МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

КРЕМЕНЧУЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

ІМЕНІ МИХАЙЛА ОСТРОГРАДСЬКОГО

Кафедра комп'ютерної інженерії та електроніки

ЗВІТ З ПРАКТИЧНИХ РОБІТ

з навчальної дисципліни

«Алгоритми та методи обчислень»

Тема «Алгоритми сортування та їх складність. Порівняння алгоритмів сортування»

Студент гр. КІ-23-1 ПІБ Кобець О. О.

Кременчук 2024

Практична робота №3

Тема. Алгоритми сортування та їх складність. Порівняння алгоритмів Сортування

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

Завдання

1. Вивчити самостійно і записати (будь-яким способом) алгоритм бульбашкового сортування. Оцінити асимптотику алгоритму сортування методом бульбашки в найгіршому і в найкращому випадку. Порівняти за цими показниками бульбашковий алгоритм з алгоритмом сортування вставлянням. Чому на практиці бульбашковий алгоритм виявляється менш ефективним у порівнянні з сортуванням методом зливанням?

2. Оцінити асимптотичну складність алгоритму сортування зливанням, скориставшись основною теоремою рекурсії.

3. Вивчити і записати (будь-яким способом) самостійно алгоритм швидкого сортування. Оцінити асимптотичну складність алгоритму швидкого сортування, скориставшись основною теоремою рекурсії.

1. Алгоритм бульбашкового сортування працює за принципом багаторазових порівнянь сусідніх елементів і їх обміну місцями, якщо вони знаходяться в неправильному порядку. Кожен прохід по масиву дозволяє "випустити" найбільший елемент у відсортовану частину масиву, тому кожен наступний прохід потребує перевірки все меншої кількості елементів.

Алгоритм:

1. Пройти через масив, порівнюючи кожен елемент з наступним.
2. Якщо поточний елемент більший за наступний, поміняти їх місцями.
3. Продовжувати проходи через масив, поки немає більше обмінів.
4. Повторювати цей процес до повного сортування.

```python
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        swapped = False
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True
        if not swapped:
            break

```

Оцінка асимптотики:

- **Найгірший випадок**: Якщо масив відсортований у зворотному порядку, алгоритм бульбашкового сортування виконає всі можливі обміни. В такому випадку кількість порівнянь дорівнює $O(n^2)$, де n — кількість елементів у масиві.
- **Найкращий випадок**: Якщо масив вже відсортований, алгоритм зробить лише один прохід без жодних обмінів, тому асимптотична складність буде $O(n)$.
- **Середній випадок**: Загальна складність також буде $O(n^2)$, оскільки в середньому алгоритм все одно виконає значну кількість обмінів.

Порівняння з сортуванням вставлянням:

- **Сортування вставлянням**: У найгіршому випадку (якщо масив відсортований у зворотному порядку), сортування вставлянням має складність $O(n^2)$, але в найкращому випадку (якщо масив вже відсортований), його складність буде $O(n)$.
- **Бульбашкове сортування**: В найгіршому випадку також має складність $O(n^2)$, але часто має більші постійні множники через зайві обміни на кожному кроці.

Чому бульбашковий алгоритм менш ефективний за сортування методом зливання?

- Бульбашкове сортування має погану ефективність через постійні зайві порівняння та обміни, що призводить до великих витрат часу, особливо при великих розмірах масиву.

- **Сортування методом зливання** використовує принцип розділяй і володарюй, що дозволяє розділяти масив на менші частини і обробляти їх окремо, що дає більш ефективну асимптотику O(n log n).

2. **Алгоритм сортування зливанням** працює за принципом розділення масиву на дві половини, сортування кожної з них рекурсивно, а потім злиття двох відсортованих частин.

```
8   def merge_sort(arr):  2 usages
9       if len(arr) <= 1:
10          return arr
11      mid = len(arr) // 2
12      left = merge_sort(arr[:mid])
13      right = merge_sort(arr[mid:])
14      return merge(left, right)
15
16  def merge(left, right):  1 usage
17      result = []
18      i = j = 0
19      while i < len(left) and j < len(right):
20          if left[i] < right[j]:
21              result.append(left[i])
22              i += 1
23          else:
24              result.append(right[j])
25              j += 1
26      result.extend(left[i:])
27      result.extend(right[j:])
28      return result
```

Асимптотика:

- При кожному розподілі масиву на дві частини складність роботи розбиття — O(1).
- Для злиття двох масивів, кожен елемент масиву опрацьовується один раз, тому складність злиття двох масивів довжиною n_1 та n_2 буде $O(n_1+n_2)$.
- Оскільки ми ділимо масив на два на кожному кроці рекурсії, на кожному рівні глибини рекурсії маємо O(n) робіт, а рівнів рекурсії буде O(log n).

3. Швидке сортування працює за принципом вибору опорного елемента, після чого всі елементи, менші за опорний, переміщуються в одну частину масиву, а більші — в іншу. Потім алгоритм рекурсивно сортує ці дві частини.

```python
def quick_sort(arr):  2 usages
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quick_sort(left) + middle + quick_sort(right)
```

Оцінка асимптоти:

- **Найгірший випадок**: Якщо масив уже відсортований або в зворотному порядку (наприклад, при виборі опорного елемента як крайнього), швидке сортування працює за $O(n^2)$, оскільки кожен розподіл буде лише на один елемент.
- **Найкращий і середній випадок**: При випадковому виборі опорного елемента і балансуванні розподілів на дві рівні частини, складність буде $O(n \log n)$.

Асимптотична складність алгоритму швидкого сортування: в середньому випадку $O(n \log n)$, але в найгіршому випадку $O(n^2)$.

Основна теорема рекурсії: Якщо ми розглянемо рекурсивні виклики для кожного рівня, то глибина рекурсії буде $O(\log n)$, а кожен рівень виконує $O(n)$ операцій (для розбиття і злиття елементів).

Контрольні питання

1. Що таке асимптотична складність алгоритму сортування і чому вона важлива для порівняння алгоритмів?

Асимптотична складність алгоритму сортування — це міра того, як час виконання або кількість операцій алгоритму змінюється залежно від розміру вхідних даних (наприклад, кількості елементів в масиві). Вона використовується для оцінки ефективності алгоритму в теоретичному плані, зазвичай в умовах великих розмірів входу, щоб визначити, як швидко алгоритм працюватиме на великих обсягах даних.

Важливість:

- Асимптотична складність дає уявлення про масштабованість алгоритму. Алгоритми з низькою асимптотичною складністю працюють ефективніше на великих наборах даних.
- Вона дозволяє порівнювати різні алгоритми, навіть якщо конкретні реалізації мають різні постійні множники. Наприклад, алгоритм з асимптотою O(n log n) може працювати швидше, ніж $O(n^2)$, навіть якщо останній працює швидше для малих розмірів входу.

2. Які алгоритми сортування мають квадратичну складність у найгіршому випадку? Поясніть, чому це може бути проблемою для великих обсягів даних.

Алгоритми з квадратичною складністю у найгіршому випадку:

- Бульбашкове сортування
- Сортування вставками
- Сортування вибором

Проблеми для великих обсягів даних:

- Для великих масивів ці алгоритми виконують занадто багато порівнянь і обмінів місцями (до n^2), що робить їх надзвичайно неефективними.
- Наприклад, для масиву з 1 мільйоном елементів кількість операцій може досягати 10^{12}, що значно перевищує час, необхідний для більш ефективних алгоритмів (наприклад, O(n log n)).

3. В чому полягає перевага сортування злиттям над сортуванням вставками для великих наборів даних?

Перевага сортування злиттям:

- **Часова складність** сортування злиттям у найгіршому випадку становить O(n log n), тоді як сортування вставками має складність $O(n^2)$ у найгіршому випадку.
- Сортування злиттям завжди має сталу складність, незалежно від початкового стану даних (чи відсортовані вони, чи ні), а сортування вставками може бути дуже ефективним при майже відсортованих даних, але не підходить для великих обсягів.

Для великих наборів даних сортування злиттям буде набагато швидшим, оскільки його складність зростає значно повільніше (за логарифмічною шкалою), що дозволяє обробляти великі обсяги даних набагато ефективніше.

4. Які алгоритми сортування використовуються для сортування списків у стандартних бібліотеках мов програмування, таких як Python, Java або C++?

- **Python**: Використовує алгоритм **Timsort**, який є адаптивною комбінацією сортування злиттям і сортування вставками. Він має складність O(n log n) у найгіршому випадку та O(n) у найкращому випадку для майже відсортованих даних.
- **Java**: Для масивів використовується **Merge Sort** або **TimSort** (в залежності від реалізації), а для списків — **MergeSort**.
- **C++**: Стандартна бібліотека STL використовує алгоритм **IntroSort**, який є гібридним і комбінує **QuickSort**, **HeapSort** і **Insertion Sort** для забезпечення високої продуктивності в різних випадках.

5. Яка різниця між алгоритмами сортування злиттям і швидким сортуванням? У яких випадках краще використовувати кожен з цих алгоритмів?

Різниця:

- **Швидке сортування** зазвичай працює швидше на середньому і найкращому випадку, має складність O(n log n), але в найгіршому випадку (коли вибір опорного елемента не вдалий) може мати складність $O(n^2)$.
- **Сортування злиттям** завжди має складність O(n log n), навіть у найгіршому випадку. Він стабільний, але вимагає додаткової пам'яті для злиття частин.

Коли використовувати:

- **Швидке сортування** краще використовувати для загальних випадків, особливо якщо дані випадкові і немає обмежень на пам'ять.

- **Сортування злиттям** краще використовувати, коли важлива стабільність сортування (не змінювати порядок елементів з однаковими значеннями) або коли потрібно обробляти дуже великі масиви з обмеженими ресурсами пам'яті.

6. Які фактори слід враховувати при виборі алгоритму сортування для конкретної задачі?

- **Розмір даних**: Для великих наборів даних алгоритми з лінійно-логарифмічною складністю, такі як швидке сортування або сортування злиттям, зазвичай кращі, ніж квадратичні.
- **Стабільність сортування**: Якщо важливо зберегти порядок елементів з однаковими значеннями, вибирайте стабільні алгоритми, такі як сортування злиттям.
- **Часова складність**: Якщо ваші дані вже відсортовані або майже відсортовані, алгоритми з квадратичною складністю, такі як сортування вставками, можуть бути достатньо ефективними.
- **Пам'ять**: Деякі алгоритми, як сортування злиттям, вимагають додаткової пам'яті, що може бути важливим фактором для великих даних.
- **Динамічні зміни даних**: Для частих оновлень даних можуть бути корисними адаптивні алгоритми, як Timsort або гібридні алгоритми (IntroSort).