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 І СТРУКТУРИ ДАНИХ**»

ЗВІТ
З ПРАКТИЧНОЇ РОБОТИ №4

Виконав:
студент групи КН-24-1
Соломка Б. О.

Перевірив:
доцент кафедри АІС
Сидоренко В. М.

Кременчук 2025

Тема: Алгоритми пошуку та їх складність

Мета: опанувати основні алгоритми пошуку та навчитись методам аналізу їх асимптотичної складності

Хід роботи

Звіт з практичної роботи

Тема: Алгоритми пошуку та їх асимптотична складність

1. Лінійний пошук

Алгоритм лінійного пошуку:

Послідовно перевіряємо кожен елемент масиву, поки не знайдемо шуканий елемент або не досягнемо кінця масиву.

```
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i  # Повертаємо індекс знайденого елемента
    return -1  # Елемент не знайдено
```

Асимптотична складність лінійного пошуку:

Найгірший випадок: O(n), коли елемент відсутній або знаходиться в кінці масиву

Найкращий випадок: O(1), коли шуканий елемент знаходиться на першій позиції

Покращення алгоритму лінійного пошуку:

Вартовий (sentinel) метод:

```
def linear_search_sentinel(arr, target):
    # Зберігаємо останній елемент
    last = arr[-1]
    # Замінюємо останній елемент на цільовий
    arr[-1] = target

    i = 0
    # Пошук без перевірки границь масиву
    while arr[i] != target:
        i += 1
```

```
    # Відновлюємо останній елемент
    arr[-1] = last

    # Перевіряємо, чи знайдено шуканий елемент
    if i < len(arr) - 1 or arr[-1] == target:
        return i
    else:
        return -1
```

Використання бінарного пошуку для відсортованих масивів

Паралельна обробка для великих масивів

Індексування або хешування для частого пошуку в стабільних даних

2. Бінарний пошук

Алгоритм бінарного пошуку:

Розділяємо відсортований масив навпіл та перевіряємо, в якій половині знаходиться шуканий елемент. Потім рекурсивно повторюємо процес для вибраної половини.

```
def binary_search(arr, target):
    left, right = 0, len(arr) - 1

    while left <= right:
        mid = left + (right - left) // 2  # Уникаємо переповнення

        if arr[mid] == target:
            return mid  # Елемент знайдено
        elif arr[mid] < target:
            left = mid + 1  # Шукаємо в правій половині
        else:
            right = mid - 1  # Шукаємо в лівій половині

    return -1  # Елемент не знайдено
```

Асимптотична складність бінарного пошуку:

Найгірший випадок: O(log n), коли елемент відсутній або знаходиться на останньому кроці пошуку

Найкращий випадок: O(1), коли шуканий елемент знаходиться посередині масиву

3. Тернарний пошук

Алгоритм тернарного пошуку:

Розділяємо відсортований масив на три приблизно рівні частини і визначаємо, в якій з цих частин знаходиться шуканий елемент. Потім рекурсивно продовжуємо пошук у вибраній частині.

```
def ternary_search(arr, target):
    left, right = 0, len(arr) - 1

    while left <= right:
        # Знаходимо дві точки поділу
        mid1 = left + (right - left) // 3
        mid2 = right - (right - left) // 3

        if arr[mid1] == target:
            return mid1  # Елемент знайдено
        elif arr[mid2] == target:
            return mid2  # Елемент знайдено

        # Визначаємо, в якій частині продовжувати пошук
        if target < arr[mid1]:
            right = mid1 - 1  # Пошук у лівій частині
        elif target > arr[mid2]:
            left = mid2 + 1  # Пошук у правій частині
        else:
            left = mid1 + 1  # Пошук у середній частині
            right = mid2 - 1

    return -1  # Елемент не знайдено
```

Асимптотична складність тернарного пошуку:

Найгірший випадок: $O(\log_3 n) \approx O(\log n / \log 3)$, оскільки на кожному кроці ми відкидаємо 2/3 масиву

Найкращий випадок: O(1), коли шуканий елемент знаходиться на одній з точок поділу

Порівняння бінарного та тернарного пошуку:

Для порівняння ефективності бінарного та тернарного пошуку можна проаналізувати кількість порівнянь, які виконуються у найгіршому випадку:

Бінарний пошук: $\log_2 n$ порівнянь, але на кожному кроці виконується 1

порівняння

Тернарний пошук: $\log_3 n$ кроків, але на кожному кроці виконується 2 порівняння

Загальна кількість порівнянь:

Бінарний: $1 \times \log_2 n = \log_2 n$

Тернарний: $2 \times \log_3 n = 2 \times (\log n / \log 3) \approx 1.26 \times \log_2 n$

Математично: $\log_2 n < 2 \times \log_3 n$, оскільки $\log 3 \approx 1.585$, а $2/1.585 \approx 1.26 > 1$

Таким чином, бінарний пошук виконує менше порівнянь у найгіршому випадку і є асимптотично ефективнішим, незважаючи на те, що тернарний пошук має меншу глибину рекурсії.

4. Експериментальне дослідження ефективності алгоритмів пошуку

```
import time
import random
import matplotlib.pyplot as plt
import numpy as np

def measure_search_time(search_func, arr, target, iterations=100):
    start_time = time.time()
    for _ in range(iterations):
        search_func(arr, target)
    end_time = time.time()
    return (end_time - start_time) / iterations

# Розміри масивів для тестування
sizes = [1000, 5000, 10000, 50000, 100000, 500000]
linear_times = []
binary_times = []
ternary_times = []

for size in sizes:
    # Створюємо відсортований масив
    arr = sorted(list(range(size)))

    # Шукаємо елемент, якого немає (найгірший випадок)
    target = size + 1
```

```
        # Вимірюємо час виконання кожного алгоритму
        linear_time = measure_search_time(linear_search, arr, target)
        binary_time = measure_search_time(binary_search, arr, target)
        ternary_time = measure_search_time(ternary_search, arr, target)

        linear_times.append(linear_time)
        binary_times.append(binary_time)
        ternary_times.append(ternary_time)

    # Побудова графіків
    plt.figure(figsize=(10, 6))
    plt.plot(sizes, linear_times, marker='o', label='Лінійний пошук')
    plt.plot(sizes, binary_times, marker='s', label='Бінарний пошук')
    plt.plot(sizes, ternary_times, marker='^', label='Тернарний пошук')
    plt.xscale('log')
    plt.yscale('log')
    plt.xlabel('Розмір масиву')
    plt.ylabel('Час виконання (секунди)')
    plt.title('Порівняння алгоритмів пошуку')
    plt.legend()
    plt.grid(True)
    plt.savefig('search_comparison.png')
    plt.close()
```

Результати експерименту:

Лінійний пошук показує лінійне зростання часу виконання зі збільшенням розміру масиву

Бінарний та тернарний пошуки демонструють логарифмічне зростання часу

Бінарний пошук у більшості випадків незначно швидший за тернарний

При великих розмірах масиву (> 100,000 елементів) різниця між бінарним і тернарним пошуками стає більш помітною

5. Вплив відсортованості списку на час виконання алгоритмів

```
# Тестування впливу відсортованості
size = 100000
sorted_arr = sorted(list(range(size)))
unsorted_arr = random.sample(range(size * 2), size)  # Несортований масив

# Порівняння для різних сценаріїв
```

```
    targets = [
        sorted_arr[0],            # Перший елемент
        sorted_arr[size // 2],    # Середній елемент
        sorted_arr[-1],           # Останній елемент
        size * 2                  # Відсутній елемент
    ]

    results = {}

    # Тестування на відсортованому масиві
    for i, target in enumerate(targets):
        linear_time = measure_search_time(linear_search, sorted_arr, target)
        binary_time = measure_search_time(binary_search, sorted_arr, target)
        ternary_time = measure_search_time(ternary_search, sorted_arr,
target)

        case = ["Перший", "Середній", "Останній", "Відсутній"][i]
        results[f"Відсортований-{case}"] = {
            "Лінійний": linear_time,
            "Бінарний": binary_time,
            "Тернарний": ternary_time
        }

    # Тестування на невідсортованому масиві (тільки лінійний пошук)
    for i, target in enumerate(targets[:3] + [size * 3]):  # Змінюємо
відсутній елемент
        linear_time = measure_search_time(linear_search, unsorted_arr,
target)

        case = ["Перший", "Середній", "Останній", "Відсутній"][i]
        results[f"Невідсортований-{case}"] = {
            "Лінійний": linear_time
        }
```

Аналіз впливу відсортованості:

Лінійний пошук:

Працює однаково для відсортованих і невідсортованих масивів

Час залежить від позиції шуканого елемента: найкращий випадок (O(1)) - елемент на початку, найгірший (O(n)) - в кінці або відсутній

Бінарний та тернарний пошуки:

Вимагають відсортованого масиву

При спробі використання на невідсортованому масиві можуть давати неправильні результати

Демонструють логарифмічну складність (O(log n)) незалежно від позиції шуканого елемента

Специфічні випадки:

Для малих масивів (< 100 елементів) різниця у швидкодії між алгоритмами майже непомітна

Для середніх та великих масивів лінійний пошук значно повільніший у найгіршому випадку

Якщо елемент знаходиться на початку масиву, лінійний пошук може бути швидшим за бінарний і тернарний

6. Сценарії використання алгоритмів пошуку

Лінійний пошук:

Коли використовувати:

Для невідсортованих масивів, коли сортування недоцільне

Для малих масивів (< 20 елементів), де накладні витрати на інші алгоритми не виправдані

Коли пошук виконується рідко, а елементи часто змінюються

Коли необхідно знайти всі входження шуканого елемента, а не тільки перше

Практичні сценарії:

Пошук у невідсортованих списках користувачів

Перевірка наявності елемента в наборі даних перед додаванням

Одноразовий пошук у динамічних даних

Пошук у масиві об'єктів за нечисловим критерієм

Бінарний пошук:

Коли використовувати:

Для відсортованих масивів з частим пошуком

Коли масив не змінюється часто або його можна попередньо

відсортувати

Для великих масивів, де ефективність пошуку критична

Практичні сценарії:

Пошук у телефонних довідниках, словниках

Пошук у базах даних з індексацією

Визначення позиції вставки нового елемента в відсортований масив

Пошук інформації в відсортованих журналах і логах

Тернарний пошук:

Коли використовувати:

Для відсортованих масивів з унімодальною функцією (для пошуку максимуму або мінімуму)

У специфічних сценаріях оптимізації, особливо для пошуку екстремумів

Коли глибина рекурсії повинна бути мінімізована за рахунок більшої кількості порівнянь на кожному кроці

Практичні сценарії:

Знаходження максимуму або мінімуму функції на інтервалі

Оптимізація параметрів у машинному навчанні

Пошук у системах, де множинні порівняння можуть бути паралелізовані

Висновки:

Під час виконання практичної роботи були зроблені такі висновки:

Асимптотична складність:

Лінійний пошук: O(n) у найгіршому випадку, O(1) у найкращому

Бінарний пошук: $O(\log_2 n)$ у найгіршому випадку, O(1) у найкращому

Тернарний пошук: $O(\log_3 n)$ у найгіршому випадку, O(1) у найкращому

Ефективність:

Бінарний пошук виконує менше порівнянь, ніж тернарний ($\log_2 n < 2 \times \log_3 n$)

Лінійний пошук значно повільніший для великих масивів, але є єдиним варіантом для невідсортованих даних

Практичне використання:

Вибір алгоритму залежить від структури даних, частоти пошуку та змін у даних

Лінійний пошук лишається важливим для невідсортованих даних та малих масивів

Бінарний пошук є оптимальним вибором для більшості сценаріїв з відсортованими даними

Тернарний пошук корисний у специфічних задачах оптимізації

Оптимальність:

З точки зору кількості порівнянь, бінарний пошук є оптимальнішим за тернарний

Однак у системах, де порівняння можуть виконуватися паралельно, тернарний пошук може мати перевагу

Таким чином, для більшості практичних застосувань бінарний пошук є оптимальним алгоритмом для відсортованих даних, а лінійний пошук залишається незамінним для невідсортованих масивів.