МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

КРЕМЕНЧУЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ   
ІМЕНІ МИХАЙЛА ОСТРОГРАДСЬКОГО

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ЕЛЕКТРИЧНОЇ ІНЖЕНЕРІЇ   
ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА АВТОМАТИЗАЦІЇ ТА ІНФОРМАЦІЙНИХ СИСТЕМ

НАВЧАЛЬНА ДИСЦИПЛІНА  
Алгоритми та методи обчислень

ЗВІТ

З ПРАКТИЧНИХ РОБІТ

Виконав:

студент групи КН-23-1

Батраков Є.Є.

Кременчук 2024

# Практична робота № 4

**Тема**. Алгоритми пошуку та їх складність

**Мета**: опанувати основні алгоритми сортування та навчитись методам

аналізу їх асимптотичної складності.

## Хід роботи

### Завдання:

1)

Найкращий випадок O(1)

Найгірший випадок O(n)

**Як можна покращити алгоритм лінійного пошуку**?

Сторожовий (Sentinel) метод, техніка швидкого виходу, індексація або хешування, бінарний пошук, пошук з переліком пропусків (Jump Search)



Найкращий випадок O(1)

Найгірший випадок O(log n)

3)

def ternary\_search(arr, x):

left = 0

right = len(arr) - 1

while left <= right:

left\_third = left + (right - left) // 3

right\_third = right - (right - left) // 3

if arr[left\_third] == x:

return left\_third

elif arr[right\_third] == x:

return right\_third

elif x < arr[left\_third]:

right = left\_third - 1

elif x > arr[right\_third]:

left = right\_third + 1

else:

left = left\_third + 1

right = right\_third - 1

return -1

arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

x = 5

index = ternary\_search(arr, x)

if index != -1:

print(f"Елемент {x} знайдено в позиції {index}.")

else:

print(f"Елемент {x} не знайдено.")

**Асимптотична складність тернарного пошуку в найгіршому випадку**:

В кожному кроці пошуку ми зменшуємо розмір проміжку пошуку приблизно на третину. Таким чином, кількість операцій буде логарифмічною відносно розміру масиву.

Тому складність тернарного пошуку у найгіршому випадку складає O(log₃ n), що еквівалентно O(log n / log 3).

**Асимптотична складність тернарного пошуку в найкращому випадку**:

У найкращому випадку, коли елемент, який шукається, знаходиться одразу на одній з точок розділення, або знаходиться в межах однієї з трьох підмасивів, алгоритм також працює за O(log₃ n), оскільки кожен крок розділення залишає нас з однією третиною вихідного проміжку.

Оптимальність тернарного пошуку порівняно з бінарним полягає в тому, що в середньому він може вимагати менше порівнянь для знаходження елемента відсортованому масиві. Наприклад, бінарний пошук зменшує проміжок вдвічі за кожну ітерацію, тоді як тернарний - втричі, що може зменшити кількість ітерацій у випадку великих масивів. Таким чином, тернарний пошук є оптимальнішим за асимптотичними оцінками порівняно з бінарним пошуком.

4)

import time

import random

import matplotlib.pyplot as plt

def linear\_search(arr, x):

for i in range(len(arr)):

if arr[i] == x:

return i

return -1

def binary\_search(arr, x):

left, right = 0, len(arr) - 1

while left <= right:

mid = left + (right - left) // 2

if arr[mid] == x:

return mid

elif arr[mid] < x:

left = mid + 1

else:

right = mid - 1

return -1

def ternary\_search(arr, x):

left, right = 0, len(arr) - 1

while left <= right:

left\_third = left + (right - left) // 3

right\_third = right - (right - left) // 3

if arr[left\_third] == x:

return left\_third

elif arr[right\_third] == x:

return right\_third

elif x < arr[left\_third]:

right = left\_third - 1

elif x > arr[right\_third]:

left = right\_third + 1

else:

left = left\_third + 1

right = right\_third - 1

return -1

def generate\_sorted\_list(size):

return list(range(size))

def measure\_time(func, arr, x):

start\_time = time.time()

func(arr, x)

end\_time = time.time()

return end\_time - start\_time

sizes = [100, 500, 1000, 5000, 10000]

linear\_times = []

binary\_times = []

ternary\_times = []

for size in sizes:

arr = generate\_sorted\_list(size)

x = random.randint(0, size - 1)

linear\_time = measure\_time(linear\_search, arr, x)

linear\_times.append(linear\_time)

binary\_time = measure\_time(binary\_search, arr, x)

binary\_times.append(binary\_time)

ternary\_time = measure\_time(ternary\_search, arr, x)

ternary\_times.append(ternary\_time)

plt.figure(figsize=(10, 6))

plt.plot(sizes, linear\_times, marker='o', linestyle='-', color='b', label='Linear Search')

plt.plot(sizes, binary\_times, marker='s', linestyle='--', color='r', label='Binary Search')

plt.plot(sizes, ternary\_times, marker='^', linestyle='-.', color='g', label='Ternary Search')

plt.xlabel('Розмір списку')

plt.ylabel('Час виконання (секунди)')

plt.title('Порівняння часу виконання алгоритмів пошуку')

plt.xticks(sizes)

plt.legend()

plt.grid(True)

plt.tight\_layout()

plt.show()

5)

**Лінійний пошук**

Здатність до роботи з відсортованими списками: Лінійний пошук працює з будь-якими списками, оскільки не використовує передумови про відсортованість. Проте, час його виконання не залежить від того, чи відсортований список. Тому, чи список відсортований чи ні, час пошуку буде в середньому однаковий.

Вплив відсортованості на час виконання: Не впливає (час виконання однаковий незалежно від стану списку).

**Бінарний пошук**

Здатність до роботи з відсортованими списками: Бінарний пошук потребує, щоб список був відсортованим. Він використовує можливість швидше знаходити елемент через діленье півідіапазону, що можливе тільки у відсортованому масиві.

Вплив відсортованості на час виконання: Час виконання бінарного пошуку у відсортованому списку значно менший, ніж у не відсортованому. У відсортованому масиві час виконання бінарного пошуку складає O(log n), тоді як у не відсортованому - O(n).

**Тернарний пошук**

Здатність до роботи з відсортованими списками: Тернарний пошук також вимагає відсортованого списку, але в порівнянні з бінарним, він робить більше порівнянь на кожному кроці (2 замість 1). Це може призвести до трохи гіршої ефективності від бінарного, але різниця в сучасних умовах зазвичай не велика.

Вплив відсортованості на час виконання: Тернарний пошук також показує логарифмічну залежність від розміру масиву в відсортованому випадку. В не відсортованому випадку його продуктивність буде аналогічно лінійному пошуку, так як він не може скористатися властивістю відсортованості для зменшення діапазону пошуку.

6)

**Лінійний пошук**

Пошук першого або всіх входжень елемента у несортованому списку.

Перегляд всіх елементів списку у випадках, коли елементи можуть знаходитися на віддалених частинах списку або відсутність відсортованості не має значення.

Якщо розмір списку невеликий або пошук відбувається нечасто, лінійний пошук може бути простим і ефективним рішенням.

**Бінарний пошук**

Пошук у великих відсортованих масивах або списках, де швидкість пошуку є критичним чинником.

При потребі швидкого доступу до елементів за значенням, наприклад, в базах даних або в індексах.

Використання у випадках, коли масив відсортований і його структура не змінюється часто, що дозволяє використовувати бінарний пошук для оптимізації швидкості пошуку.

**Тернарний пошук**

Аналогічно бінарному пошуку, тернарний пошук використовується для швидкого пошуку в відсортованих масивах, але з додатковими порівняннями у кожному кроці.

У випадках, коли немає заздалегідь відомого, який саме алгоритм пошуку буде оптимальним, тернарний пошук може виявитися конкурентоспроможним варіантом.