Міністерство освіти і науки України

Державний університет «Житомирська політехніка»

Факультет інформаційно-комп’ютерних технологій

Кафедра комп’ютерних наук

**Звіт**

з контрольної роботи №2

з дисципліни «Алгоритми та структури даних»

Виконав студент 1-го курсу, групи ВТ-23-1

спеціальності 121 «Інженерія програмного забеспечення»

Варіант: 4

Нагорний Т. Г.

Керівник Петросян Р. В.

Житомир – 2024

Зміст

[**Швидке сортування** 3](#_Toc166530359)

[**Сортування за розрядами** 3](#_Toc166530360)

[**Хеш-таблиця з відкритою адресацією** 4](#_Toc166530361)

[**Алгоритм Дейкстри** 5](#_Toc166530362)

[**Хеш-таблиця з ланцюжками** 7](#_Toc166530363)

[**Алгоритм Флойда** 8](#_Toc166530364)

[**Алгоритм Шеннон-Фано** 10](#_Toc166530365)

[**Алгоритм Форда-Беллмана** 11](#_Toc166530366)

[**Алгоритм Хаффмана** 12](#_Toc166530367)

[**Алгоритм Крута-Морріса-Прата** 13](#_Toc166530368)

## **Швидке сортування**

Хід роботи

**Опис:** швидке сортування - це рекурсивний алгоритм сортування, який працює на принципі "розділяй і володарюй". Алгоритм обирає один елемент масиву як опорний і розташовує всі елементи менше опорного наліво, а всі елементи більше опорного - справа. Після цього опорний елемент вже знаходиться на своєму правильному місці. Алгоритм рекурсивно застосовується до лівої та правої частини масиву, що знаходяться відносно опорного елемента, доки весь масив не буде відсортований.

**Використані структури даних:** рекурсія, масив.

**Лістинг програми**:

def quicksort(arr):  
 if len(arr) <= 1:  
 return arr  
 pivot = arr[len(arr) // 2]  
 left = [x for x in arr if x < pivot]  
 middle = [x for x in arr if x == pivot]  
 right = [x for x in arr if x > pivot]  
 return quicksort(left) + middle + quicksort(right)  
  
  
test\_data = [5, 3, 8, 6, 2, 7, 1, 4]  
print("Не відсортований масив:", test\_data)  
print("Відсортований масив:", quicksort(test\_data))

**Тестові данні**: [5, 3, 8, 6, 2, 7, 1, 4]



Результат виконання програми

## **Сортування за розрядами**

Хід роботи

**Опис:** сортування за розрядами - це алгоритм сортування, який сортує числа за їхніми розрядами, починаючи з найменш значущого розряду і закінчуючи найбільш значущим. Спочатку числа розподіляються по корзинам на основі значення їхніх розрядів. Потім числа збираються разом у відсортованому порядку. Алгоритм повторює цей процес для кожного розряду, поки весь масив не буде відсортований за всіма розрядами.

**Використані структури даних:** масив.

**Лістинг програми**:

def counting\_sort(arr, exp):  
 n = len(arr)  
 output = [0] \* n  
 count = [0] \* 10  
  
 for i in range(n):  
 index = arr[i] // exp  
 count[index % 10] += 1  
  
 for i in range(1, 10):  
 count[i] += count[i - 1]  
  
 i = n - 1  
 while i >= 0:  
 index = arr[i] // exp  
 output[count[index % 10] - 1] = arr[i]  
 count[index % 10] -= 1  
 i -= 1  
  
 for i in range(n):  
 arr[i] = output[i]  
  
def radix\_sort(arr):  
 max\_num = max(arr)  
 exp = 1  
 while max\_num // exp > 0:  
 counting\_sort(arr, exp)  
 exp \*= 10  
  
test\_data = [170, 45, 75, 90, 802, 24, 2, 66]  
print("Не відсортований масив:", test\_data)  
radix\_sort(test\_data)  
print("Відсортований масив:", test\_data)

**Тестові данні**: [170, 45, 75, 90, 802, 24, 2, 66]



Результат виконання програми

## **Хеш-таблиця з відкритою адресацією**

Хід роботи

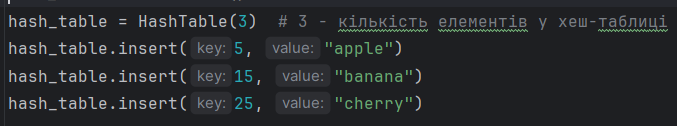
**Опис:** хеш-таблиця з відкритою адресацією - це метод збереження інформації у вигляді пар "ключ-значення", де доступ до значень відбувається через визначений хеш-код ключа. У випадку колізій (коли два ключі відображаються на один і той же індекс хеш-таблиці), використовується відкрита адресація, коли елемент зберігається в наступній вільній комірці. У мові програмування Python тип даних dict реалізований саме за допомогою використання хеш-таблиць.

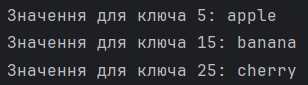
**Використані структури даних:** хеш-таблиця, масив.

**Лістинг програми**:

class HashTable:  
 def \_\_init\_\_(self, size):  
 self.size = size  
 self.hash\_table = [None] \* self.size  
  
 def hash\_function(self, key):  
 return key % self.size  
  
 def insert(self, key, value):  
 index = self.hash\_function(key)  
 while self.hash\_table[index] is not None:  
 index = (index + 1) % self.size  
 self.hash\_table[index] = (key, value)  
  
 def search(self, key):  
 index = self.hash\_function(key)  
 while self.hash\_table[index] is not None and self.hash\_table[index][0] != key:  
 index = (index + 1) % self.size  
 if self.hash\_table[index] is None:  
 return None  
 return self.hash\_table[index][1]  
  
  
hash\_table = HashTable(3) # 3 - кількість елементів у хеш-таблиці  
hash\_table.insert(5, "apple")  
hash\_table.insert(15, "banana")  
hash\_table.insert(25, "cherry")  
  
print("Значення для ключа 5:", hash\_table.search(5))  
print("Значення для ключа 15:", hash\_table.search(15))  
print("Значення для ключа 25:", hash\_table.search(25))

**Тестові данні**:





Результат виконання програми

## **Алгоритм Дейкстри**

Хід роботи

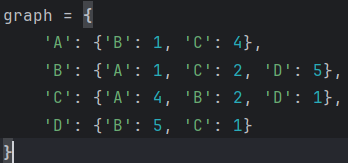
**Опис:** алгоритм Дейкстри - це алгоритм пошуку найкоротших шляхів в графі з невід’ємними вагами ребер від одної вершини до всіх інших вершин. Алгоритм використовується для знаходження найкоротшого шляху від початкової вершини до всіх інших вершин графа. Алгоритм підтримує два списки вершин: список вершин, для яких вже знайдено найкоротший шлях, та список вершин, для яких шлях поки що невідомий. На кожному кроці алгоритм вибирає вершину з другого списку з найменшою вагою шляху, оновлює ваги сусідніх вершин і переносить вибрану вершину до списку вершин з відомими шляхами.

**Використані структури даних:** граф, пріоритетна черга, масив.

**Лістинг програми**:

import heapq  
  
def dijkstra(graph, start):  
 distances = {vertex: float('infinity') for vertex in graph}  
 distances[start] = 0  
 priority\_queue = [(0, start)]  
  
 while priority\_queue:  
 current\_distance, current\_vertex = heapq.heappop(priority\_queue)  
  
 if current\_distance > distances[current\_vertex]:  
 continue  
  
 for neighbor, weight in graph[current\_vertex].items():  
 distance = current\_distance + weight  
 if distance < distances[neighbor]:  
 distances[neighbor] = distance  
 heapq.heappush(priority\_queue, (distance, neighbor))  
  
 return distances  
  
graph = {  
 'A': {'B': 1, 'C': 4},  
 'B': {'A': 1, 'C': 2, 'D': 5},  
 'C': {'A': 4, 'B': 2, 'D': 1},  
 'D': {'B': 5, 'C': 1}  
}  
start\_vertex = 'A'  
print(f"Найкоротші відстані від вершини {start\_vertex} до інших вершин:")  
print(dijkstra(graph, start\_vertex))

**Тестові данні**:





Результат виконання програми

**Пояснення до виконання програми:**

* відстань від вершини А до самої себе 0 (містер очевидність)
* від вершини А до вершини В 1 (пряме ребро між вершинами)
* від вершини А до вершини С 3 (від вершини А до вершини В 1 та від вершини В до вершини С 2, тому і 3, а не 4 як це могло б бути при шляху через одне ребро між А та С)
* від вершини А до вершини D 4 (згадуємо шлях між вершинами А та С + додаємо те, що між вершиною С та D ціна 1, тому ціна шляху 4)

## **Хеш-таблиця з ланцюжками**

Хід роботи

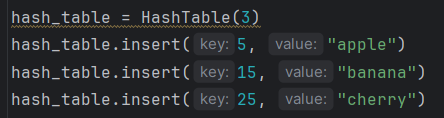
**Опис:** хеш-таблиця з ланцюжками - це метод збереження даних, який використовує хеш-функцію для визначення індексу, де буде збережено значення. Колізії вирішуються шляхом зберігання кожного значення у вигляді ланцюжка (списку) елементів, які мають однаковий хеш. Кожен елемент цього ланцюжка містить ключ та відповідне значення.

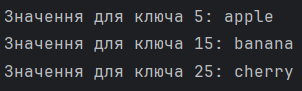
**Використані структури даних:** Хеш-таблиця, масив, список.

**Лістинг програми**:

class HashTable:  
 def \_\_init\_\_(self, size):  
 self.size = size  
 self.hash\_table = [[] for \_ in range(size)]  
  
 def hash\_function(self, key):  
 return key % self.size  
  
 def insert(self, key, value):  
 index = self.hash\_function(key)  
 self.hash\_table[index].append((key, value))  
  
 def search(self, key):  
 index = self.hash\_function(key)  
 for k, v in self.hash\_table[index]:  
 if k == key:  
 return v  
 return None  
  
  
hash\_table = HashTable(3)  
hash\_table.insert(5, "apple")  
hash\_table.insert(15, "banana")  
hash\_table.insert(25, "cherry")  
  
print("Значення для ключа 5:", hash\_table.search(5))  
print("Значення для ключа 15:", hash\_table.search(15))  
print("Значення для ключа 25:", hash\_table.search(25))

**Тестові данні**:





Результат виконання програми

## **Алгоритм Флойда**

Хід роботи

**Опис:** алгоритм Флойда використовується для пошуку найкоротших відстаней між усіма парами вершин у напрямленому або ненапрямленому зваженому графі. Він базується на динамічному програмуванні та використовує матрицю відстаней між вершинами. Алгоритм працює у кілька етапів, на кожному з яких відновлюється матриця найкоротших відстаней, використовуючи проміжні вершини. В кінці алгоритму матриця відстаней містить найкоротші відстані між всіма парами вершин.

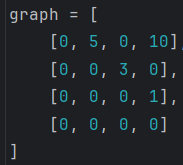
**Використані структури даних:** матриця, граф.

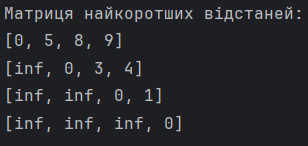
**Лістинг програми**:

def floyd\_warshall(graph):  
 n = len(graph)  
 distance = [[float('inf')] \* n for \_ in range(n)] # float('inf') - нескінченність, нескінченне число  
  
 for i in range(n):  
 for j in range(n):  
 if i == j:  
 distance[i][j] = 0  
 elif graph[i][j] != 0:  
 distance[i][j] = graph[i][j]  
  
 for k in range(n):  
 for i in range(n):  
 for j in range(n):  
 distance[i][j] = min(distance[i][j], distance[i][k] + distance[k][j])  
  
 return distance

graph = [  
 [0, 5, 0, 10],  
 [0, 0, 3, 0],  
 [0, 0, 0, 1],  
 [0, 0, 0, 0]  
]  
print("Матриця найкоротших відстаней:")  
for row in floyd\_warshall(graph):  
 print(row)

**Тестові данні**:





Результат виконання програми

**Пояснення до виконання програми:**

* від вершини 0
  + до вершини 0 – 0 (очевидно)
  + до вершини 1 – 5 (пряме ребро)
  + до вершини 2 – 8 (від вершини 0 до вершини 1 – 5 + від вершини 1 до вершини 2 3, сумарно 8)
  + до вершини 3 – 9 (ребро між 0 та 1 – 5 + ребро між 1 та 2 – 3 + ребро між 3 та 4 – 1, сумарно 9)
* від вершини 1
  + до вершини 0 – відсутній
  + до вершини 1 – 0 (містер очевидність)
  + до вершини 2 – 3 (пряме ребро)
  + до вершини 3 – 4 (пряме ребро)
* від вершини 2
  + до вершини 0 – відсутній
  + до вершини 1 – відсутній
  + до вершини 2 – 0
  + до вершини 3 – 1 (пряме ребро)
* від вершини 3
  + до вершини 0 – відсутній
  + до вершини 1 - відсутній
  + до вершини 2 - відсутній
  + до вершини 3 – 0 (очевидно)

## **Алгоритм Шеннон-Фано**

Хід роботи

**Опис:** алгоритм Шеннона-Фано використовується для стиснення даних. Спочатку він розділяє символи або групи символів на дві частини таким чином, щоб сума ймовірностей символів в кожній з частин була приблизно однаковою. Потім він присвоює бітові коди кожному символу так, щоб коди були унікальними і не містили префіксів один одного. Таким чином, виходить більш ефективне кодування, що дозволяє зменшити кількість бітів, потрібних для представлення даних.

**Використані структури даних:** масив, хеш-таблиця (у вигляді dict python`а).

**Лістинг програми**:

def shannon\_fano(data):  
 if len(data) == 1:  
 return {data[0]: '0'}  
  
 data\_freq = {char: data.count(char) for char in set(data)}  
 sorted\_data\_freq = sorted(data\_freq.items(), key=lambda x: x[1], reverse=True)  
  
 def divide(freq\_list):  
 total\_freq = sum(freq for \_, freq in freq\_list)  
 running\_total = 0  
 split\_index = None  
 for i, (\_, freq) in enumerate(freq\_list):  
 running\_total += freq  
 if running\_total >= total\_freq / 2:  
 split\_index = i  
 break  
 return freq\_list[:split\_index + 1], freq\_list[split\_index + 1:]  
  
 def assign\_codes(freq\_list, code\_prefix):  
 if len(freq\_list) == 1:  
 return {freq\_list[0][0]: code\_prefix}  
 left\_freq, right\_freq = divide(freq\_list)  
 codes = {}  
 codes.update(assign\_codes(left\_freq, code\_prefix + '0'))  
 codes.update(assign\_codes(right\_freq, code\_prefix + '1'))  
 return codes  
  
 codes = assign\_codes(sorted\_data\_freq, '')  
 return codes  
  
  
  
test\_data = "abbcccddddeeeee"  
expected\_codes = {'e': '00', 'd': '01', 'c': '10', 'b': '110', 'a': '111'}  
  
print(shannon\_fano(test\_data), '\n', expected\_codes)

**Тестові данні**:





Результат виконання програми

## **Алгоритм Форда-Беллмана**

Хід роботи

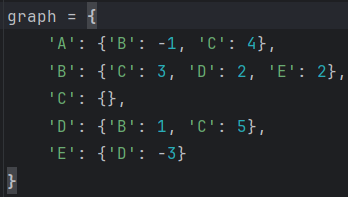
**Опис:** алгоритм Форда-Беллмана використовується для знаходження найкоротших шляхів в графі з вагами на ребрах, навіть якщо він має ребра з від'ємними вагами. Алгоритм працює для напрямлених і ненапрямлених графів.

**Використані структури даних:** граф, масив.

**Лістинг програми**:

def bellman\_ford(graph, start):  
 distance = {node: float('inf') for node in graph}  
 distance[start] = 0  
  
 for \_ in range(len(graph) - 1):  
 for node in graph:  
 for neighbor, weight in graph[node].items():  
 if distance[node] + weight < distance[neighbor]:  
 distance[neighbor] = distance[node] + weight  
  
 for node in graph:  
 for neighbor, weight in graph[node].items():  
 if distance[node] + weight < distance[neighbor]:  
 raise ValueError("Граф містить цикл з від'ємною вагою")  
  
 return distance  
  
graph = {  
 'A': {'B': -1, 'C': 4},  
 'B': {'C': 3, 'D': 2, 'E': 2},  
 'C': {},  
 'D': {'B': 1, 'C': 5},  
 'E': {'D': -3}  
}  
start\_node = 'A'  
  
print(bellman\_ford(graph, start\_node))

**Тестові данні**:





Результат виконання програми

## **Алгоритм Хаффмана**

Хід роботи

**Опис:** алгоритм Хаффмана використовується для стиснення даних, особливо ефективний при стисненні текстових даних. Він будує оптимальне бінарне дерево для кожного символу на основі його частоти входження у текст. Часті символи отримують коротші коди, а рідкі - довші, що дозволяє зменшити загальний обсяг даних.

**Використані структури даних:** масив.

**Лістинг програми**:

import heapq  
from collections import Counter  
  
  
def huffman\_encoding(data):  
 freq = Counter(data)  
 priority\_queue = [[weight, [symbol, ""]] for symbol, weight in freq.items()]  
 heapq.heapify(priority\_queue)  
  
 while len(priority\_queue) > 1:  
 left = heapq.heappop(priority\_queue)  
 right = heapq.heappop(priority\_queue)  
 for pair in left[1:]:  
 pair[1] = '0' + pair[1]  
 for pair in right[1:]:  
 pair[1] = '1' + pair[1]  
 heapq.heappush(priority\_queue, [left[0] + right[0]] + left[1:] + right[1:])

huffman\_code = {}  
 for pair in priority\_queue[0][1:]:  
 huffman\_code[pair[0]] = pair[1]  
  
 encoded\_data = ''.join(huffman\_code[char] for char in data)  
  
 return encoded\_data, huffman\_code  
  
  
def huffman\_decoding(encoded\_data, huffman\_code):  
 reverse\_code = {code: char for char, code in huffman\_code.items()}  
 decoded\_data = ''  
 current\_code = ''  
 for bit in encoded\_data:  
 current\_code += bit  
 if current\_code in reverse\_code:  
 decoded\_data += reverse\_code[current\_code]  
 current\_code = ''  
 return decoded\_data  
  
  
  
data = "hello, world!"  
  
encoded\_data, huffman\_code = huffman\_encoding(data)  
decoded\_data = huffman\_decoding(encoded\_data, huffman\_code)  
  
print("Закодовані дані:", encoded\_data)  
print("Декодовані дані:", decoded\_data)

**Тестові данні**: hello, world!



Результат виконання програми

## **Алгоритм Крута-Морріса-Прата**

Хід роботи

**Опис:** алгоритм Крута-Морріса-Прата використовується для ефективного пошуку всіх входжень підстроки у великому тексті. Він працює на основі зіставлення символів і може знаходити всі входження шуканої підстроки без зайвих порівнянь.

**Використані структури даних:** масив.

**Лістинг програми**:

def compute\_lps\_array(pattern):  
 length = len(pattern)  
 lps = [0] \* length  
 j = 0  
  
 for i in range(1, length):  
 while j > 0 and pattern[i] != pattern[j]:  
 j = lps[j-1]  
 if pattern[i] == pattern[j]:  
 j += 1  
 lps[i] = j  
 return lps  
  
def kmp\_search(text, pattern):  
 count = 0  
 lps = compute\_lps\_array(pattern)  
 i, j = 0, 0  
  
 while i < len(text):  
 if text[i] == pattern[j]:  
 i += 1  
 j += 1  
 if j == len(pattern):  
 count += 1  
 j = lps[j-1]  
 elif i < len(text) and text[i] != pattern[j]:  
 if j != 0:  
 j = lps[j-1]  
 else:  
 i += 1  
 return count  
  
text = "123 12 123 111"  
pattern = "123"  
  
matches\_count = kmp\_search(text, pattern)  
print("Кількість входжень підстроки:", matches\_count)

**Тестові данні**:





Результат виконання програми