s++; } }}
```

3. Функція додавання елементу в кінець деку:

```
void push_back(int n)
{if (s == N) { cout << "Full\n"; }
else if (s == 0) { d[end] = n; s++; }
else {if (end == N-1) { end = 0; d[end] = n; s++; }
else { end++; d[end] = n; s++; } }}
```

4. Функція видаляє елемент на початку дека

```
int pop_front()
{int b;
if (s != 0 && (start == end)) { b = d[start]; s--; return b; }
else if (s == 0) { return -1; }
else { b = d[start];
if (start == N - 1) { start = 0; s--; }
else { start++; s--; }
return b;}}
```

5. Функція видаляє елемент в кінці деку

```
int pop_back()
{int b;
if (s != 0 && (start == end)) { b = d[end]; s--; return b; }
else if (s == 0) { return -1; }
else {b = d[end];
if (end == 0) { end = N - 1; s--; }
else { end--; s--; }
return b;}}
```

6. Функції звернення до елемента на початку, в кінці деку

```
int front() { if (s == 0) return -1; else return d[start]; }
int back() { if (s == 0) return -1; else return d[end]; }
```

7. Функція знаходження розміру деку

```
int size() { return s; }
```

8. Функція видалення елементів в деку

```
void clear()
{ s= 0;
start = end = 0;}
```

Контрольні питання

1. Яка структура називається деком?
1. Які основні операції можна виконувати з деком?
2. До якого типу структурних даних відноситься дек?

Порядок виконання лабораторної роботи

1. Реалізувати дек та операції з ним з використанням шаблону deque.
2. Реалізувати дек та методи роботи з ним за допомогою масиву.
3. Сформувати два дека. Об’єднати ці деки в один.

Лабораторна робота №10
Алгоритми сортування

Мета роботи: ознайомитись з алгоритмами сортування масиву.

Теоретичні відомості

Алгоритмом сортування називається алгоритм для впорядкування деякої множини елементів. Зазвичай під алгоритмом сортування мають на увазі алгоритм упорядкування безлічі елементів за зростанням або спаданням.

У разі наявності елементів з однаковими значеннями в упорядкованій послідовності вони розташовуються поруч один за одним у будь-якому порядку. Такий алгоритм сортування називається нестійким. Однак іноді буває корисно зберігати початковий порядок елементів із однаковими значеннями. Тоді такий алгоритм відноситься до стійких алгоритмів сортування.

У алгоритмах сортування лише частина даних використовується як ключ сортування. Ключем сортування називається атрибут (або кілька атрибутів), за значенням якого визначається порядок елементів. Ключ частково або повністю збігається з даними.

При класифікації алгоритмів сортування враховують такі основні показники:

• **час сортування** – основний параметр, який характеризує швидкодію алгоритму;

• **стійкість** – це параметр, який відповідає за те, що сортування не змінює взаємне розташування рівних елементів.

• **природність поведінки** – параметр, який вказує на ефективність методу обробки вже відсортованих, або частково відсортованих даних. Алгоритм веде себе природньо, якщо враховує цю характеристику вхідної послідовності та працює краще;

• **пам'ять** – один із параметрів, який характеризується тим, що низка алгоритмів сортування потребує виділення додаткової пам'яті під тимчасове зберігання даних:

- внутрішнє сортування – це алгоритм сортування, який у процесі впорядкування даних використовує лише оперативну пам'ять (ОЗП) комп'ютера.
- зовнішнє сортування – це алгоритм сортування, який під час проведення впорядкування даних використовує зовнішню пам'ять, зазвичай жорсткі диски.

До алгоритмів стійкого сортування відносять:

- Сортування бульбашкою (Bubble sort);
- Сортування вставками (Insertion sort);
- Сортування злиттям (Merge Sort);
- Сортування підрахунком (Counting Sort).

До алгоритмів нестійкого сортування відносять:

- Сортування вибором (Selection sort)
- Сортування Шела (Shell sort)
- Пірамідальне сортування (Heapsort)
- Швидке сортування (Quicksort)

Алгоритм сортування бульбашкою

Алгоритм полягає в повторюваних проходах по масиву, що сортується. На кожній ітерації послідовно порівнюються сусідні елементи, і якщо порядок у парі невірний, то елементи міняють місцями. За кожен прохід масивом як мінімум один елемент встає на своє місце, тому необхідно зробити не більше $n-1$ проходів, де n розмір масиву, щоб відсортувати масив.

Сортування бульбашкою має такі властивості:

- Висока обчислювальна складність необхідно зробити $O(N^2)$ порівнянь та $O(N^2)$ перестановок. Якщо дані вже відсортовані, необхідно зробити всього O(N) операцій;

- Не потребує використання додаткової пам'яті (O(1));
- Стійкий алгоритм;
- Простий у реалізації.

Алгоритм сортування включенням

На кожному кроці алгоритму вибирається один із вхідних елементів та вставляється на потрібну позицію вже відсортованої частини масиву, алгоритм виконується доки набір вхідних даних не буде вичерпаний. Метод вибору чергового елемента з вихідного масиву довільний. Зазвичай (і з метою отримання сталого алгоритму сортування), елементи вставляються за порядком їх появи у вхідному масиві.

Наведений нижче алгоритм використовує цю стратегію вибору.

```
void InsertSort(int* A, int n)
{int key, i;
   for (int k = 1; k < n; k++) {
      key = A[k];
      i = k - 1;
      while ((i >= 0)&&(A[i]>key)) {
         A[i + 1] = A[i];
         i = i - 1;
      }
      A[i + 1] = key;
   }
}
```

Алгоритм сортування включеннями має такі характеристики:

- Висока обчислювальна складність: необхідно зробити $O(N^2)$ порівнянь та $O(N^2)$ перестановок. Якщо дані вже відсортовані, необхідно зробити всього O(N) операцій;
- Не потребує використання додаткової пам'яті (O(1));
- Стійкий;
- Простий у реалізації.

Алгоритм сортування Шелла

Сортування Шелла є вдосконаленим варіантом сортування включенням. Ідея даного методу полягає у порівнянні елементів, що стоять не тільки поруч, але й на певній відстані один від одного. Є велика кількість модифікацій, що відрізняються вибором кроку відстані кожної ітерації.

Функція алгоритму сортування Шелла:

```
void Shellsort(int* A, int n)
{
   int d, count, i, j;
   d = n;
   d = d / 2;
   while (d > 0)
   {
      for (i = 0; i<(n-d);i++)
      {   j = i;
      while (j >= 0 && A[j] > A[j + d])
      {
      count = A[j];
      A[j] = A[j + d];
      A[j + d] = count;
      j--;}
         }
      d = d / 2;
   }
}
```

Сортування Шелла має такі властивості:

- Обчислювальна складність у середньому дорівнює O($n^{3/2}$)). Якщо дані вже відсортовані, необхідно зробити всього O($n \cdot \lg n$) операцій;
- Не потребує використання додаткової пам'яті (O(1));
- Нестійкий.

Алгоритм сортування злиттям

В алгоритмі вхідний масив розбивається на частини однакового розміру. Кожен з підмасивів сортується окремо, за цим самим же принципом, тобто проводиться повторне розбиття до тих пір, поки довжина підмасиву не досягне одиниці. Кожен одиничний масив вважається відсортованим. Після цього здійснюється злиття всіх підмасивів в один, у результаті отримуємо відсортовані дані.

Рекурсивна функція алгоритму сортування злиттям має вигляд:

```
void Mergesort(int* A, int n)
{ if (n < 2)
      return;
   Mergesort(A, n / 2);
   Mergesort(&A[n / 2], n - (n / 2));
   int* buf = new int[n];
   int idbuf = 0, idl = 0, idr = n / 2;
   while ((idl < n / 2) && (idr < n))
      if (A[idl] < A[idr])
         buf[idbuf++] = A[idl++];
      else
         buf[idbuf++] = A[idr++];
   while (idl < n / 2)
      buf[idbuf++] = A[idl++];
   while (idr < n)
      buf[idbuf++] = A[idr++];
   for (idl = 0; idl < n;idl++)
      A[idl] = buf[idl];
   delete[]buf;}
```

Сортування злиттям має такі властивості:

- Обчислювальна складність у середньому дорівнює O($N \cdot lgN$);
- Вимагає використання додаткової пам'яті для злиття послідовностей (O(N));
- Стійкий.

Алгоритм сортування підрахунком

Алгоритм сортування підрахунком є одним із найоптимальніших рішень, якщо кількість можливих різних елементів у множині відносно невелика.

Головна ідея алгоритму: створюємо масив довжини діапазону k, кожен елемент якого вказує кількість елементів, які дорівнюють даному. Проходимо по масиву і підраховуємо кількість входжень кожного числа.

Функція алгоритму сортування підрахунком:

```
void CountingSort(int* A, int n)
{
    int max = INT_MIN, min = INT_MAX;
    for (int i = 0; i < n; i++) {
        if (A[i] > max)
            max = A[i];
        if (A[i] < min)
            min = A[i];
    }
    int* c = new int[max + 1 - min];
    for (int i = 0; i < max + 1 - min; i++) {
        c[i] = 0;
    }
    for (int i = 0; i < n; i++) {
        c[A[i] - min] = c[A[i] - min] + 1;
    }
    int i = 0;
    for (int j = min; j < max + 1; j++) {
        while (c[j - min] != 0) {
            A[i] = j;
            c[j - min]--;
            i++;
        }
    }
}
```

Сортування підрахунком має такі властивості:

- Обчислювальна складність у найкращому дорівнює O(n),у середньому - O(n+k), у найгіршому - O(k);
- Вимагає використання додаткової пам'яті O(k);
- Стійкий.

Алгоритм сортування вибором

В алгоритмі масив поділяється на дві частини: масив з відсортованими елементами та масив з елементами, які потрібно відсортувати. Спочатку знаходиться найменший елемент у другому масиві. Він додається наприкінці першого. Таким чином, алгоритм поступово формує масив від найменшого до найбільшого. Так відбувається доти, доки не буде відсортовано масив.

Функція алгоритму сортування вибором:

```
void Selectionsort(int* A, int n)
{
   int count, key;
   for (int i = 0; i < n - 1; i++)
   {
      count = A[i]; key = i;
      for (int j = i + 1; j < n; j++)
         if (A[j] < A[key]) key = j;
      if (key != i)
      {
         A[i] = A[key];
         A[key] = count;
      }
   }
}
```

Сортування вставками має такі властивості:

- Висока обчислювальна складність необхідно зробити $O(n^2)$ порівнянь та O(n) перестановок
- Не потребує використання додаткової пам'яті (O(1));
- Нестійка (є стійкі модифікації);
- Простота у реалізації.

Алгоритм пірамідального сортування

За основу пірамідального сортування береться спеціальний тип бінарного дерева, що називається пірамідою.

Алгоритм складається з двох частин: побудови піраміди та самого сортування.

Побудова піраміди (рис. 10.1) починається з кореня, а потім послідовно заповнюються рівні дерева зліва направо. Виходить, що корінь дерева - це нульовий елемент масиву, його нащадки - це перший і другий елемент і, відповідно, у лівої гілки нащадки - це 3 і 4 елементи, а у правого - 5 і 6 і так далі.

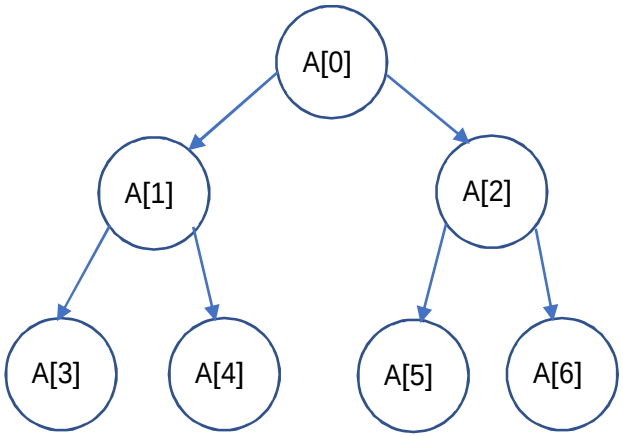


Рис.10.1 Побудова піраміди

При сортуванні за зростанням потрібно перетворити це дерево таким чином, щоб у корені дерева знаходився максимальний елемент, а значення в кожному вузлі було більше ніж значення його нащадків. Якщо ця умова не виконується, то знаходимо максимальний елемент серед нащадків та міняємо його місцем з батьком. Так проходимося по кожному рівню дерева знизу вгору і в результаті отримуємо відсортоване дерево, в корені якого знаходиться максимальне число. На цьому етапі найбільший елемент зберігається в корені дерева. Замінюємо його на останній елемент піраміди, а потім зменшуємо розмір на 1. Таким чином перетворюємо в дерево з новим коренем. Повторюємо вищезгадані кроки, поки розмір купи більше 1. При сортуванні за спаданням коренем дерева повинен стати мінімальний елемент.

Функція побудови піраміди та алгоритму пірамідального сортування має вигляд:

```
void heapify(int* A, int n, int i)
{
   int largest = i;
   int l = 2 * i + 1; // левый = 2*i + 1
   int r = 2 * i + 2; // правый = 2*i + 2
```

```
    if (l < n && A[l] > A[largest])
        largest = l;
    if (r < n && A[r] > A[largest])
        largest = r;
    if (largest != i)
    {
        swap(A[i], A[largest]);
        heapify(A, n, largest);
    }
}
void heapSort(int* A, int n)
{
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(A, n, i);
    for (int i = n - 1; i >= 0; i--)
    {
        swap(A[0], A[i]);
        heapify(arr, i, 0);
    }
}
```

Пірамідальне сортування має такі властивості:

• Обчислювальна складність у середньому дорівнює O($n \cdot \lg n$);

• Не потребує використання додаткової пам'яті (O(1));

• Нестійкий;

• Складний у реалізації.

Алгоритм швидкого сортування

Швидке сортування порівнює всі елементи масиву з опорним елементом і тим самим ділить масив на дві частини - в одну потрапляють менші числа опорного, а в іншу - більші опорного. Потім кожна з цих двох частин також піддається аналогічному сортуванню, і так доти, поки чергова частина не буде складатися з єдиного елемента.

Рекурсивна функція швидкого сортування має вигляд:

```
void quickSort(int* A, int n)
{
    int mid = n / 2;
    int i = 0, j = n-1;
    do
    {
```

```
        while (A[i] < A[mid])
            i++;
        while (A[j] > A[mid])
            j--;

        if (i <= j)
        {
            swap(A[i], A[j]);
            i++;
            j--;
        }
    } while (i <= j);

    if (j > 0)
        quickSort(A, j+1);
    if (n > i)
        quickSort(&A[i],n-i);
}
```

Швидке сортування має такі властивості:

• Обчислювальна складність у середньому дорівнює O($n \cdot lg(n)$), у гірших випадках, за великої кількості неунікальних ключів може сягати O(n^2)

• Вимагає використання додаткової пам'яті (у середньому O(lg n), у найгіршому O(n));

• Нестійкий.

Контрольні питання

1. . Що таке сортування?
2. За якими ознаками виконується класифікація алгоритмів сортування?
3. Які основні відмінності сортування включенням від «бульбашкового»?
4. Які основні відмінності сортування злиттям від «бульбашкового»?
5. Які основні відмінності сортування злиттям та включенням?
6. Які відмінні риси швидкого сортування?

7. Які особливості сортування Шелла?

8. Які відомі Вам методи сортування часова складність, яких залежить від обсягу пам'яті, що використовується?

Порядок виконання лабораторної роботи

1. Згенерувати одномірний масив з n випадкових чисел, де n вводиться з клавіатури.

2. Реалізувати алгоритм сортування бульбашкою.

3. Розробити та налагодити програми, з використанням алгоритмів сортування.

4. Впорядкувати даний масив і порівняти часову складність даних алгоритмів сортування.

5. Побудувати графіки складності.

Лабораторна робота №11
Алгоритми пошуку

Мета роботи: вивчити методи лінійного та бінарного пошуку.

Теоретичні відомості

Завдання пошуку полягає у визначенні одного або кількох елементів у множині з певною властивістю. Ця властивість може бути абсолютною або відносною.

Таким чином, завдання пошуку мають наступні кроки:

1) визначення властивості елемента;

2) порівняння властивостей елемента з еталонною властивістю (для абсолютних властивостей) або порівняння властивостей двох елементів (для відносних властивостей);

3) перебір елементів множини.

Лінійний пошук. Алгоритм пошуку полягає в знаходженні потрібного елементу в невідсортованому масиві шляхом перегляду всіх елементів.

Перед алгоритмом пошуку стоїть завдання визначення місцезнаходження ключа, тому він повертає індекс запису, що містить потрібний ключ. Якщо пошук завершився невдачею (ключове значення не знайдено), алгоритм пошуку зазвичай повертає значення індексу, що перевищує верхню межу масиву.

Аналіз найгіршого випадку. У алгоритму лінійного пошуку два найгірші випадки. У першому випадку шуканий елемент стоїть останнім у масиві. У другому його зовсім немає у масиві. В обох випадках алгоритм виконає n порівнянь, де n – число елементів у масиві.

Аналіз середнього випадку. Цільове значення може займати одне з N можливих положень. Будемо припускати, що ці положення рівноймовірні, т. т. можливість зустріти кожне їх дорівнює $\frac{1}{n}$. Отже, для знаходження і-го елемента A[i] потрібно і порівнянь. В результаті для складності в середньому випадку отримуємо рівність:

$$T(n) = \frac{1}{n}\sum_{i=1}^{n} i = \frac{1}{n} \cdot \frac{n(n+1)}{2} = \frac{n+1}{2}.$$

Бінарний пошук застосовується до відсортованого масиву елементів. Суть цього методу полягає в розбитті інтервалу на дві половини поки не дійде до двох півінтервалів, розміри яких дорівнюють одиниці. Цільове значення порівнюється з середнім елементом відсортованого масиву. При порівнянні елементів можливий один із трьох результатів: значення рівні, цільове значення менше елементу масиву, або цільове значення більше за елемент масиву. Перший випадок є найкращім і пошук завершується. У інших випадках є можливість відкинути половину списку. Насправді, коли цільове значення менше середнього елемента і воно є у масиві, то знаходиться перед цим середнім елементом. Якщо цільове значення більше за середній елемент, то знаходиться після цього елементу.

Аналіз найгіршого випадку. Оскільки алгоритм щоразу поділяє список навпіл, припускатимемо при аналізі, що $n = 2^k - 1$ для деякого k. Зрозуміло, що на деякому проході цикл має справу зі списком $2^j - 1$ елементів. Остання ітерація циклу проводиться, коли розмір частини, що залишилася, стає рівною 1, а це відбувається при j = 1 (оскільки $2^1 - 1 = 1$). Це означає, що за $n = 2^k - 1$ число проходів не перевищує k. Отже, у найгіршому випадку число проходів дорівнює $k = \log_2(n+1)$.

Аналіз середнього випадку. Можливі два випадки. У першому випадку цільове значення точно міститься у масиві, а в другому його може там і не бути. У першому випадку цільове значення n можливих положень, кожне з яких рівноймовірне. Дані можна представити у вигляді бінарного дерева порівнянь, яке відповідає бінарному пошуку. Для елементів у вузлах рівня i потрібно i порівнянь. На рівні i є 2^{i-1} вузлів, і при $n = 2^k - 1$ в дереві k рівнів. Число порівнянь для всіх можливих випадків можна підрахувати, підсумувавши добуток числа вузлів на кожному рівні на число порівнянь на цьому рівні.

$$T(n) = \frac{1}{n}\sum_{i=1}^{k} i2^{i-1}.$$

Можна показати, що це співвідношення після перетворень буде мати вигляд

$$T(n) = \log_2(n + 1) - 1.$$

У другому випадку число можливих положень елемента у масиві, як і раніше, дорівнює n. Проте цього разу є ще n + 1 можливість для цільового значення знаходження в масиві. Число можливих випадків дорівнює n + 1, оскільки цільове значення може бути менше першого елементу масиву, більше першого, але менше другого, більше другого, але менше третього, і так далі, аж до того, що цільове значення може бути більше n-го елементу. У кожному з цих випадків відсутність елемента у масиві виявляється після k порівнянь. Усього у обчисленні бере участь 2n + 1 можливих випадків.

$$T(n) = \frac{1}{2n+1}\left(\sum_{i=1}^{k} i2^{i-1} + (n + 1)k\right).$$

Після перетворень це співвідношення буде мати вигляд:

$$T(n) = \log_2(n + 1) - \frac{1}{2}.$$

Отже, складність бінарного алгоритму пошуку має логарифмічний вигляд $O(\log_2 n)$.

Функція алгоритму бінарного пошуку має вигляд:

```
void binnary(int* A, int n)
{int key;
cin >> key;
   bool f = false;
   int l = 0;
   int r = n;
   int mid;
   while ((l <= r) && (f != true)) {
      mid = (l + r) / 2;
      if (A[mid] == key) f = true;
      if (A[mid] > key) r = mid - 1;
      else l = mid + 1;
   }
   if (f)
cout << mid;
   else cout << " there is no element in the array ";}
```

Контрольні питання

1. Що таке пошук?
2. У чому полягає метод лінійного пошуку?
3. Яке розташування елементу в масиві є найкращім випадком для лінійного пошуку?
4. У чому полягає метод бінарного пошуку?
5. Яке розташування елементу в масиві є найкращім випадком для бінарного пошуку?
6. Яку складність має алгоритм бінарного пошуку?

Порядок виконання лабораторної роботи

1. Згенерувати масив випадкових чисел. Розмір масиву вводиться з клавіатури.
2. Розробити програми лінійного та бінарного пошуку.
3. Виконати пошук мінімального, максимального елементів та вивести індекси цих елементів.
4. Виконати пошук елемента, якого немає в масиві.
5. Оцінити складність алгоритмів.

Лабораторна робота №12
Способи задання графів

Мета роботи: вивчення способів представлення заданих графів.

Теоретичні відомості

Графом G називається впорядкована пара (V, E), де V — множина вершин графа, E — мультимножина ребер графу.

Якщо ребро з'єднує дві вершини, то кажуть, що воно інцидентне цим вершинам, а вершини, які з'єднані таким ребром, називаються суміжними. Якщо кінці ребра належать одній вершині, то таке ребро називається петлею. Вершини, які не є кінцями жодного з ребер у графі, називаються ізольованими.

На рисунку 12.1 представлено граф, у якого вершина 1 є ізольованою, а вершини 2 і 3, 3 і 4 є суміжними. Ребро a утворює петлю у вершині 2. Ребро b інцидентне вершинам 2, 3, а ребро c – вершинам 3, 4.

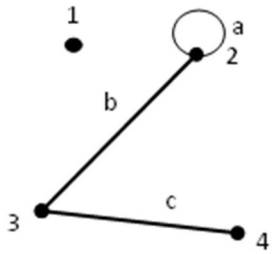


Рис. 12.1.

Степенем (валентністю) вершини v (deg(v) або d(v)), називають кількість ребер, інцидентних цій вершині. Вершина степені 0 (d (v) = 0) є ізольованою. Якщо степінь вершини d (v) = 1, то вершину називають кінцевою, або висячою.

Орієнтований граф (орграф) G(V, E), складається з множини V вершин і множини E впорядкованих пар елементів з V, яка називається множиною орієнтованих ребер або дуг. Якщо (a, b)∈ E, то (a, b) називають орієнтованим ребром (дугою), a – початковою вершиною, b - кінцевою вершиною ребра (a, b).

Степенем виходу вершини v називають кількість ребер, для яких v - початкова вершина (outdeg(v)). Степенем входу вершини v називають кількість

ребер, для яких v - кінцева вершина (indeg(v)). Якщо indeg(v) = 0, то вершину v називають джерелом. Якщо outdeg(v) = 0, то вершину v називають стоком.

Способи представлення графів

1. Матрицею інцидентності неорієнтованого графа G називають p×q матрицю M, рядки якої позначені вершинами, а стовпці - ребрами графа, з елементами

$$m_{ij} = \begin{cases} 1, & \text{якщо вершина і інцидентна ребру } j, \\ 0, & \text{інакше.} \end{cases}$$

Степінь вершини дорівнює сумі одиниць рядка цієї вершини.

Для орієнтованих графів елементи матриці мають вигляд:

$$m_{ij} = \begin{cases} 1, & \text{якщо дуга } e_j \text{ виходить з вершини } v_i \\ -1, & \text{якщо дуга } e_j \text{ входить у вершину } v_i \\ 2, & \text{якщо дуга } e_j \text{ петля у вершині } v_i \\ 0, & \text{в інших випадках} \end{cases}$$

Для матриці інцидентності обсяг пам'яті n(p,q) = O(pq).

2. Матрицею суміжності простого графа називають булеву p × p матрицю M, рядки і стовпці якої позначені вершинами графа в однаковому порядку, а елементи

$$m_{ij} = \begin{cases} 1, & \text{якщо є ребро з } v_i \text{ у } v_j \\ 0, & \text{інакше} \end{cases}$$

Для матриці суміжності n(p,q) = O(p2).

3. Списком суміжності називають структуру, яка відображає суміжність вершин і складається з масиву покажчиків Г. Для цього використовується масив списків (по одному списку на кожну вершину), який для кожної вершини v_i містить у довільному порядку (вказівники на) вершини з множини Г (v_i).

Для неорієнтованих графів $n(p,q) = O(p+2q)$. Для орієнтованих графів складність становить $n(p,q) = O(p+q)$.

4. Списком (масивом) ребер (для орграфів списком (масивом) дуг) називають подання графа за допомогою масиву структур *E*, що відображає список пар суміжних вершин.

Контрольні питання

1. Дайте означення графу.
2. Які вершини називаються суміжними?
3. Яке ребро називається інцидентним вершині?
4. Яку матрицю називають матрицею інцидентності?
5. Яку матрицю називають матрицею суміжності?

Порядок виконання лабораторної роботи

1. Програмно реалізувати граф за допомогою матриць суміжності та інцидентності.
2. Написати програму представлення графу у вигляді списку ребер та суміжності.

Лабораторна робота №13
Обхід графів

Мета роботи: ознайомитись з алгоритмами обходу графів.

Теоретичні відомості

Обхід графа - це перехід від однієї його вершини до іншої у пошуках властивостей зв'язків цих вершин. Зв'язки (лінії, що з'єднують вершини) називаються напрямками, шляхами, гранями чи ребрами графа. Вершини графа також називаються вузлами.

Двома основними алгоритмами обходу графа є пошук в ширину (Breadth-First Search, BFS) та пошук у глибину (Depth-First Search, DFS).

Пошук в ширину. BFS

В алгоритмі пошуку обхід вершин заданого графу здійснюється в порядку їх віддаленості від деякої заздалегідь обраної, або зазначеної як початкова, вершини, яка має назву кореня пошуку в ширину.

Опис алгоритму:

1) Помістити номер вершини, з якого починається пошук, в порожню чергу.

2) Витягти з початку черги вершину U та в окремому масиві помітити її як використану.

3) В кінець черги додати всі вершини, до яких можемо дійти, використовуючи ребро з вершини U, і які ще не позначені.

4) Якщо черга порожня, всі вузли зв'язкового графа було переглянуто, інакше повернутися до п. 2.

Складність алгоритму:

Так як у гіршому випадку алгоритм відвідує всі вузли графа, при зберіганні графа у вигляді списків суміжності, часова складність алгоритму становить O(|V|+|E|), де V – множина вершин графа, а E – множина ребер.

Іншими словами, алгоритм у гіршому випадку працює за кількість ребер плюс кількість вершин.

```
void BFS(int** a,int n)
{
   for (int i = 0; i < n; i++)
      nodes[i] = 0; // на початку всі вершини дорівнюють 0
   Queue.push(0); // розміщуємо в чергу першу вершину
   while (!Queue.empty())
   { int node = Queue.front(); // вилучаємо вершину
      Queue.pop();
      nodes[node] = 2; // помічаємо вершину, як відвідану
      for (int j = 0; j < n; j++)
      { // перевіряємо всі суміжні вершини
         if (a[node][j] == 1 && nodes[j] == 0)
         {Queue.push(j); // додаємо її в чергу
            nodes[j] = 1; // помічаємо вершину як знайдену}
      }
      cout << node + 1 << endl;
   }
}
```

Пошук в глибину. DFS

На кожному кроці пошуку в глибину алгоритм вибирає нову вершину, суміжну з поточною і продовжує пошук із неї. Таке «поглиблення» продовжується доти, доки не буде знайдена вершина, суміжна з кінцевою або алгоритм не зайде в глухий кут.

Нехай треба знайти шлях із вершини S у вершину F. Пошук у глибину (DFS) працює наступним чином:

1. Якщо номери вершин S і F збіглися - шлях знайдений;
2. Якщо з вершини S є ребро у вершину X і вершина X була відвідана раніше – то знайти шлях з X до F рекурсивно.
3. Другий крок виконується для всіх вершин, на які можна перейти з S.
4. Якщо всі дуги з S оброблені і по жодній з них не було знайдено шлях - значить такого шляху немає.

Оскільки обходимо кожного «сусіда» кожного вузла, ігноруючи тих, яких відвідували раніше, маємо час виконання, що дорівнює O(V + E). V+E означає, що для кожної вершини оцінюємо грані, що примикають до неї. Кожна

вершина має певну кількість граней і, у гіршому випадку, обійдемо всі вершини (O(V)) і досліджуємо всі грані (O(E)). Тому отримуємо V+E.

Оскільки використовуємо рекурсію для обходу кожної вершини, це означає, що використовується стек (нескінченна рекурсія призводить до помилки переповнення стека). Тому просторова складність становить O(V).

Рекурсивна функція алгоритму DFS:

```
void DFS (int st, int n)
{int r;
cout<<st+1<<" ";
nodes[st] = 1;// масив пройдених вершин
for (r = 0; r < n; r++)
if ((mas[st][r]!=0)&&(nodes[r]==0))// mas- матриця суміжності
DFS (r,n);}
```

Контрольні питання

1. Що таке обхід графа?
2. Що таке шлях в графі?
3. Яка відмінність алгоритмів BFS та DFS?

Порядок виконання лабораторної роботи

1. Ознайомитись з теоретичною частиною роботи.

2. Реалізувати алгоритм пошуку в ширину.

3. Реалізувати алгоритм пошуку в глибину.

ЛІТЕРАТУРА

1. Алгоритми та методи обчислень [Електронний ресурс] : навч. посіб. – Київ : КПІ ім. Ігоря Сікорського, 2019. – 407 с. https://ela.kpi.ua/bitstream/123456789/27864/1/Alhorytmy_ta_metody_obchislenn.pdf

2. Бородкіна І. Теорія алгоритмів. Посібник для студентів вищих навчальних закладів. – К.: Центр навчальної літератури, 2019.- 184с.

3. Вергунова І.М. Побудова та аналіз алгоритмів. Лекції. – Вінниця : ТВОРИ, 2020. – 164 с.

4. Горлова Т.М. Теорія алгоритмів. [Електронний ресурс]: конспект лекцій для студентів напряму підготовки 6.050101 «Комп'ютерні науки» денної та заочної форм навчання / Т.М. Горлова, К.Є. Бобрівник, Н.В. Ліманська – К.: НУХТ, 2015. – 95 с

5. Креневич А.П. Алгоритми і структури даних. Підручник. – К.: ВПЦ "Київський Університет", 2021. – 200 с.

6. Машина Поста: Методичні вказівки до лабораторної роботи з курсу «Алгоритми і структури даних» для студентів базового напрямку 6.050101 «Комп’ютерні науки» / Укл.: А.Б.Керницький, П.Ю.Денисюк, М.Р.Мельник. - Львів: Національний університет «Львівська політехніка», 2009.- 16 с.

7. Прийма С.М. Теорія алгоритмів: Навчальний посібник. – Мелітополь: ФОП Однорог Т.В., 2018. – 116 с.

Навчальне видання

ТЕОРІЯ АЛГОРИТМІВ

методичні вказівки до лабораторних робіт студентам
факультету математики, фізики та інформаційних технологій
першого (бакалаврського) рівня освіти,
спеціальності 122 «Комп'ютерні науки»

Укладачі:

Ганна Валентинівна Коренкова

Лариса Ярославівна Мартинович

Оксана Миколаївна Зуй

В авторський редакції

Підп. до друку 24.03.2023. Формат 60х84/16
Папір крейд. Ум. друк. арк. 3,95.
Наклад 100 пр. Замовлення 0171.

Видавець і виготовлювач
Видавництво та друкарня «ТЕС»
(Свідоцтво ДК № 771)
м. Одеса, вул. Дальницька, 25/7
Тел.: (0482) 428972, (050) 3953403