

Міністерство освіти і науки України
Національний технічний університет України «Київський політехнічний інститут
імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра інформатики та програмної інженерії

Звіт

з лабораторної роботи №7 з дисципліни
«Алгоритми та структури даних 2. Структури даних»

«Проектування і аналіз алгоритмів
пошуку»

Варіант 3

Виконав студент ІП-15, Борисик Владислав Тарасович
(шифр, прізвище, ім'я, по батькові)

Перевірів Соколовський Владислав Володимирович
(прізвище, ім'я, по батькові)

Мета лабораторної роботи

Мета роботи – вивчити основні підходи аналізу обчислювальної складності алгоритмів пошуку оцінити їх ефективність на різних структурах даних.

Завдання

Згідно варіанту , написати алгоритм пошуку за допомогою псевдокоду (чи іншого способу за вибором).

Провести аналіз часової складності пошуку в гіршому, кращому і середньому випадках і записати часову складність в асимптотичних оцінках.

Виконати програмну реалізацію алгоритму на будь-якій мові програмування для пошуку індексу елемента по заданому ключу в масиві і двохзв'язному списку з фіксацією часових характеристик оцінювання (кількість порівнянь).

Для варіантів з Хеш-функцією замість масиву і двохзв'язного списку. використати безіндексну структуру даних розмірності n , що містить пару ключ-значення рядкового типу. Ключ – унікальне рядкове поле до 20 символів, значення – рядкове поле до 200 символів. Виконати пошук значення по заданому ключу. Розмірність хеш-таблиці регулювати відповідно потребам, а початкову її розмірність обрати самостійно.

Провести ряд випробувань алгоритму на структурах різної розмірності (100, 1000, 5000, 10000, 20000 елементів) і побудувати графіки залежності часових характеристик оцінювання від розмірності структури.

Для проведення випробувань у варіантах з хешуванням рекомендується розробити генератор псевдовипадкових значень полів структури заданої розмірності.

3	Пошук Фібоначчі
---	-----------------

Псевдокод алгоритму

```
def fibonacci_search(input_array: list[int], num_to_find: int):
    array_size := len(input_array)

    # ініціалізуємо числа Фібоначчі
    fib1 := 0
    fib2 := 1
    fib3 := 1

    # якщо число Фібоначчі менше за розмірність масиву
    Поки fib3 < array_size:
        # йдемо далі по числам Фібоначчі
        fib1 := fib2
        fib2 := fib3
        fib3 := fib2 + fib1

    # Зміщення
    offset := -1

    Поки fib3 > 1:
        # індекс (найменше значення між offset + fib1 і array_size - 1)
        index = min(offset + fib1, array_size - 1)

        # якщо число в масиві за індексом index менше за шукане число
        Якщо input_array[index] < num_to_find:
            offset := index
            # змінюємо числа Фібоначчі
            fib3 := fib2
            fib2 := fib1
            fib1 := fib3 - fib2
        # якщо число в масиві за індексом index більше за шукане число
        Інакше якщо input_array[index] > num_to_find:
            fib3 := fib1
            fib2 := fib2 - fib1
            fib1 := fib3 - fib2
        # якщо число в масиві за індексом index дорівнює шуканому числу
        Інакше:
            Повернути index

    Якщо fib2 і input_array[array_size - 1] == num_to_find:
        повернути array_size - 1,

    # якщо шукане число в масиві не знайшли, то повертаємо -1
    повернути -1,
```

Аналіз часової складності

Найкраща швидкодія: $O(1)$

Середня швидкодія: $O(\log n)$

Найгірша швидкодія: $O(\log n)$

Програмна реалізація алгоритму на мові Python

```
def fibonacci_search(input_array: list[int], num_to_find: int):
    array_size = len(input_array)

    # ініціалізуємо числа Фібоначчі
    fib1 = 0
    fib2 = 1
    fib3 = 1

    # якщо число Фібоначчі менше за розмірність масиву
    while fib3 < array_size:
        # йдемо далі по числам Фібоначчі
        fib1 = fib2
        fib2 = fib3
        fib3 = fib2 + fib1

    # Зміщення
    offset = -1

    while fib3 > 1:
        # індекс (найменше значення між offset + fib1 і array_size - 1)
        index = min(offset + fib1, array_size - 1)

        # якщо число в масиві за індексом index менше за шукане число
        if input_array[index] < num_to_find:
            offset = index
            # змінюємо числа Фібоначчі
            fib3 = fib2
            fib2 = fib1
            fib1 = fib3 - fib2

        # якщо число в масиві за індексом index більше за шукане число
        elif input_array[index] > num_to_find:
            fib3 = fib1
            fib2 = fib2 - fib1
            fib1 = fib3 - fib2

        # якщо число в масиві за індексом index дорівнює шуканому числу
        else:
            return index

    if fib2 and input_array[array_size - 1] == num_to_find:
        return array_size - 1,

    # якщо шукане число в масиві не знайшли, то повертаємо -1
    return -1,

def fibonacci_search_DoublyLinkedList(linked_list: DoublyLinkedList, num_to_find: int,
list_size: int):
    # ініціалізуємо числа Фібоначчі
    fib1 = 0
    fib2 = 1
    fib3 = 1

    # якщо число Фібоначчі менше за розмірність масиву
```

```
while fib3 < list_size:
    # йдемо далі по числам Фібоначчі
    fib1 = fib2
    fib2 = fib3
    fib3 = fib2 + fib1

# Зміщення
offset = -1

while fib3 > 1:
    # індекс (найменше значення між offset + fib1 і array_size - 1)
    index = min(offset + fib1, list_size - 1)

    # якщо число в масиві за індексом index менше за шукане число
    if linked_list.get_node(index).data < num_to_find:
        offset = index
        # змінюємо числа Фібоначчі
        fib3 = fib2
        fib2 = fib1
        fib1 = fib3 - fib2
    # якщо число в масиві за індексом index більше за шукане число
    elif linked_list.get_node(index).data > num_to_find:
        fib3 = fib1
        fib2 = fib2 - fib1
        fib1 = fib3 - fib2
    # якщо число в масиві за індексом index дорівнює шуканому числу
    else:
        return index

if fib2 and linked_list.get_node(list_size - 1).data == num_to_find:
    return list_size - 1

# якщо шукане число в масиві не знайшли, то повертаємо -1
return -1
```

Вихідний код

functions.py:

```
from classes import DoublyLinkedList

def fibonacci_search(input_array: list[int], num_to_find: int):
    array_size = len(input_array)

    # ініціалізуємо числа Фібоначчі
    fib1 = 0
    fib2 = 1
    fib3 = 1

    # якщо число Фібоначчі менше за розмірність масиву
    while fib3 < array_size:
        # йдемо далі по числам Фібоначчі
        fib1 = fib2
        fib2 = fib3
        fib3 = fib2 + fib1

    # Зміщення
    offset = -1

    while fib3 > 1:
        # індекс (найменше значення між offset + fib1 і array_size - 1)
        index = min(offset + fib1, array_size - 1)

        # якщо число в масиві за індексом index менше за шукане число
        if input_array[index] < num_to_find:
            offset = index
            # змінюємо числа Фібоначчі
            fib3 = fib2
            fib2 = fib1
            fib1 = fib3 - fib2
        # якщо число в масиві за індексом index більше за шукане число
        elif input_array[index] > num_to_find:
            fib3 = fib1
            fib2 = fib2 - fib1
            fib1 = fib3 - fib2
        # якщо число в масиві за індексом index дорівнює шуканому числу
        else:
            return index

    if fib2 and input_array[array_size - 1] == num_to_find:
        return array_size - 1,

    # якщо шукане число в масиві не знайшли, то повертаємо -1
    return -1,

def fibonacci_search_DoublyLinkedList(linked_list: DoublyLinkedList, num_to_find: int,
list_size: int):
    # ініціалізуємо числа Фібоначчі
    fib1 = 0
```



```

fib2 = 1
fib3 = 1

# якщо число Фібоначчі менше за розмірність масиву
while fib3 < list_size:
    # йдемо далі по числам Фібоначчі
    fib1 = fib2
    fib2 = fib3
    fib3 = fib2 + fib1

# Зміщення
offset = -1

while fib3 > 1:
    # індекс (найменше значення між offset + fib1 і array_size - 1)
    index = min(offset + fib1, list_size - 1)

    # якщо число в масиві за індексом index менше за шукане число
    if linked_list.get_node(index).data < num_to_find:
        offset = index
        # змінюємо числа Фібоначчі
        fib3 = fib2
        fib2 = fib1
        fib1 = fib3 - fib2
    # якщо число в масиві за індексом index більше за шукане число
    elif linked_list.get_node(index).data > num_to_find:
        fib3 = fib1
        fib2 = fib2 - fib1
        fib1 = fib3 - fib2
    # якщо число в масиві за індексом index дорівнює шуканому числу
    else:
        return index

if fib2 and linked_list.get_node(list_size - 1).data == num_to_find:
    return list_size - 1

# якщо шукане число в масиві не знайшли, то повертаємо -1
return -1

```

classes.py:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next_node = None
        self.previous_node = None

class DoublyLinkedList:
    def __init__(self):
        self.head_node = None

    # Додаємо елемент в кінець двохзв'язного списку
    def push(self, new_value):
        # ініціалізуємо новий вузол
        new_node = Node(new_value)
        # ініціалізуємо наступний вузол
        new_node.next_node = self.head_node

        # якщо перший вузол не порожній
        if self.head_node is not None:
            # ініціалізуємо попередній вузол
            self.head_node.previous_node = new_node

        # ініціалізуємо новий вузол
        self.head_node = new_node

    # Print the Doubly Linked list
    def print(self, node):
        while node is not None:
            print(node.data),
            node = node.next_node

    def get_node(self, index):
        temp_node = self.head_node

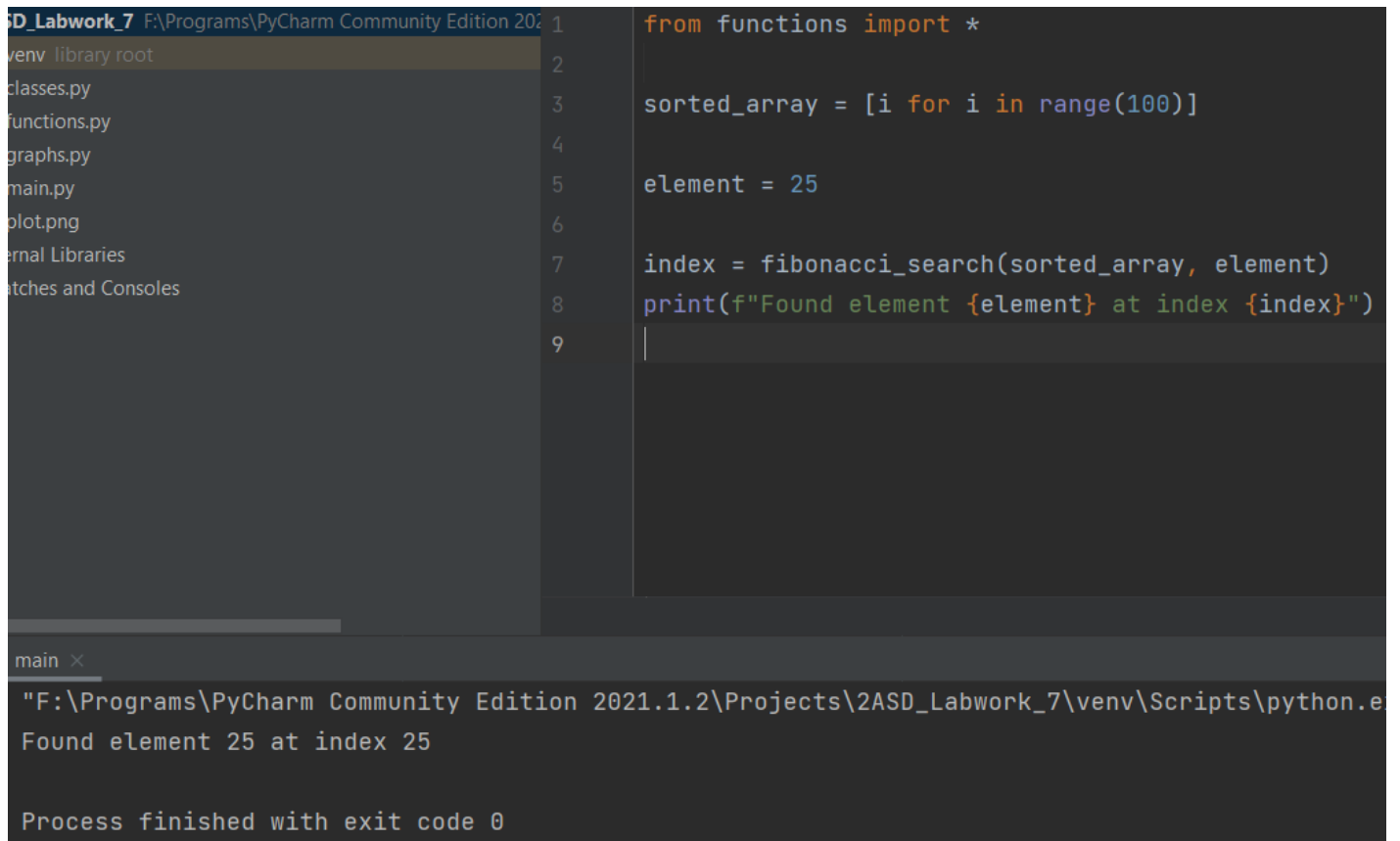
        for i in range(index):
            temp_node = temp_node.next_node

        return temp_node
```

Приклади роботи

Пошук елемента в масиві на 100 елементів:

Для цього прикладу згенеруємо відсортований масив і знайдемо в ньому індекс елемента 25.



The screenshot shows the PyCharm IDE interface. On the left is a file explorer with a project named '2ASD_Labwork_7'. The main editor window displays a Python script with the following code:

```
1 from functions import *
2
3 sorted_array = [i for i in range(100)]
4
5 element = 25
6
7 index = fibonacci_search(sorted_array, element)
8 print(f"Found element {element} at index {index}")
9
```

Below the editor, the 'Run' tab is active, showing the output of the script:

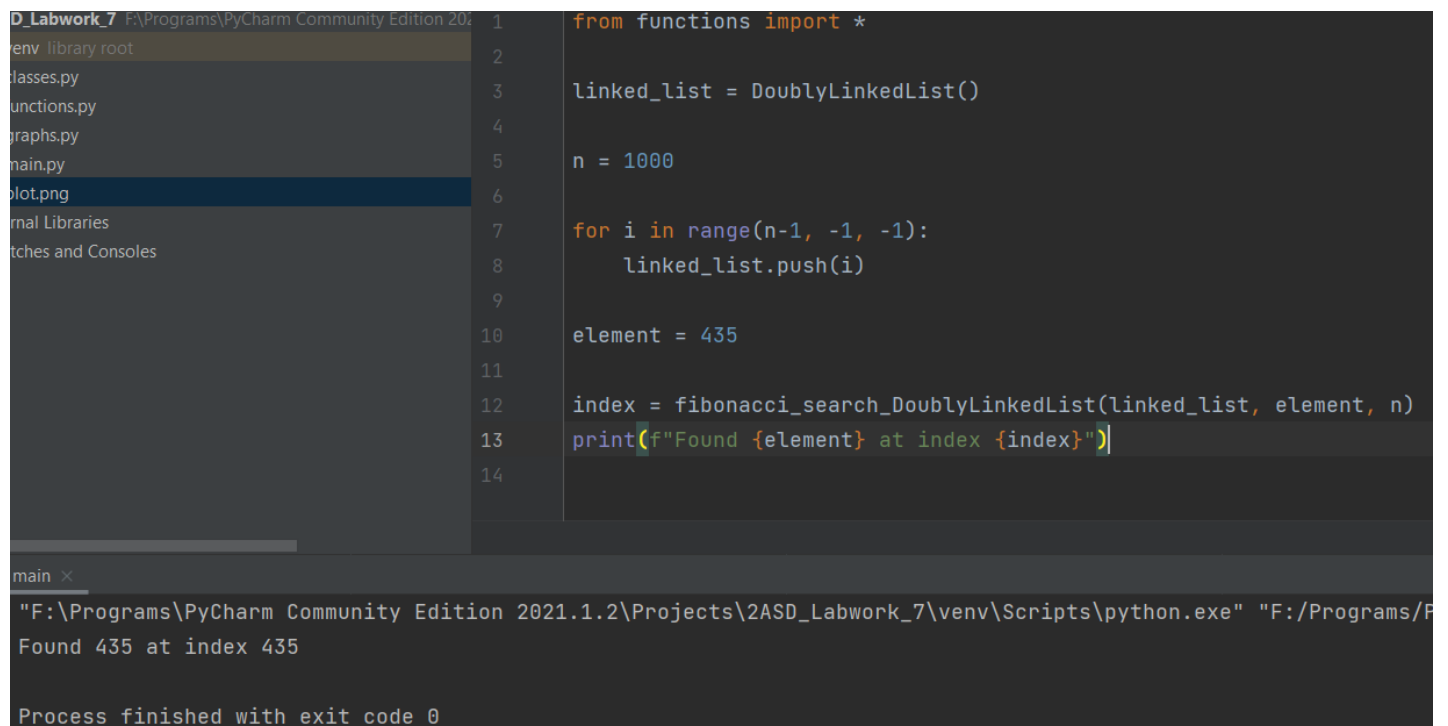
```
"F:\Programs\PyCharm Community Edition 2021.1.2\Projects\2ASD_Labwork_7\venv\Scripts\python.exe
Found element 25 at index 25

Process finished with exit code 0
```

Елемент було знайдено правильно.

Пошук елемента в двохзв'язному списку на 1000 елементів:

Для цього прикладу згенеруємо відсортований двохзв'язний список і знайдемо в ньому індекс елемента 435.



The image shows a PyCharm IDE window with a project named 'D_Labwork_7'. The left sidebar displays a file explorer with the following files: 'env', 'library root', 'classes.py', 'functions.py', 'graphs.py', 'main.py', 'plot.png', 'External Libraries', and 'atches and Consoles'. The 'main.py' file is selected and open in the editor. The code in 'main.py' is as follows:

```
1 from functions import *
2
3 linked_list = DoublyLinkedList()
4
5 n = 1000
6
7 for i in range(n-1, -1, -1):
8     linked_list.push(i)
9
10 element = 435
11
12 index = fibonacci_search_DoublyLinkedList(linked_list, element, n)
13 print(f"Found {element} at index {index}")
14
```

Below the editor, the 'Run' console is visible, showing the execution output:

```
main x
"F:\Programs\PyCharm Community Edition 2021.1.2\Projects\2ASD_Labwork_7\venv\Scripts\python.exe" "F:/Programs/P
Found 435 at index 435

Process finished with exit code 0
```

Елемент було знайдено правильно.

Часові характеристики оцінювання

В таблицях наведені характеристики оцінювання числа порівнянь при пошуку елемента і числа звертань при «Пошуку Фібоначчі» для масивів різної розмірності і двохзв'язних списків різної розмірності.

Таблиця 1 – Число порівнянь в масиві при пошуку різних елементів.

Розмірність масиву	Число порівнянь в масиві (шукаємо перший елемент)	Число порівнянь в масиві (шукаємо середній елемент)	Число порівнянь в масиві (шукаємо останній елемент)
100	19	25	15
1000	30	34	20
5000	35	43	23
10000	38	46	30
20000	42	44	26

Таблиця 2 – Число порівнянь в двохзв'язному списку при пошуку різних елементів.

Розмірність двохзв'язного списку	Число порівнянь в списку (шукаємо перший елемент)	Число порівнянь в списку (шукаємо середній елемент)	Число порівнянь в списку (шукаємо останній елемент)
100	19	25	15
1000	30	34	20
5000	35	43	23
10000	38	46	30
20000	42	44	26

Можемо зробити висновок, що кількість порівнянь в масиві і двохзв'язному списку при пошуку різних елементів однакові.

Таблиця 3 – Число звертань до елементів масиву при пошуку різних елементів.

Розмірність масиву	Число звертань до елементів масиву (шукаємо перший елемент)	Число звертань до елементів масиву (шукаємо середній елемент)	Число звертань до елементів масиву (шукаємо останній елемент)
100	4	7	2
1000	7	9	2
5000	8	12	2
10000	9	13	5
20000	10	11	2

Таблиця 4 – Число звертань до елементів двохзв'язного списку при пошуку різних елементів.

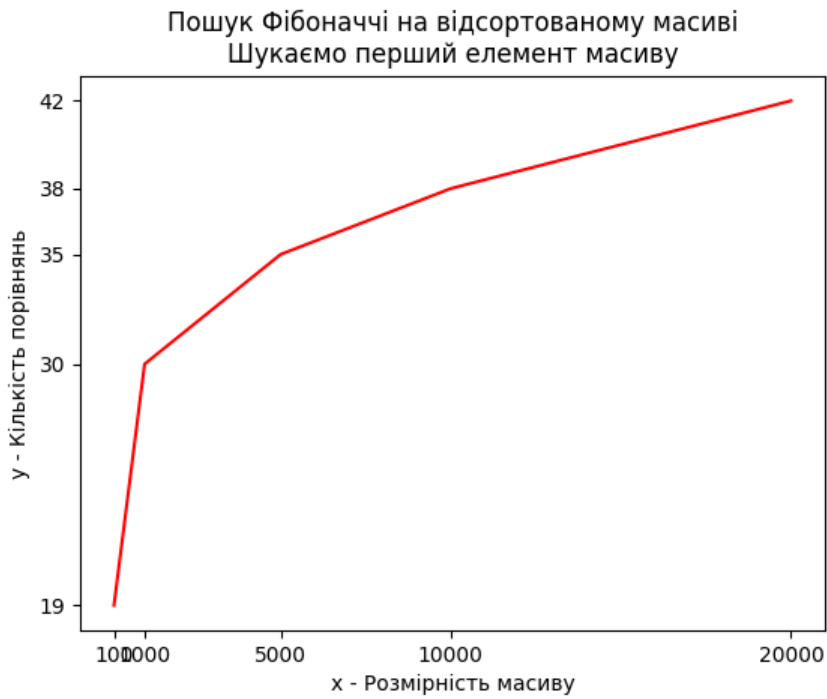
Розмірність двохзв'язного списку	Число звертань до елементів списку (шукаємо перший елемент)	Число звертань до елементів списку (шукаємо середній елемент)	Число звертань до елементів списку (шукаємо останній елемент)
100	4	7	2
1000	7	9	2
5000	8	12	2
10000	9	13	5
20000	10	11	2

Можемо зробити висновок, що кількість звертань до елементів в масиві і двохзв'язному списку при пошуку різних елементів однакові.

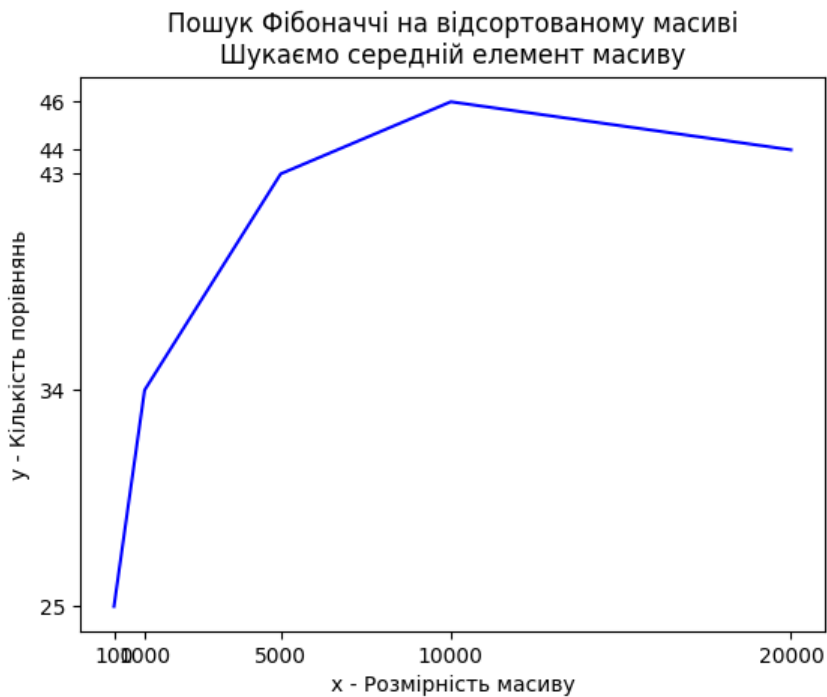
Графіки залежності часових характеристик оцінювання від розмірності структури

1) Графіки кількості порівнянь в масиві при пошуку різних елементів:

Шукаємо перший елемент:

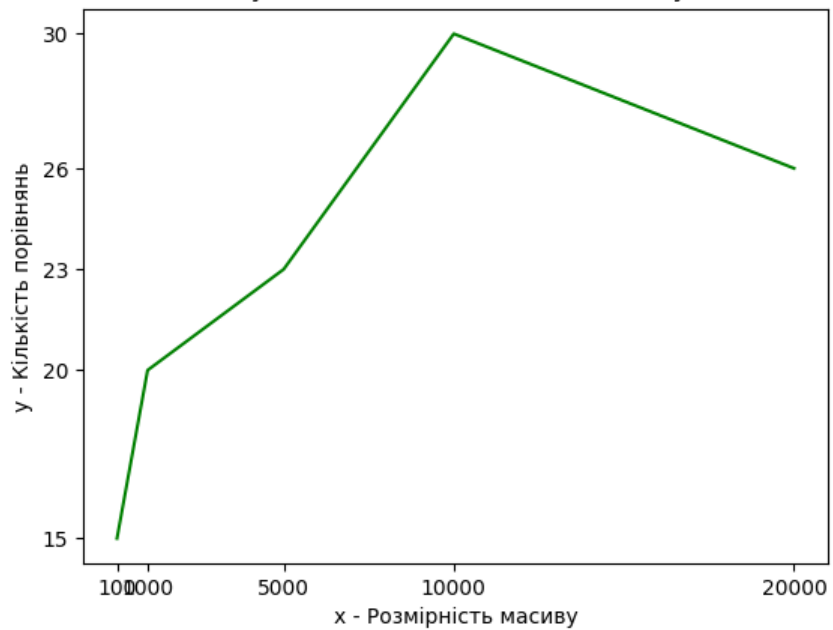


Шукаємо середній елемент:



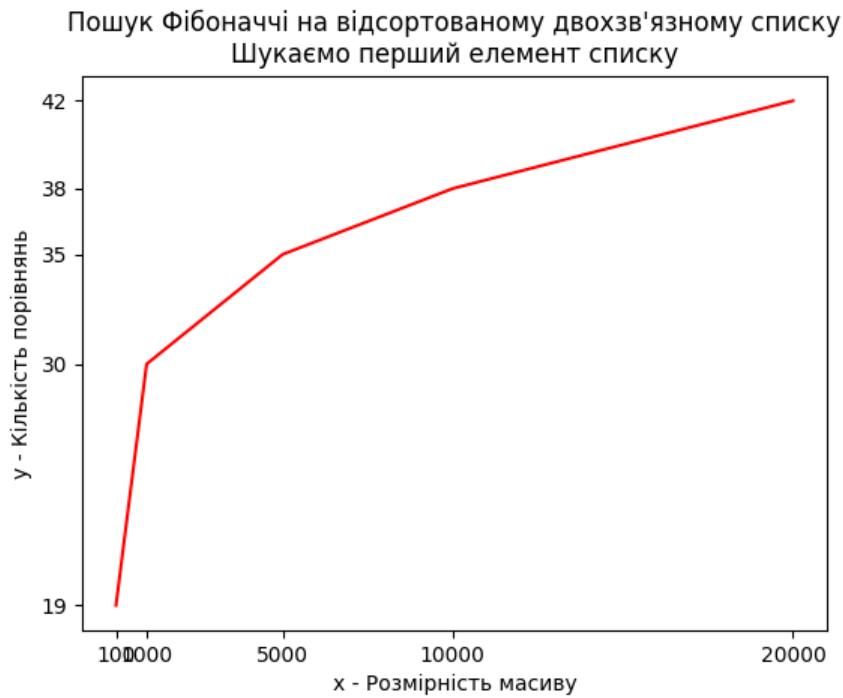
Шукаємо останній елемент:

Пошук Фібоначчі на відсортованому масиві
Шукаємо останній елемент масиву

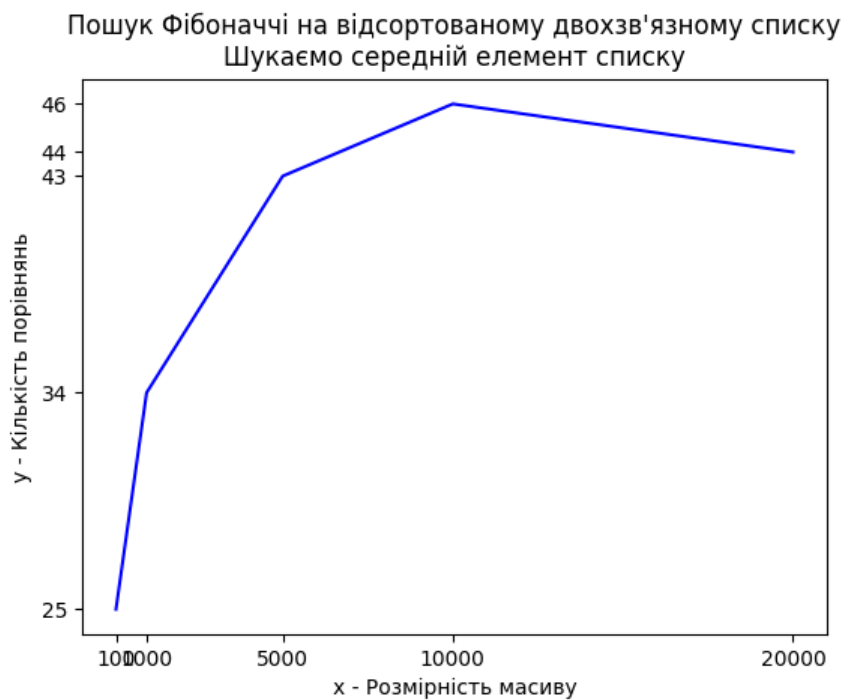


2) Графіки кількості порівнянь в двохзв'язному списку при пошуку різних елементів:

Шукаємо перший елемент:

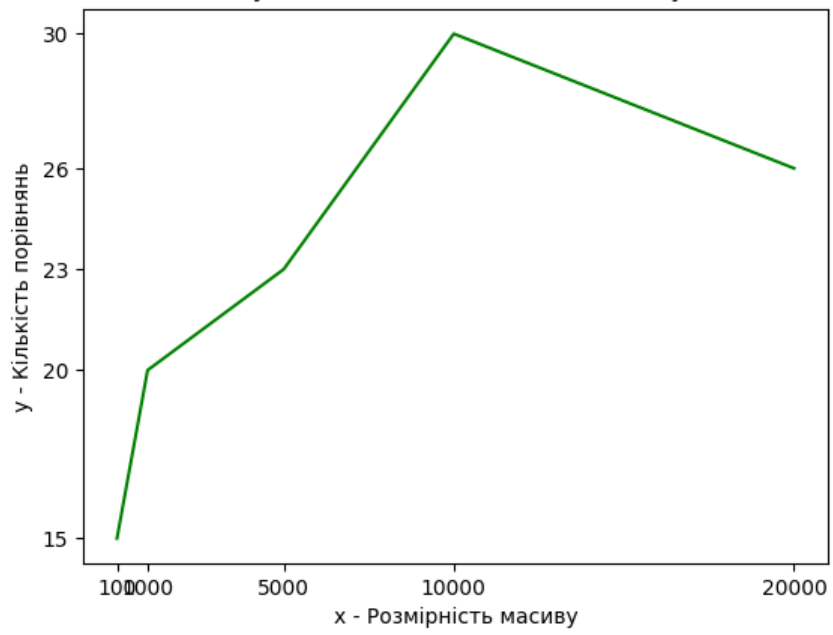


Шукаємо середній елемент:



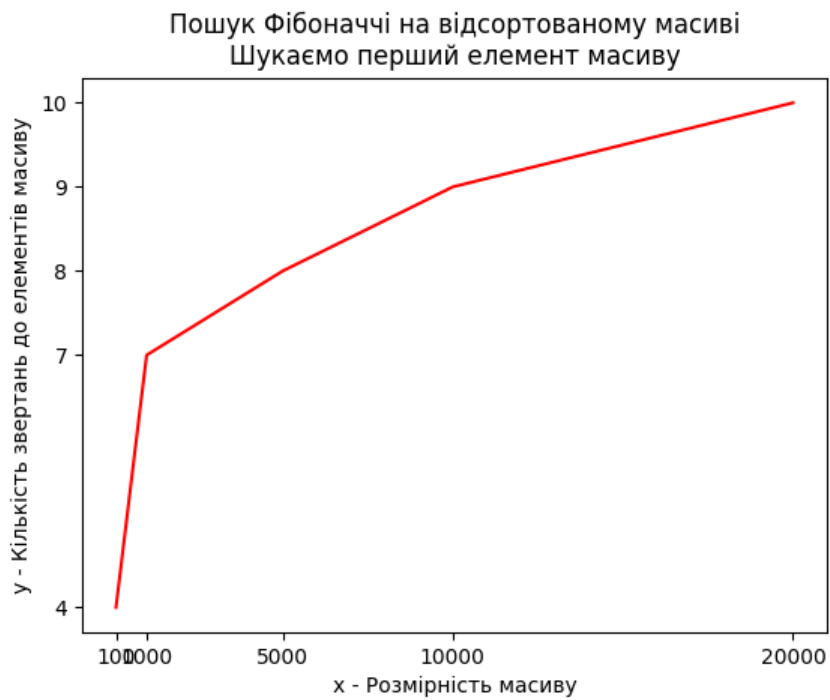
Шукаємо останній елемент:

Пошук Фібоначчі на відсортованому двохзв'язному списку
Шукаємо останній елемент списку

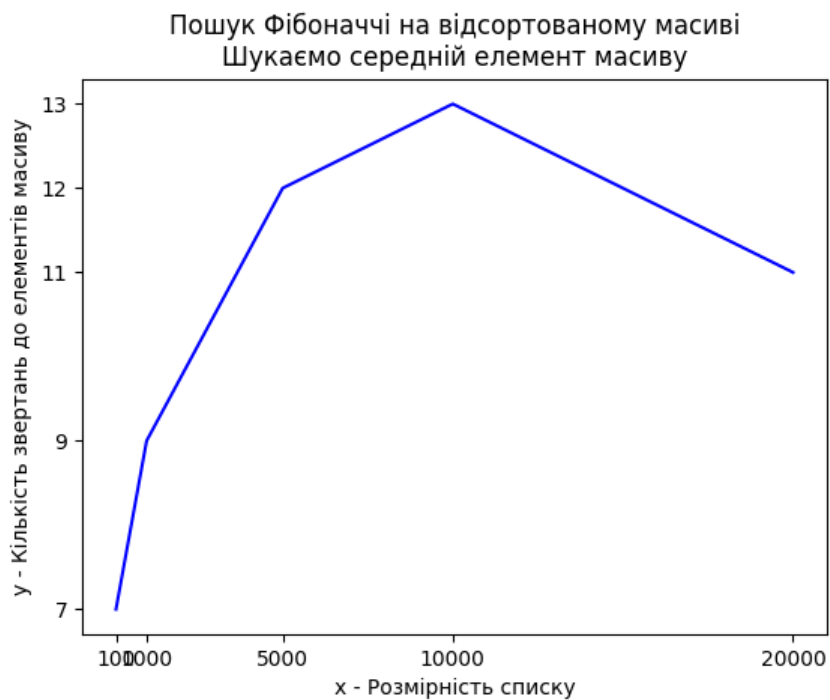


3) Графіки кількості звертань до елементів в масиві при пошуку різних елементів:

Шукаємо перший елемент:

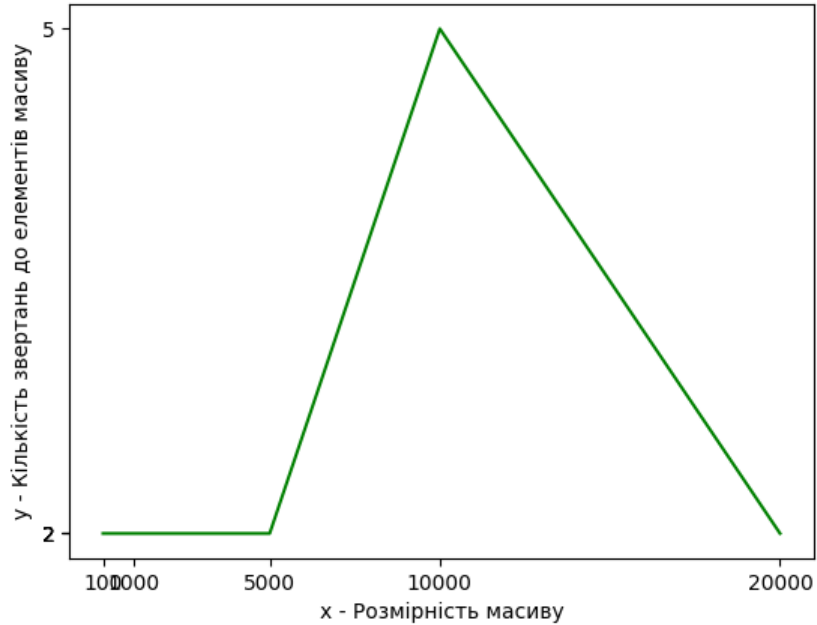


Шукаємо середній елемент:



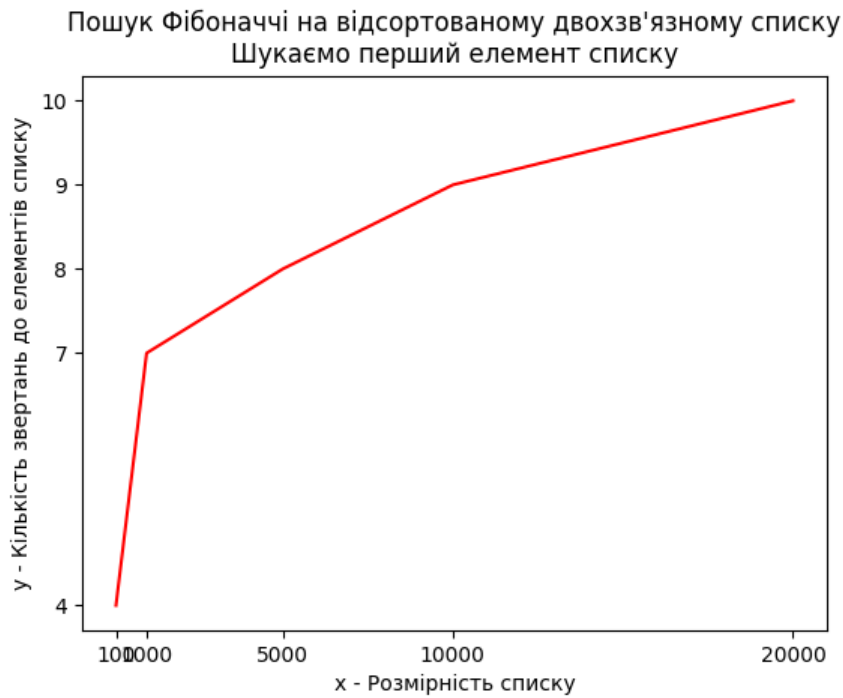
Шукаємо останній елемент:

Пошук Фібоначчі на відсортованому масиві
Шукаємо останній елемент масиву

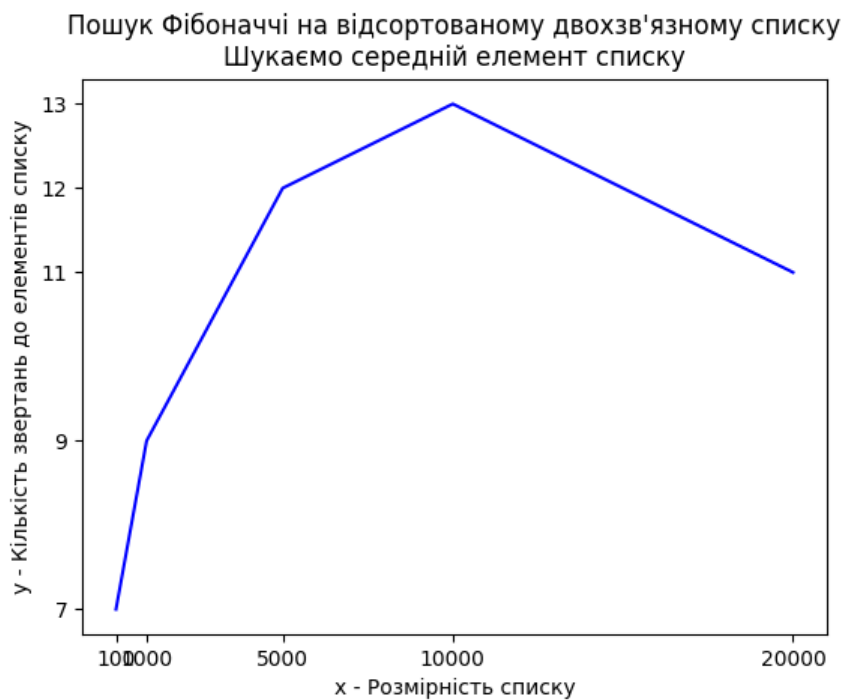


4) Графіки кількості звертань до елементів в двохзв'язному списку при пошуку різних елементів:

Шукаємо перший елемент:

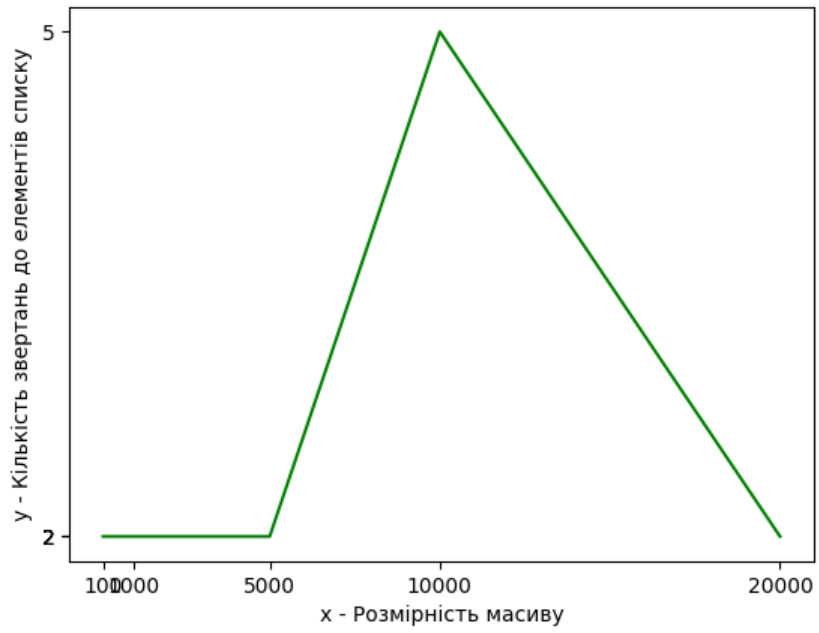


Шукаємо середній елемент:



Шукаємо останній елемент:

Пошук Фібоначчі на відсортованому двохзв'язному списку
Шукаємо останній елемент списку



Висновок

В рамках виконання даної лабораторної роботи я вивчив основні підходи аналізу обчислювальної складності алгоритму пошуку "Пошук Фібоначчі", оцінив його ефективність на різних структурах даних: під час порівняння ефективності даного алгоритму можна зробити висновок, що кількість порівнянь і кількість звертань до елементів в масиві і двохзв'язному списку при пошуку однакових елементів співпадають. Також навів графіки залежності часових характеристик оцінювання від розмірності структури.