Практична робота № 4

Тема. Алгоритми пошуку та їх складність

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

Постановка задачі.

Виконати індивідуальне завдання. Завдання полягає у розв'язку всіх задач, наведених нижче. Тобто студенти виконують єдиний варіант для всіх.

Завдання.

1. Оцінка асимптотичної складності алгоритму лінійного пошуку

Алгоритм лінійного пошуку (Linear Search) передбачає перевірку кожного елемента списку послідовно від початку до кінця.

- **Найгірший випадок**: Елемент, який ми шукаємо, знаходиться в самому кінці списку або відсутній.

  Часова складність: $O(n)$

- **Найкращий випадок**: Елемент знаходиться на першій позиції списку.

  Часова складність: $O(1)$

  Покращення алгоритму лінійного пошуку:

1. **Сторожовий елемент (sentinel)**: Додати шуканий елемент у кінець списку як сторожовий, що дозволяє уникнути перевірки виходу за межі списку.
2. **Паралельний пошук**: Використовувати кілька потоків для одночасного пошуку в різних частинах списку.

2. Оцінка асимптотичної складності алгоритму бінарного пошуку

Алгоритм бінарного пошуку (Binary Search) працює тільки на відсортованих списках і ділить список навпіл на кожній ітерації.

- **Найгірший випадок**: Необхідно шукати до останньої можливої позиції.
  - Часова складність: $O(\log n)$
- **Найкращий випадок**: Елемент знаходиться точно посередині списку.
  - Часова складність: $O(1)$

3. Алгоритм тернарного пошуку та оцінка його асимптотичної складності

Тернарний пошук (Ternary Search) також працює на відсортованих списках і ділить список на три частини.

Псевдокод:

```
def ternary_search(arr, left, right, x):
  if right >= left:
    mid1 = left + (right - left) // 3
    mid2 = right - (right - left) // 3

    if arr[mid1] == x:
      return mid1
    if arr[mid2] == x:
      return mid2

    if x < arr[mid1]:
      return ternary_search(arr, left, mid1 - 1, x)
    elif x > arr[mid2]:
      return ternary_search(arr, mid2 + 1, right, x)
    else:
      return ternary_search(arr, mid1 + 1, mid2 - 1, x)

  return -1
```

Оцінка асимптотичної складності:

- **Найгірший випадок**: Необхідно ділити список до останньої можливої позиції.
  - Часова складність: $O(\log_3 n)=O(\log n/\log 3)=O(\log n)$
- **Найкращий випадок**: Елемент знаходиться точно на одній з трьох розділених позицій.
  - Часова складність: $O(1)$O(1)

Оптимальність алгоритмів:

- Обидва алгоритми, бінарний і тернарний пошук, мають асимптотичну складність $O(\log n)$.
- Бінарний пошук зазвичай більш ефективний на практиці через меншу кількість порівнянь на кожному кроці (2 порівняння замість 4 в тернарному пошуку).

4. Експериментальне дослідження ефективності алгоритмів пошуку

Для проведення експериментального дослідження створимо Python-скрипт, який порівняє час виконання лінійного, бінарного і тернарного пошуку на різних розмірах вхідних списків.

```
import time

import random


def linear_search(arr, x):

  for i in range(len(arr)):

    if arr[i] == x:

      return i

  return -1


def binary_search(arr, left, right, x):

  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, left, right, x):
    if right >= left:
        mid1 = left + (right - left) // 3
        mid2 = right - (right - left) // 3
        if arr[mid1] == x:
            return mid1
        if arr[mid2] == x:
            return mid2
        if x < arr[mid1]:
            return ternary_search(arr, left, mid1 - 1, x)
        elif x > arr[mid2]:
            return ternary_search(arr, mid2 + 1, right, x)
        else:
            return ternary_search(arr, mid1 + 1, mid2 - 1, x)
    return -1

sizes = [10**i for i in range(1, 7)]
search_value = -1  # Assuming search for a value not in list to simulate worst case

for size in sizes:
    arr = sorted(random.sample(range(size * 10), size))

    start_time = time.time()
    linear_search(arr, search_value)
    print(f"Linear Search: Size = {size}, Time = {time.time() - start_time:.6f} seconds")

    start_time = time.time()
    binary_search(arr, 0, len(arr) - 1, search_value)
```

```
print(f"Binary Search: Size = {size}, Time = {time.time() - start_time:.6f} seconds")


start_time = time.time()
ternary_search(arr, 0, len(arr) - 1, search_value)
print(f"Ternary Search: Size = {size}, Time = {time.time() - start_time:.6f} seconds")
```

Цей код проводить експериментальне дослідження часу виконання для різних розмірів вхідних списків, порівнюючи лінійний, бінарний та тернарний пошуки. Ви можете використати бібліотеки для побудови графіків, такі як Matplotlib, для візуалізації результатів.

5. Порівняння алгоритмів пошуку за їхньою здатністю працювати з відсортованими та не відсортованими списками

- **Лінійний пошук**:
  - Працює як з відсортованими, так і з невідсортованими списками.
  - Час виконання не залежить від відсортованості списку.
- **Бінарний пошук**:
  - Працює тільки з відсортованими списками.
  - Відсортованість списку є обов'язковою умовою.
- **Тернарний пошук**:
  - Працює тільки з відсортованими списками.
  - Відсортованість списку є обов'язковою умовою.

6. Сценарії використання кожного з алгоритмів пошуку у практичних задачах

- **Лінійний пошук**:
  - Використовується, коли дані не відсортовані або коли набір даних дуже малий.
  - Наприклад, пошук конкретного елемента в списку імен, який рідко сортується.
- **Бінарний пошук**:
  - Використовується, коли дані відсортовані і необхідно забезпечити швидкий пошук.
  - Наприклад, пошук у телефонній книзі або базі даних, де дані вже відсортовані за ключем.
- **Тернарний пошук**:
  - Може бути корисним у випадках, коли зменшення кількості рекурсивних викликів важливіше, ніж кількість порівнянь.

- Наприклад, пошук у великих відсортованих масивах, де можлива оптимізація за кількістю рекурсивних викликів.

Підсумовуючи, вибір алгоритму залежить від характеристик вхідних даних (відсортованість, розмір) і конкретних вимог до швидкості та ефективності у даній задачі.

Контрольні запитання

1. Що таке алгоритм пошуку і чому він важливий у контексті комп'ютерних наук?

Алгоритм пошуку – це процедура або формула, яка дозволяє знайти певний елемент у колекції даних, такій як масив, список або база даних. Вони важливі в контексті комп'ютерних наук через їхню широку застосовність у різноманітних задачах, від простого пошуку елементів до складних операцій у базах даних, інформаційних системах, алгоритмах оптимізації та штучному інтелекті. Ефективність алгоритмів пошуку безпосередньо впливає на продуктивність програмного забезпечення і систем в цілому.

2. Які основні критерії оцінки ефективності алгоритмів пошуку?

Основні критерії оцінки ефективності алгоритмів пошуку включають:

- **Часова складність**: Визначає, скільки часу займає виконання алгоритму відносно розміру вхідних даних.
- **Просторова складність**: Визначає, скільки пам'яті потребує алгоритм під час свого виконання.
- **Найгірший, середній і найкращий випадки**: Оцінка продуктивності алгоритму в різних сценаріях.
- **Стабільність**: Важливо для деяких пошукових алгоритмів, щоб зберегти порядок однакових елементів.
- **Простота реалізації**: Наскільки легко реалізувати і підтримувати алгоритм.

3. Що таке лінійний пошук, і як він працює?

Лінійний пошук (Linear Search) – це алгоритм, який перевіряє кожен елемент в послідовності, поки не знайде шуканий елемент або не досягне кінця послідовності. Він працює наступним чином:

3. Починає з першого елемента списку.
4. Порівнює кожен елемент зі шуканим значенням.
5. Якщо знайдено шуканий елемент, алгоритм повертає його позицію.

6. Якщо досягнуто кінця списку без знаходження елемента, алгоритм повертає, що елемент не знайдений.

4. Які умови повинні бути виконані для успішного застосування бінарного пошуку?

Для успішного застосування бінарного пошуку (Binary Search) повинні бути виконані наступні умови:

7. **Відсортований список**: Дані повинні бути впорядковані.
8. **Доступ до середнього елемента**: Можливість ділити список на дві половини і порівнювати середній елемент з шуканим значенням.

5. Які переваги та недоліки використання бінарного пошуку порівняно з іншими алгоритмами пошуку?

Переваги бінарного пошуку:

- **Ефективність**: Часова складність $O(\log n)$, що робить його значно швидшим для великих списків порівняно з лінійним пошуком.
- **Мала просторова складність**: Потребує мало додаткової пам'яті, зазвичай $O(1)$.

Недоліки бінарного пошуку:

- **Вимога відсортованості**: Список повинен бути попередньо відсортованим, що може потребувати додаткового часу.
- **Складність реалізації**: Може бути складнішим для реалізації та налагодження порівняно з лінійним пошуком.

6. Що таке тернарний пошук, і в чому його відмінність від бінарного пошуку?

Тернарний пошук (Ternary Search) – це алгоритм, який розбиває список на три частини замість двох, як у бінарному пошуку. Він працює наступним чином:

9. Визначає дві середні точки, що ділять список на три частини.
10. Порівнює шуканий елемент з елементами на цих середніх позиціях.
11. В залежності від результату порівняння, продовжує пошук в одній з трьох частин.
12. Повторює процес до знаходження елемента або завершення списку.

Відмінності від бінарного пошуку:

- **Поділ на три частини**: Тернарний пошук ділить список на три частини, що теоретично зменшує кількість рекурсивних викликів.

- **Складність реалізації**: Тернарний пошук складніший в реалізації через більше порівнянь і визначень середніх точок.
- **Часова складність**: Теоретично $O(\log_3 n)$, що еквівалентно $O(\log_2 n)$, але на практиці частіше менш ефективний через більшу кількість порівнянь на кожному кроці.