**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»

**«Проєктування і аналіз алгоритмів пошуку»**

**Виконав** ІП-11Трикош Іван Володимирович

**Перевірила** Халус Олена Андріївна

Київ 2022

1 МЕТА ЛАБОРАТОРНОЇ РОБОТИ

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

2 ЗАВДАННЯ

Згідно варіанту (таблиця 2.1), 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.

Зробити висновок з лабораторної роботи.

Таблиця 2.1 – Варіанти алгоритмів

|  |  |
| --- | --- |
| **№** | **Алгоритм пошуку** |
| 31 | Пошук Фібоначчі |

3 ВИКОНАННЯ

3.1 Псевдокод алгоритму

function fibonacci\_search1(double\_linked\_tree – двозв'язний список, value – шукане значення):

fib\_prev2 = 0

fib\_prev1 = 1

fib\_current = fib\_prev2 + fib\_prev1

n = double\_linked\_tree.length

while (fib\_current < n):

fib\_prev2 = fib\_prev1

fib\_prev1 = fib\_current

fib\_current = fib\_prev2 + fib\_prev1

end while

offset = -1

while (fib\_current > 1):

i = min(offset+fib\_prev2, n-1)

temp = double\_linked\_tree.head

for j = 0 to i do:

temp = temp.next\_item

end for

if (temp.value < value) then:

fib\_current = fib\_prev1

fib\_prev1 = fib\_prev2

fib\_prev2 = fib\_current - fib\_prev1

offset = i

else if (temp.value > value) then:

fib\_current = fib\_prev2

fib\_prev1 = fib\_prev1 - fib\_prev2

fib\_prev2 = fib\_current - fib\_prev1

else:

return i

end if

if (fib\_prev1 && double\_linked\_tree.tail.value == value) then:

return n-1

end if

end while

return -1

end function fibonacci\_search1

function fibonacci\_search2(arr – масив, value – шукане значення):

fib\_prev2 = 0

fib\_prev1 = 1

fib\_current = fib\_prev2 + fib\_prev1

n = arr.length

while (fib\_current < n):

fib\_prev2 = fib\_prev1

fib\_prev1 = fib\_current

fib\_current = fib\_prev2 + fib\_prev1

end while

offset = -1

while (fib\_current > 1):

i = min(offset+fib\_prev2, n-1)

if (arr[i] < value) then:

fib\_current = fib\_prev1

fib\_prev1 = fib\_prev2

fib\_prev2 = fib\_current - fib\_prev1

offset = i

else (if arr[i] > value) then:

fib\_current = fib\_prev2

fib\_prev1 = fib\_prev1 - fib\_prev2

fib\_prev2 = fib\_current - fib\_prev1

else:

return i

end if

end while

if (fib\_prev1 && arr[n-1] == value) then:

return n-1

end if

return -1

end function fibonacci\_search2

3.2 Аналіз часової складності

На рисунку 3.1 зображено графік залежності кількості ітерацій від розміру масиву/списку.

Рисунок 3.1 – Графік залежності кількості ітерацій від розміру масиву

По графіку бачимо, що часова складність дорівнює О(log n).

3.3 Програмна реалізація алгоритму

3.3.1 Вихідний код

from random import randint

class List\_node:

'''Цей клас є представленням елемента двозв'язного списку'''

# Конструктор для ініціалізації значення елемента та покажчика на попередній та наступний елемент

def \_\_init\_\_(self, value):

self.value = value

self.next = None

self.previous = None

return

# Метод для визначення, чи дорівнює елемент списку переданому значенню

def is\_value(self, value):

if value == self.value:

return True

else:

return False

class Double\_List:

'''Цей клас є представленням двозв'язного списку'''

# Констуктор для ініціалізації початку та кінця двозв'язного списку

def \_\_init\_\_(self):

self.head = None

self.tail = None

return

# Метод для визначення довжини списку

def length\_of\_list(self):

count = 0

this\_node = self.head

while this\_node is not None:

count = count + 1

this\_node = this\_node.next

return count

# Метод для виводу двозв'язного списку

def output\_list(self):

this\_node = self.head

while this\_node is not None:

print(this\_node.value, end = ", ")

this\_node = this\_node.next

return

# Пошук Фібоначчі для двозв'язного списку

def search\_of\_fibonacci(self, value):

fib\_prev2 = 0 # Позапопереднє число

fib\_prev1 = 1 # Попереднє число

fib\_current = fib\_prev2 + fib\_prev1 # Поточне число

n = self.length\_of\_list() # Довжина списку

counter1 = 0 # Кількість порівнянь

counter2 = 0 # Кількість звертань

counter3 = 0 # Кількість ітерацій

# Знаходимо мінімальне число Фібоначчі, яке більше за розмір списку

while fib\_current < n:

counter1 += 1

counter3 += 1

fib\_prev2 = fib\_prev1

fib\_prev1 = fib\_current

fib\_current = fib\_prev2 + fib\_prev1

counter1 += 1

offset = -1 # Для позначення виключених спереду значень

while fib\_current > 1:

counter1 += 1

counter3 += 1

i = min(offset+fib\_prev2, n-1)

# Знаходимо елемент з індексом і

temp = self.head

counter2 += 1

for j in range(i):

counter2 += 1

temp = temp.next

counter1 += 1

counter2 += 1

# Якщо елемент менший за потрібний, то беремо ліву частину

if temp.value < value:

fib\_current = fib\_prev1

fib\_prev1 = fib\_prev2

fib\_prev2 = fib\_current - fib\_prev1

offset = i

# Якщо елемент більший за потрібний, то беремо праву частину

elif temp.value > value:

counter1 += 1

counter2 += 1

fib\_current = fib\_prev2

fib\_prev1 = fib\_prev1 - fib\_prev2

fib\_prev2 = fib\_current - fib\_prev1

# Інакше елемент знайдено

else:

counter1 += 1

counter2 += 1

return i, counter1, counter2, counter3

counter1 += 1

counter2 += 1

# Перевірка останнього елемента

if fib\_prev1 and self.tail.value == value:

return n-1, counter1, counter2, counter3

return -1, counter1, counter2, counter3 # Елемент не знайдено

# Метод для додавання елемента в кінець списку

def add\_node(self, node):

if self.head is None:

self.head = node

node.previous = None

node.next = None

self.tail = node

else:

self.tail.next = node

node.previous = self.tail

self.tail = node

return

# Пошук Фібоначчі для масиву

def search\_of\_fibonacci(value, arr):

fib\_prev2 = 0 # Позапопереднє число

fib\_prev1 = 1 # Попереднє число

fib\_current = fib\_prev2 + fib\_prev1 # Поточне число

n = len(arr) # Довжина масиву

counter1 = 0 # Кількість порівнянь

counter2 = 0 # Кількість звертань

counter3 = 0 # Кількість ітерацій

# Знаходимо мінімальне число Фібоначчі, яке більше за розмір масиву

while fib\_current < n:

counter1 += 1

counter3 += 1

fib\_prev2 = fib\_prev1

fib\_prev1 = fib\_current

fib\_current = fib\_prev2 + fib\_prev1

counter1 += 1

offset = -1 # Для позначення виключених спереду значень

while fib\_current > 1:

counter1 += 1

counter3 += 1

i = min(offset+fib\_prev2, n-1)

# Якщо елемент менший за потрібний, то беремо ліву частину

counter1 += 1

counter2 += 1

if arr[i] < value:

fib\_current = fib\_prev1

fib\_prev1 = fib\_prev2

fib\_prev2 = fib\_current - fib\_prev1

offset = i

# Якщо елемент більший за потрібний, то беремо праву частину

elif arr[i] > value:

counter1 += 1

counter2 += 1

fib\_current = fib\_prev2

fib\_prev1 = fib\_prev1 - fib\_prev2

fib\_prev2 = fib\_current - fib\_prev1

# Інакше елемент знайдено

else:

counter1 += 1

counter2 += 1

return i, counter1, counter2, counter3

counter1 += 1

counter2 += 1

# Перевірка останнього елемента

if fib\_prev1 and arr[n-1] == value:

return n-1, counter1, counter2, counter3

return -1, counter1, counter2, counter3 # Елемент не знайдено

def main():

# Введення розміру масиву

n = int(input("Введіть розмір масиву/списку: "))

# Ініціалізація масиву унікальними числами

l = []

for i in range(n):

temp = randint(1, n \*\* 2)

while temp in l:

temp = randint(1, n \*\* 2)

l += [temp]

l.sort()

# Створення двозв'язного списку

doublelist = Double\_List()

for i in range(n):

temp = List\_node(l[i])

doublelist.add\_node(temp)

# Виведення масиву та списку

print("Масив: ", l)

print("Двозв'язний список:")

doublelist.output\_list()

# Введення елемента

value = int(input("\nВведіть елемент: "))

# Пошук Фібоначчі

index1, counter1, counter2, counter3 = search\_of\_fibonacci(value, l)

if index1 == -1:

print(f"Елемента {value} в масиві немає!")

else:

print(f"Елемент {value} має індекс {index1 + 1} у масиві")

print(f"Кількість порівнянь у масиві: {counter1}, кількість звертань: {counter2} , кількість ітерацій: {counter3}")

index2, counter1, counter2, counter3 = doublelist.search\_of\_fibonacci(value)

if index2 == -1:

print(f"Елемента {value} в списку немає!")

else:

print(f"Елемент {value} має індекс {index2 + 1} у списку")

print(f"Кількість порівнянь у списку: {counter1}, кількість звертань: {counter2} , кількість ітерацій: {counter3}")

return

if \_\_name\_\_ == "\_\_main\_\_":

main()

3.2.2 Приклад роботи

На рисунках 3.2, 3.3 і 3.4 показані приклади роботи програми для пошуку індекса елемента за ключем для масиву на 100 елементів і двозв'язного списку на 1000 елементів.

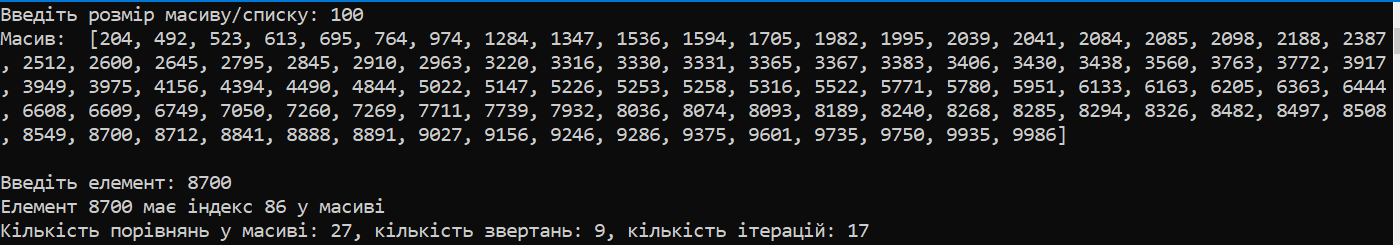


Рисунок 3.2 – Пошук елемента в масиві на 100 елементів

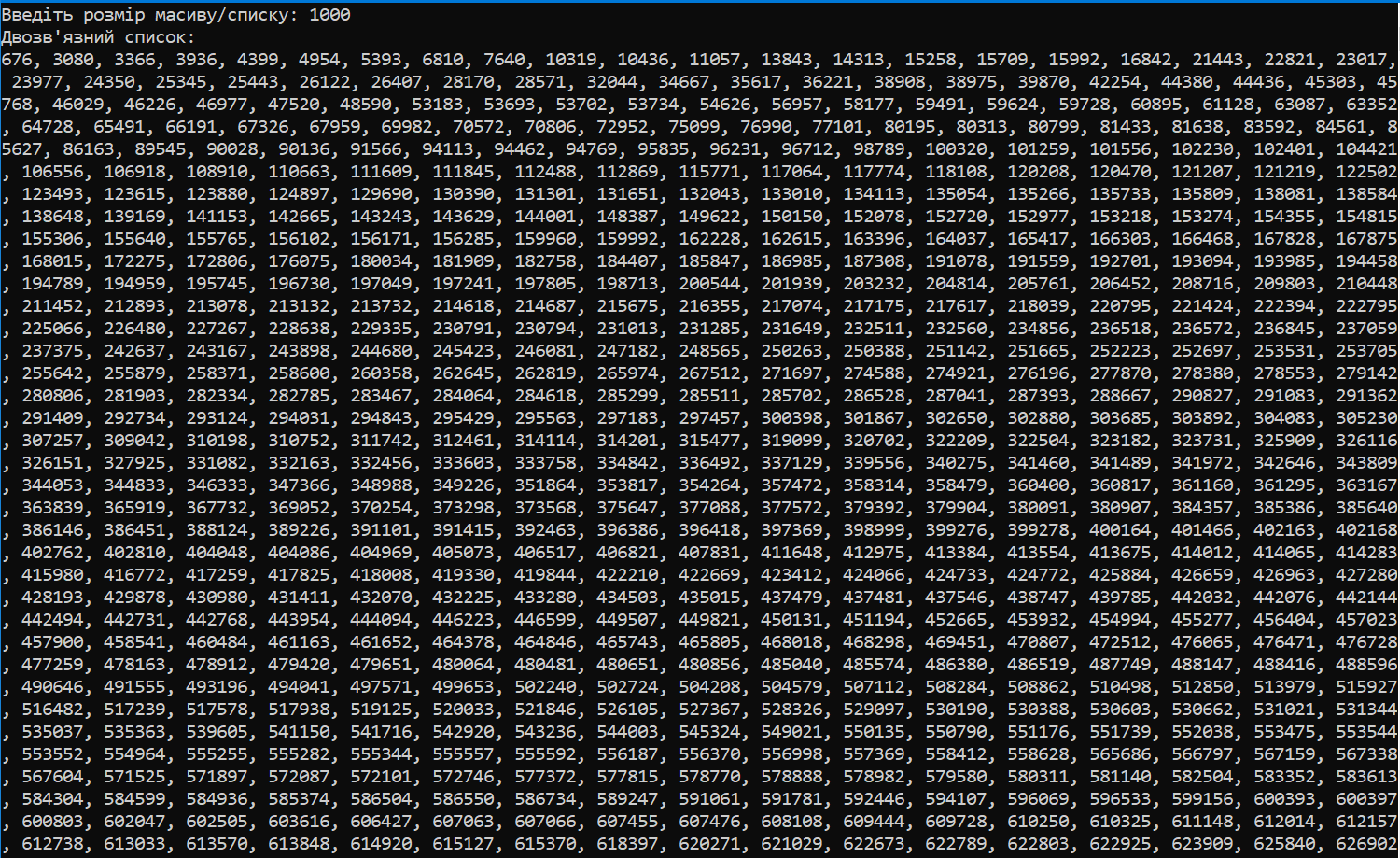


Рисунок 3.3 – Пошук елемента в двозв'язному списку на 1000 елементів (част. 1)

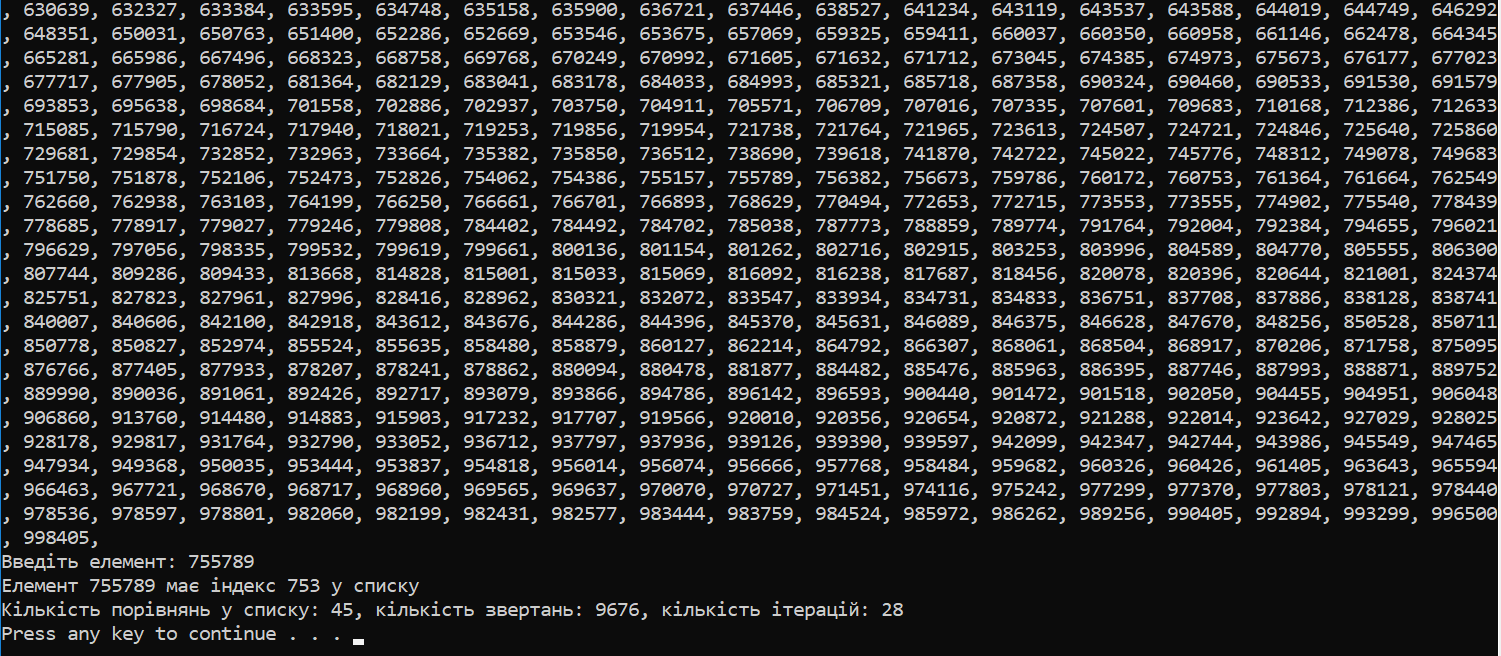


Рисунок 3.4 – Пошук елемента в двозв'язному списку на 1000 елементів (част. 2)

3.4 Тестування алгоритму

3.4.1 Часові характеристики оцінювання

В таблиці 3.1 наведені характеристики оцінювання числа порівнянь при

пошуку елемента і числа звертань при пошуку Фібоначчі для масивів різної

розмірності і двозв'язних списків різної розмірності.

Таблиця 3.1 – Характеристики оцінювання алгоритму пошуку Фібоначчі

|  |  |  |  |
| --- | --- | --- | --- |
| Розмірність масиву/дво-зв'язного списку | Число порівнянь в  масиві/двозв'язному  списку | Число звертань  до елементів  масиву | Число звертань до елементів  двозв'язного  списку |
| 100 | 30 | 11 | 729 |
| 1000 | 42 | 16 | 9587 |
| 5000 | 52 | 19 | 65305 |
| 10000 | 56 | 20 | 145228 |
| 20000 | 59 | 22 | 287276 |

3.4.2 Графіки залежності часових характеристик оцінювання від розмірності структури

На рисунках 3.5 – 3.7 показані графіки залежності часових характеристик оцінювання від розмірності масиву і двозв'язного списку.

Рисунок 3.5 – Графік залежності кількості порівнянь від розміру масиву/списку

Рисунок 3.6 – Графік залежності кількості звертань до елементів масиву від розміру масиву

Рисунок 3.7 – Графік залежності кількості звертань до елементів списку від розміру списку

ВИСНОВОК

В рамках виконання даної лабораторної роботи я опанував алгоритм пошуку Фібоначчі, удосконалив свої навички роботи з масивами та двозв'язними списками, оцінив часову складність пошуку Фібоначчі та побудував графіки залежності часових характеристик від розміру масиву/списку.