ĐẠI HỌC QUỐC GIA TP.HCM

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

KHOA CÔNG NGHỆ THÔNG TIN

**Logo

Description automatically generated**

ĐỀ TÀI:

**KHẢO SÁT CÁC THUẬT TOÁN SẮP XẾP**

**Giảng viên hướng dẫn :** Lê Thanh Phong

**Bộ môn :** Cấu trúc dữ liệu và giải thuật

**Thành phố Hồ Chí Minh, tháng 12 năm 2023**

**THÀNH VIÊN NHÓM**

|  |  |
| --- | --- |
| **Họ tên** | **Mã số sinh viên** |
| **Lê Phan Minh Đạt** | **21120046** |
| **Phạm Gia Khang** | **22120152** |
| **Lê Anh Khôi** | **22120165** |
| **Nguyễn Tuấn Kiệt** | **22120173** |

[GIỚI THIỆU 6](#_Toc155384986)

[CÁC THUẬT TOÁN SẮP XẾP 7](#_Toc155384987)

[1. Selection Sort 7](#_Toc155384988)

[1.1. Giới thiệu 7](#_Toc155384989)

[1.2. Ý tưởng 7](#_Toc155384990)

[1.3. Thuật toán 7](#_Toc155384991)

[1.4. Độ phức tạp 9](#_Toc155384992)

[1.5. Cải tiến: 9](#_Toc155384993)

[2. Insertion Sort 9](#_Toc155384994)

[2.1. Giới thiệu 9](#_Toc155384995)

[2.2. Ý tưởng 10](#_Toc155384996)

[2.3. Thuật toán 10](#_Toc155384997)

[2.4. Độ phức tạp 11](#_Toc155384998)

[3. Bubble Sort 11](#_Toc155384999)

[3.1. Giới thiệu 11](#_Toc155385000)

[3.2. Ý tưởng 11](#_Toc155385001)

[3.3. Thuật toán 11](#_Toc155385002)

[3.4. Độ phức tạp 13](#_Toc155385003)

[4. Heap Sort 14](#_Toc155385004)

[4.1. Nguồn gốc 14](#_Toc155385005)

[4.2. Cấu trúc Heap 14](#_Toc155385006)

[4.3. Ý tưởng 14](#_Toc155385007)

[4.4. Thuật toán 14](#_Toc155385008)

[4.5. Độ phức tạp 15](#_Toc155385009)

[5. Merge Sort 16](#_Toc155385010)

[5.1. Nguồn gốc 16](#_Toc155385011)

[5.2. Ý tưởng 16](#_Toc155385012)

[5.3. Thuật toán 17](#_Toc155385013)

[5.4. Độ phức tạp 18](#_Toc155385014)

[6. Quick Sort 18](#_Toc155385015)

[6.1. Nguồn gốc 18](#_Toc155385016)

[6.2. Ý tưởng 18](#_Toc155385017)

[6.3. Thuật toán 19](#_Toc155385018)

[6.4. Độ phức tạp 24](#_Toc155385019)

[7. Shaker Sort 25](#_Toc155385020)

[7.1. Ý tưởng 25](#_Toc155385021)

[7.2. Thuật toán 25](#_Toc155385022)

[7.3. Độ phức tạp 29](#_Toc155385023)

[8. Shell Sort 30](#_Toc155385024)

[8.1. Ý tưởng 30](#_Toc155385025)

[8.2. Thuật toán 30](#_Toc155385026)

[8.3. Độ phức tạp 31](#_Toc155385027)

[9. Counting Sort 32](#_Toc155385028)

[9.1. Ý tưởng 32](#_Toc155385029)

[9.2. Thuật toán 32](#_Toc155385030)

[9.3. Độ phức tạp 36](#_Toc155385031)

[10. Radix Sort 37](#_Toc155385032)

[10.1. Ý tưởng 37](#_Toc155385033)

[10.2. Thuật toán 37](#_Toc155385034)

[10.3. Độ phức tạp 40](#_Toc155385035)

[11. Flash Sort 40](#_Toc155385036)

[11.1. Ý tưởng 40](#_Toc155385037)

[11.2. Thuật toán 40](#_Toc155385038)

[11.3. Độ phức tạp 46](#_Toc155385039)

[KẾT QUẢ THỰC NGHIỆM 47](#_Toc155385040)

[1. Dữ liệu đã được sắp xếp tăng dần: 47](#_Toc155385041)

[2. Dữ liệu sắp xếp gần đúng: 50](#_Toc155385042)

[3. Dữ liệu được sắp xếp ngược: 52](#_Toc155385043)

[4. Dữ liệu ngẫu nhiên: 55](#_Toc155385044)

[TỔNG QUAN VỀ CHƯƠNG TRÌNH 57](#_Toc155385045)

[1. Cách tổ chức source code 57](#_Toc155385046)

[2. Các thư viện đặc biệt 58](#_Toc155385047)

[3. Thông số máy tính và IDE đã dùng 58](#_Toc155385048)

[4. Chương trình Command 58](#_Toc155385049)

[TÀI LIỆU THAM KHẢO 62](#_Toc155385050)

# GIỚI THIỆU

Bài tập "Khảo sát các thuật toán sắp xếp" là một nghiên cứu chi tiết về các phương pháp sắp xếp dữ liệu trong lĩnh vực khoa học máy tính bao gồm thuật toán bằng C++, báo cáo về ý tưởng, thuật toán, hiệu suất, số liệu chứng minh,…. Chúng em đã tập trung vào việc hiểu rõ cách mà các thuật toán như Selection Sort, Insertion Sort, Bubble Sort, Shell Sort, Shaker Sort, Heap Sort, Merge Sort, Quick Sort, Radix Sort, Flash Sort, Counting Sort hoạt động trên dữ liệu khác nhau như Ranom Data, Reversed Data, Sorted Data, và Nearly Sorted Data, với kích thước trải dài từ 10000 phần tử đến 500000 phần tử.

Với mục tiêu đề xuất những cách tiếp cận hiệu quả nhất để sắp xếp dữ liệu, đồ án này không chỉ giới thiệu cơ bản về mỗi thuật toán mà còn phân tích hiệu suất của chúng dựa trên các trường hợp thử nghiệm.

Chúng em đã đánh giá sự ổn định, hiệu suất, và độ phức tạp của từng thuật toán để cung cấp cái nhìn toàn diện về cách chúng hoạt động trong các ngữ cảnh khác nhau. Những kiến thức này không chỉ hỗ trợ việc chọn lựa thuật toán phù hợp cho một tình huống cụ thể mà còn giúp hiểu rõ hơn về cơ sở lý thuyết của sắp xếp dữ liệu, từ đó trang bị những hành trang đầy đủ nhất khi thực hành các bài tập về sắp xếp.

Đồ án này không chỉ mang tính chất nghiên cứu mà còn hướng dẫn cách triển khai mỗi thuật toán trong ngôn ngữ lập trình C++. Chúng em hy vọng rằng đồ án sẽ cung cấp cái nhìn sâu sắc và thiết thực, giúp bạn hiểu rõ hơn về các thuật toán sắp xếp và ứng dụng chúng trong thực tế.

# CÁC THUẬT TOÁN SẮP XẾP

# Selection Sort

### Giới thiệu

Selection Sort là một thuật toán sắp xếp đơn giản và cổ điển, được phát triển bởi nhà toán học Robert W. Floyd vào năm 1964. Tuy nhiên, trước đó, nó đã được phát triển bởi nhà toán học người Anh John von Neumann vào những năm 1945.

Thuật toán Selection Sort là một trong những thuật toán đơn giản nhất để sắp xếp một danh sách các phần tử. Nó hoạt động bằng cách lần lượt tìm kiếm phần tử nhỏ nhất trong danh sách và đưa nó về vị trí đầu tiên. Sau đó, thuật toán tiếp tục tìm kiếm phần tử nhỏ nhất trong phần còn lại của danh sách và đưa nó về vị trí thứ hai. Thuật toán tiếp tục thực hiện như vậy cho tới khi toàn bộ danh sách được sắp xếp.

Mặc dù Selection Sort có hiệu quả không cao như các thuật toán sắp xếp khác, nhưng nó vẫn được sử dụng trong một số trường hợp đơn giản hoặc khi các dữ liệu đầu vào có kích thước nhỏ.

### Ý tưởng

**Bước 1:** Khởi tạo biến index = -1 đại diện cho vị trí lớn nhất của mảng con đã được sắp xếp, duyệt mảng con chưa sắp xếp từ trái qua phải từ vị trí index + 1 đến vị trí n - 1, tìm phần tử cực trị, đổi chỗ phần tử cực trị này với phần tử index + 1, tăng i lên 1.

**Bước 2:** Lặp lại bước 1 với điều kiện dừng là index = n - 1.

**Bước 3:** Trả về mảng cuối cùng. Mảng cuối cùng là mảng đã được sắp xếp.

### Thuật toán

Giả sử cho một mảng s có 7 phần tử “3, 8, 1, 5, 4, 2, 7” và yêu cầu sắp xếp chúng theo thứ tự tăng dần.

**Bước 1:** Khởi tạo biến index = -1. Duyệt mảng từ trái sang phải mảng chưa sắp xếp (s1).

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3 | 8 | 1 | 5 | 4 | 2 | 7 |

* index = -1:
* Min(s1) = 1
* Đổi chỗ s[index + 1] và 1, ta được mảng mới:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 8 | 3 | 5 | 4 | 2 | 7 |

**Bước 2:** Lặp lại bước 1 với điều kiện dừng là index = n-1:

* index = 0:
* Min(s1) = 2
* Đổi chỗ s[index + 1] và 2, ta được mảng mới:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 5 | 4 | 8 | 7 |

* index=1:
* Min(s1) = 3
* Đổi chỗ s[index + 1] và 3, ta được mảng mới:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 5 | 4 | 8 | 7 |

* index=2:
* Min(s1) = 4
* Đổi chỗ s[index + 1] và 4, ta được mảng mới:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 8 | 7 |

* index=3:
* Min(s1)=5
* Đổi chỗ s[index + 1] và 5, ta được mảng mới:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 8 | 7 |

* index=4:
* Min(s1)=7
* Đổi chỗ s[index + 1] và 7, ta được mảng mới:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 7 | 8 |

* index=5:
* Min(s1)=8
* Đổi chỗ s[index + 1] và 8, ta được mảng mới:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 7 | 8 |

**Bước 3:** Trả về mảng cuối cùng từ **Bước 2**. Mảng này là mảng đã sắp xếp:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 7 | 8 |

### Độ phức tạp

|  |  |  |  |
| --- | --- | --- | --- |
| Trường hợp | Tốt nhất | Trung bình | Xấu nhất |
| Độ phức tạp | O(n2) | O(n2) | O(n2) |

Giải thích:

Với cả ba trường hợp xấu nhất, trung bình, và tốt nhất, tổng số vòng lặp luôn là:

### Cải tiến:

Có nhất thiết phải hoán đổi giá trị trùng nhau?

Đôi khi phần tử min của mảng con chưa sắp xếp (s1) bằng với phần tử max của mảng con đã sắp xếp (s0), trong trường hợp này không cần hoán đổi. Chỉ thêm một bước so sánh giữa min của s1 và max của s2, ta có thể giản lược bớt quá trình đổi chỗ các phần tử trùng nhau.

## Insertion Sort

### Giới thiệu

Insertion Sort là một thuật toán sắp xếp đơn giản hoạt động tương tự như cách bạn sắp xếp các thẻ chơi trong tay của mình. Mảng hầu như được chia thành một phần được sắp xếp và một phần chưa được sắp xếp. Các giá trị từ phần chưa được sắp xếp được chọn và đặt ở vị trí chính xác trong phần được sắp xếp.

### Ý tưởng

* Chia dãy thành hai phần: một phần đã sắp xếp và một phần chưa sắp xếp.
* Duyệt qua từng phần tử trong phần chưa sắp xếp và chèn nó vào đúng vị trí trong phần đã sắp xếp.

### Thuật toán

* Bắt đầu với phần tử thứ hai, so sánh nó với tất cả các phần tử trong phần đã sắp xếp và chèn vào đúng vị trí.
* Lặp lại quá trình trên cho các phần tử tiếp theo.
* Kết thúc khi toàn bộ dãy đã sắp xếp.

Sắp xếp mảng tăng dần có 5 phần tử sau:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 4 | 10 | 2 | 12 |

**Bước 1:** Chia mảng làm 2 phần , một phần đã sắp xếp là a[0] và phần còn lại chưa sắp xếp từ a[1] tới a[4]

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 4 | 10 | 2 | 12 |

**Bước 2:** So sánh a[1] với a[0] (4 < 5). chèn 4 trước 5

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 4 | 5 | 10 | 2 | 12 |

**Bước 3:** So sánh a[2] với từ a[0] đến a[1] (5 < 10). Giữ nguyên mảng.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 4 | 5 | 10 | 2 | 12 |

**Bước 4:** So sánh a[3] với từ a[0] đến a[2] (2 < 4) . Chèn tại vị trí i = 0 và dịch chuyển các giá trị 4, 5, 10 sang phải

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2 | 4 | 5 | 10 | 12 |

**Bước 5:** So sánh a[4] từ a[0] đến a[3] . Giữ nguyên vị trí của a[4].

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2 | 4 | 5 | 10 | 12 |

Ta được mảng đã sắp xếp xong là: 2 - 4 - 5 - 10 - 12.

### Độ phức tạp

|  |  |  |  |
| --- | --- | --- | --- |
| Trường hợp | Tốt nhất | Trung bình | Xấu nhất |
| Độ phức tạp | O(n) | O(n2) | O(n2) |

## Bubble Sort

### Giới thiệu

Bubble Sort là một thuật toán sắp xếp đơn giản, với thao tác cơ bản là so sánh hai phần tử kề nhau, nếu chúng chưa đứng đúng thứ tự thì đổi chỗ. Có thể tiến hành từ trên xuống (bên trái sang) hoặc từ dưới lên (bên phải sang). Nó sử dụng phép so sánh các phần tử nên là một giải thuật sắp xếp kiểu so sánh.

Được phát triển vào những năm 1950, thuật toán này đã trở thành một phần quan trọng trong việc giảng dạy các thuật toán sắp xếp cơ bản trong lĩnh vực khoa học máy tính.

### Ý tưởng

Bubble Sort là thuật toán hoán vị nhiều lần các phần tử liền kề nếu chúng sai thứ tự (thứ tự này có thể là tăng dần hoặc giảm dần).

### Thuật toán

Sắp xếp tăng dần mảng có n phần tử sau: (xét n = 5)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 7 | 3 | 4 | 2 | 5 |

**Bước 1:** Duyệt mảng

Duyệt từ phần tử đầu tiên (i = 0) đến phần tử (n – 2), sau mỗi vòng chạy cập nhật lại giá trị của n

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 7 | 3 | 4 | 2 | 5 |

**Bước 2:** So sánh và hoán vị

So sánh phần tử thứ i và i + 1, nếu giá trị của phần tử thứ i + 1 < giá trị phần tử thứ i thì thực hiện hoán vị (swap) hai phần tử này.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 7 | 3 | 4 | 2 | 5 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 3 | 7 | 4 | 2 | 5 |

**Bước 3:** Lặp lại bước 1 và bước 2 cho đến khi thu được mảng tăng dần

Vòng chạy thứ nhất: (n = 5)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 7 | 3 | 4 | 2 | 5 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 3 | 7 | 4 | 2 | 5 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 3 | 4 | 7 | 2 | 5 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 3 | 4 | 2 | 7 | 5 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 3 | 4 | 2 | 5 | 7 |

Vòng chạy thứ hai: (n = 4)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 3 | 4 | 2 | 5 | 7 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 3 | 4 | 2 | 5 | 7 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 3 | 2 | 4 | 5 | 7 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 3 | 2 | 4 | 5 | 7 |

Vòng chạy thứ ba: (n = 3)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 3 | 2 | 4 | 5 | 7 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2 | 3 | 4 | 5 | 7 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2 | 3 | 4 | 5 | 7 |

Vòng chạy thứ tư: (n = 2)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2 | 3 | 4 | 5 | 7 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2 | 3 | 4 | 5 | 7 |

### Độ phức tạp

Độ phức tạp thời gian:

|  |  |  |  |
| --- | --- | --- | --- |
| Trường hợp | Tốt nhất | Trung bình | Xấu nhất |
| Độ phức tạp | O(n2) | O(n2) | O(n2) |

Độ phức tạp không gian: O(1)

## Heap Sort

### Nguồn gốc

Heap Sort được đề xuất bởi J.W.J.Williams năm 1981, thuật toán không những đóng góp một phương pháp sắp xếp hiệu quả mà còn xây dựng một cấu trúc dữ liệu quan trọng để biểu diễn hàng đợi có độ ưu tiên: Cấu trúc dữ liệu Heap.

### Cấu trúc Heap

Cấu trúc Heap: Heap là tập hợp n phần tử (a0, a1, ... an-1) mà mỗi phần tử (ở vị trí **i**) ở nửa phần đầu của tập hợp có giá trị lớn hơn hoặc bằng (Max heap) hoặc nhỏ hơn hoặc bằng (Min heap) phần tử ở vị trí **2i + 1** và **2i +2**.

### Ý tưởng

Sử dụng cấu trúc Heap để sắp xếp mảng tăng dần hoặc giảm dần.

Về ý tưởng thì Heap Sort khá giống Selection Sort là đều tìm gọi cực trị và đưa lên đầu nhưng Heap Sort nhanh hơn vì nó lưu lại lịch sử vị trí của các giá trị còn lại.

### Thuật toán

Sắp xếp tăng dần mảng gồm n phần tử sau: (xét n = 6)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 8 | 4 | 7 | 1 | 3 | 5 |

0 1 2 3 4 5 6

**Bước 1:** 8 được đổi chỗ bằng 5.

**Bước 2:** 8 bị ngắt kết nối khỏi heap vì 8 hiện đang ở đúng vị trí.

**Bước 3:** Max-heap được tạo và 7 được hoán đổi bằng 3.

**Bước 4:** 7 bị ngắt kết nối khỏi heap.

**Bước 5:** Vùng heap tối đa được tạo và 5 được hoán đổi bằng 1.

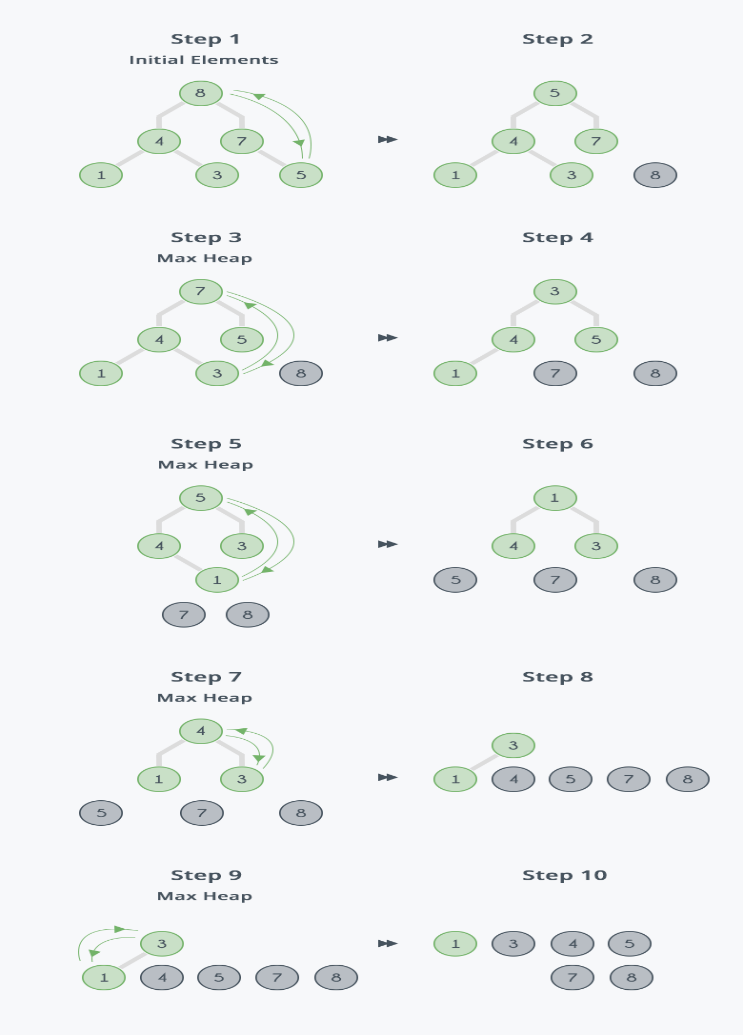
**Bước 6:** 5 bị ngắt kết nối khỏi heap.

**Bước 7:** Vùng heap tối đa được tạo và 4 được hoán đổi bằng 3.

**Bước 8:** 4 bị ngắt kết nối khỏi heap.

**Bước 9:** Vùng heap tối đa được tạo và 3 được hoán đổi bằng 1.

**Bước 10:** 3 bị ngắt kết nối.



Nguồn: [https://www.hackerearth.com/practice/algorithms/Sorting/heap-Sort/tutorial/](https://www.hackerearth.com/practice/algorithms/sorting/heap-sort/tutorial/)

Cuối cùng, ta thu được mảng đã được sắp xếp sau:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 3 | 4 | 5 | 7 | 8 |

0 1 2 3 4 5 6

### Độ phức tạp

Độ phức tạp thời gian:

|  |  |  |  |
| --- | --- | --- | --- |
| Trường hợp | Tốt nhất | Trung bình | Xấu nhất |
| Độ phức tạp | O(nlog n) | O(nlog n) | O(nlog n) |

Độ phức tạp không gian: O(1)

## Merge Sort

### Nguồn gốc

Merge Sort là một thuật toán sắp xếp đệ quy được phát minh bởi nhà khoa học máy tính John von Neumann vào năm 1945. Thuật toán merge Sort được xây dựng trên nguyên tắc "chia để trị" (divide and conquer) và được thiết kế để giải quyết các vấn đề sắp xếp dữ liệu trên máy tính.

Ban đầu, thuật toán Merge Sort được sử dụng để giải quyết vấn đề sắp xếp các bộ dữ liệu trên đĩa từ trong các máy tính đầu tiên. Sau đó, nó được ứng dụng rộng rãi trong các lĩnh vực khác như kinh doanh, khoa học, công nghệ và các lĩnh vực khác.

Vì tính hiệu quả của nó, Merge Sort là một trong những thuật toán sắp xếp phổ biến nhất trong lĩnh vực khoa học máy tính và được sử dụng trong nhiều ứng dụng khác nhau.

### Ý tưởng

Ý tưởng chính của thuật toán Merge Sort là chia một danh sách cần sắp xếp thành các danh sách con nhỏ hơn, rồi sắp xếp từng danh sách con đó và sau đó trộn các danh sách con đã sắp xếp lại với nhau cho đến khi nhận được danh sách ban đầu đã sắp xếp.

Thuật toán Merge Sort được thực hiện theo các bước sau:

+, Chia danh sách cần sắp xếp thành các danh sách con độc lập. Điều này được thực hiện bằng cách chia đôi danh sách ban đầu và tiếp tục chia đôi cho đến khi chỉ còn một phần tử trong mỗi danh sách con.

+, Sắp xếp các danh sách con này bằng cách sử dụng đệ quy. Tức là tiếp tục chia danh sách con đó thành các danh sách con nhỏ hơn và sắp xếp chúng cho đến khi chỉ còn một phần tử trong mỗi danh sách con.

+, Trộn các danh sách con đã sắp xếp lại với nhau để tạo ra danh sách lớn hơn, được sắp xếp đúng thứ tự. Quá trình trộn này được thực hiện bằng cách so sánh phần tử đầu tiên của mỗi danh sách con và chọn phần tử nhỏ hơn để đưa vào danh sách kết quả. Tiếp tục thực hiện cho đến khi một trong hai danh sách con đã được trộn hết.

+, Trả về danh sách đã được sắp xếp.

### Thuật toán

A diagram of numbers and arrows

Description automatically generated

Có 1 mảng chưa có thứ tự với 7 phần tử

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 38 | 27 | 43 | 3 | 9 | 82 | 10 |

**Bước 1:** Chia mảng làm 2 bằng cách N/2 = 7/2=3. Mảng thứ nhất có các phần tử từ 0->3, mảng thứ hai là còn lại (dòng 2 trong hình minh họa)

38 | 27 | 43 | 3 và 9 | 82 | 10

**Bước 2:** Tiếp tục chia cho tới khi mảng con chỉ còn 1 phần tử hoặc không chia được nữa. Ta được các mảng con như sau (dòng 4 trong hình minh họa)

38 27 43 3 9 82 10

**Bước 3:** Sắp xếp các phần tử của mảng bằng cách so sánh (dòng 5, 6 hình minh họa)

27 | 38 3|43 9 |82 10

3|27|38|43 9|10|82

**Bước 4:** Trộn lại để thành mảng có thứ tự (dòng 7 hình mình họa)

3|9|10|27|38|43|82

### Độ phức tạp

Thời gian :

|  |  |  |  |
| --- | --- | --- | --- |
| Trường hợp | Tốt nhất | Trung bình | Xấu nhất |
| Độ phức tạp | O(nlogn) | O(nlogn) | O(nlogn) |

Không gian bộ nhớ sử dụng: O(n)

## Quick Sort

### Nguồn gốc

Quick Sort là một thuật toán sắp xếp được phát minh bởi nhà khoa học máy tính người Anh Sir Charles Antony Richard Hoare vào năm 1959 khi ông đang làm việc tại câu lạc bộ Turing ở Luân Đôn.

Thuật toán này được coi là một trong những thuật toán sắp xếp nhanh nhất và được sử dụng rộng rãi trong thực tế

### Ý tưởng

Ý tưởng chính của thuật toán Quick Sort là dựa trên thuật toán Chia để trị chọn một phần tử làm trục (pivot) và phân vùng mảng đã cho xung quanh trục đã chọn bằng cách đặt trục vào đúng vị trí của nó trong mảng được sắp xếp.

Thuật toán Quick Sort hoạt động bằng cách chọn một phần tử trong dãy đầu vào làm "pivot". Sau đó, các phần tử trong dãy đầu vào được phân thành hai phần: các phần tử nhỏ hơn hoặc bằng pivot được đưa về phía trước của pivot và các phần tử lớn hơn pivot được đưa về phía sau của pivot. Sau khi phân hoạch này được thực hiện, pivot sẽ được đặt vào giữa hai phần được phân chia.

Sau đó, thuật toán sẽ đệ quy áp dụng phương pháp trên cho các phần tử nằm bên trái và phải của pivot cho đến khi dãy được sắp xếp hoàn toàn. Quá trình đệ quy sẽ kết thúc khi chỉ còn một phần tử hoặc không còn phần tử nào để phân chia.

Với mỗi đệ quy, thuật toán Quick Sort sẽ có thể chọn một pivot khác nhau. Có nhiều cách để chọn pivot, một cách đơn giản là chọn phần tử ở giữa dãy đầu vào. Tuy nhiên, sử dụng cách chọn pivot khác nhau sẽ ảnh hưởng đến hiệu suất của thuật toán Quick Sort

### Thuật toán

Thuật toán Quick Sort bao gồm các bước sau:

**Bước 1:** Chọn mốc (pivot) là một số ngẫu nhiên của mảng (thường là số cuối, số đầu, hoặc số trung vị).

**Bước 2:** Phân hoạch mảng thành hai mảng con “ảo” (nghĩa là một phần của mảng ban đầu mà không phải là khởi tạo mảng mới), mảng con bên trái bao gồm những phần tử nhỏ hơn/lớn hơn pivot, mảng con bên phải bao gồm những phần tử lớn hơn/nhỏ hơn pivot.

**Bước 3:** Dùng đệ quy lặp lại **Bước 1** và **Bước 2** bằng cùng một phương pháp chọn pivot và phân hoạch (nghĩa là không được thay đổi cách chọn pivot hay cách phân hoạch) với các mảng con từ việc phân hoạch. Kết thúc đệ quy khi chỉ còn một phần tử ở mỗi mảng con, lúc này mảng đã được sắp xếp.

Ví dụ: Cho mảng có 7 phần tử “3, 8, 1, 5, 4, 2, 7” và yêu cầu sắp xếp chúng theo thứ tự tăng dần.

**Bước 1:** Chọn pivot là số trung vị: 6

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3 | 8 | 1 | 5 | 4 | 2 | 7 |

**Bước 2:** Phân hoạch mảng thành hai phần, phần bên trái nhỏ hơn pivot và phần bên phải lớn hơn pivot.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3 | 2 | 1 | 4 | 5 | 8 | 7 |

**Bước 3:** Dùng đệ quy lặp lại **Bước 1** và **Bước 2:**

* Lần đệ quy thứ nhất:
* Phần bên trái 5:

**Bước 1:** Chọn pivot là 2

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3 | 2 | 1 | 4 | 5 | 8 | 7 |

**Bước 2:** Phân hoạch mảng thành hai phần:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 8 | 7 |

* Phần bên phải 5:

**Bước 1:** Chọn pivot là 8

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 8 | 7 |

**Bước 2:** Phân hoạch mảng thành hai phần:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 7 | 8 |

* Lần đệ quy thứ hai:
* Phần bên trái 2:

**Bước 1:** Chọn pivot là 1.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 7 | 8 |

**Bước 2:** Phân hoạch mảng thành hai phần: Không thành công.

* Phần bên phải 2:

**Bước 1:** Chọn pivot là 3.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 7 | 8 |

**Bước 2:** Phân hoạch mảng thành hai thành phần:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 7 | 8 |

* Phần bên trái 8:

**Bước 1:** Chọn pivot là 7.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 7 | 8 |

**Bước 2:** Phân hoạch mảng thành hai thành phần: Không thành công.

* Phần bên phải 8: Không tồn tại.
* Lần đệ quy thứ ba:
* Phần bên trái 1: Không tồn tại.
* Phần bên phải 1: Không tồn tại.
* Phần bên trái 3: Không tồn tại
* Phần bên phải 3:

**Bước 1:** Chọn pivot là 4.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 7 | 8 |

**Bước 2:** Phân hoạch mảng thành hai thành phần: Không thành công.

* Phần bên trái 7: Không tồn tại.
* Phần bên phải 7: Không tồn tại.

Dừng đệ quy, kết quả là mảng đã được sắp xếp:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 7 | 8 |

Thuật toán Quick Sort còn được hiểu là quá trình đệ quy của việc phân hoạch mảng ban đầu, vì thế, dựa vào các cách phân hoạch khác nhau ta có những thuật toán Quick Sort khác nhau.

Có hai cách phân hoạch phổ biến:

+, Cách 1 được phát minh bởi nhà khoa học tên Lomuto, vì thế nó được gọi là thuật toán Lomuto Partition (Phân hoạch kiểu Lomuto):

+, Cách 2 được phát minh bởi nhà khoa học tên C.A.R. Hoare, vì thế nó được gọi là thuật toán Hoare Partition (Phân hoạch kiểu Hoare)

Chi tiết thuật toán:

Lomuto Partition: (Chọn pivot là phần tử cuối để làm mẫu)

**Bước 1:** Khởi tạo biến, left = 0 là biên trái mảng, right = size - 2 là biên phải mảng, i = left – 1, j = 0 (không tính pivot)

**Bước 2:** Tạo vòng lặp cho j chạy từ biên trái đến biên phải và thực hiện chương trình sau:

Nếu phần tử tại vị trí thứ j nhỏ hơn pivot, tráo đổi phần tử ở vị trí thứ j và vị trí thứ i + 1. Sau đó tăng i lên 1.

Nếu phần tử thứ j lớn hơn pivot, không làm gì, kết thúc lần lặp.

**Bước 3:** Sau khi kết thúc vòng lặp, tráo đổi pivot với phần tử ở vị trí thứ i + 1. Mảng nhận về là mảng đã phân hoạch.

Ví dụ: Phân hoạch mảng sau: a = {3, 8, 1, 5, 4, 2, 7} với pivot là 7.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3 | 8 | 1 | 5 | 4 | 2 | 7 |

Bước 1:

i = -1

j = 0

left = 0

right = 5

Bước 2:

* j=0: a[0] = 3 < pivot:

Tráo đổi phần tử a[0] và phần tử a[i + 1]=a[0]

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3 | 8 | 1 | 5 | 4 | 2 | 7 |

Tăng i lên 0.

* j = 1: a[1] = 8 > pivot: Không làm gì
* j = 2: a[2] = 1 < pivot:

Tráo đổi phần tử a[2] và a[i + 1] = a[1]

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3 | 1 | 8 | 5 | 4 | 2 | 7 |

Tăng i lên 1.

* j = 3: a[3] = 5 < pivot:

Tráo đổi phần tử a[3] và a[i +1] = a[2]:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3 | 1 | 5 | 8 | 4 | 2 | 7 |

Tăng i lên 2.

* j = 4: a[4] = 4 <pivot:

Tráo đổi phần tử a[4] và a[i + 1] = a[3]:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3 | 1 | 5 | 4 | 8 | 2 | 7 |

Tăng i lên 3.

* j = 5: a[5] = 2 < pivot:

Tráo đổi phần tử a[5] và a[i + 1] = a[4]:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3 | 1 | 5 | 4 | 2 | 8 | 7 |

Tăng i lên 4.

Kết thúc vòng lặp.

**Bước 3:** Hoán đổi vị trí pivot và a[i + 1] = a[5]

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3 | 1 | 5 | 4 | 2 | 7 | 8 |

Ta được mảng đã phân hoạch với pivot là 7.

Hoare Partition: (Chọn phần tử bên trái làm pivot)

**Bước 1 :** Khởi tạo biến pivot = 0, i = pivot – 1, right = size – 1, j = right.

**Bước 2 :** Tạo vòng lặp cho i chạy từ biên trái cho đến khi gặp được phần tử lớn hơn hoặc bằng pivot thì dừng, sau đó j chạy từ biến phải cho đến khi gặp được phần tử nhỏ hơn pivot thì dừng. Tráo đổi hai phần tử này và tiếp tục chạy cặp i - j. Điều kiện dừng là i = j.

**Bước 3 :** Sau khi kết thúc vòng lặp, tráo đổi pivot với phần tử ở vị trí i. Mảng nhận về là mảng đã phân hoạch.

**Ví dụ :** Phân hoạch mảng sau: a = {3, 8, 1, 5, 4, 2, 7} với pivot là 3.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3 | 8 | 1 | 5 | 4 | 2 | 7 |

Bước 1:

i = -1

right = 6

j = 6

Bước 2:

* i = 0: a[0] = 3 ≥ 3: Chạy j:
* j = 6: a[j] = 7 > 3: Giảm j:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3 | 8 | 1 | 5 | 4 | 2 | 7 |

* j = 5: a[j] = 2 < 3: Tráo đổi a[i] và a[j]:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 2 | 8 | 1 | 5 | 4 | 3 | 7 |

* i = 1: a[1] = 8 > 3: Chạy j:
* j = 5: a[j] = 3: Giảm j:
* j = 4: a[j] = 4 > 3: Giảm j:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 2 | 8 | 1 | 5 | 4 | 3 | 7 |

* j = 3: a[j] = 5 >3 : Giảm j:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 2 | 8 | 1 | 5 | 4 | 3 | 7 |

* j = 2: a[j] = 1 < 3: Tráo đổi a[1] và a[2]:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 2 | 1 | 8 | 5 | 4 | 3 | 7 |

* i = 2: i = j, dừng vòng lặp.

**Bước 3:** Tráo dổi a[i] và pivot. Ta được mảng:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 2 | 1 | 3 | 5 | 4 | 8 | 7 |

Mảng trên là mảng đã được phân hoạch

### Độ phức tạp

|  |  |  |  |
| --- | --- | --- | --- |
| Trường hợp | Tốt nhất | Trung bình | Xấu nhất |
| Độ phức tạp | O(nlogn) | O(nlogn) | O(n2) |

## Shaker Sort

### Ý tưởng

Thuật toán Shaker Sort là một phiên bản nâng cấp của thuật toán Bubble Sort

Tóm tắt ý tưởng: Thuật toán rung lắc (Shaker Sort) được thiết kế để lặp lại việc đưa một cực trị (cực đại/cực tiểu) của dãy nổi lên đầu dãy, đồng thời đưa cực trị đối nghịch (cực tiểu/cực đại) chìm xuống cuối dãy bằng cách lặp lại việc duyệt và so sánh các cặp phần từ kế nhau từ trái sang phải và từ phải sang trái. Cụ thể:

**Bước 1:** Duyệt dãy số từ trái sang phải để dìm phần tử cực trị (cực đại/cực tiểu) về cuối dãy số và duyệt dãy số từ phải sang trái để nâng phần tử cực trị đối nghịch (cực tiểu/cực đại) lên đầu dãy số.

**Bước 2:** Lặp lại **Bước 1** với điều kiện dừng là khi count = 0 (xét điều kiện này mỗi khi kết thúc một vòng lặp (hai lần duyệt)).

**Bước 3:** Trả về dãy cuối cùng. Dãy của lần lặp cuối cùng chính là dãy đã được sắp xếp.

### Thuật toán

Giả sử chúng ta có một dãy ngẫu nhiên S có 7 phần tử “3, 8, 1, 5, 4, 2, 7” và yêu cầu sắp xếp chúng theo thứ tự tăng dần.

Theo thuật toán Shaker Sort, ta cần lặp lại việc đưa cực tiểu của dãy số nổi lên đầu dãy số và đưa cực đại chìm xuống cuối dãy số bằng cách lặp lại việc duyệt dãy số từ trái sang phải và từ phải sang trái kết hợp với so sánh các cặp phần tử kế cận.

**Bước 1:** Duyệt dãy số từ trái sang phải để dìm phần tử cực đại về cuối dãy số và duyệt dãy số từ phải sang trái để nâng phần tử cực tiểu lên đầu dãy số. Đồng thời khởi tạo biến count để đếm số lần đổi chỗ trong khi duyệt, đồng thời khởi tạo biến count = 0 để đếm số lần đổi chỗ của dãy số.

1. Dìm phần tử cực đại về cuối dãy số.

Khởi tạo i = 0, duyệt và so sánh các cặp phần tử kế cận i và i + 1, nếu i > i + 1 thì đổi chỗ chúng:

* i = 0, i + 1 = 1:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3 | 8 | 1 | 5 | 4 | 2 | 7 |

3 < 8, nghĩa là S[0] < S[1]: không thực hiện thao tác đổi chỗ, tăng i.

* i = 1, i + 1 = 2:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3 | 8 | 1 | 5 | 4 | 2 | 7 |

8 > 1, nghĩa là S[1] > S[2]: đổi chỗ hai phần tử này, tăng i, tăng count, ta được dãy số:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3 | 1 | 8 | 5 | 4 | 2 | 7 |

* i = 2, i + 1 = 3:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3 | 1 | 8 | 5 | 4 | 2 | 7 |

8 > 5, nghĩa là S[2] > S[3]: đổi chỗ hai phần tử này, tăng i, tăng count, ta được dãy số:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3 | 1 | 5 | 8 | 4 | 2 | 7 |

* i = 3, i + 1 = 4:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3 | 1 | 5 | 8 | 4 | 2 | 7 |

8 > 4, nghĩa là S[3] > S[4]: đổi chỗ hai phần tử này, tăng i, tăng count, ta được dãy số:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3 | 1 | 5 | 4 | 8 | 2 | 7 |

* i = 4, i + 1 = 5:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3 | 1 | 5 | 4 | 8 | 2 | 7 |

8 > 2, nghĩa là S[4] > S[5]: đổi chỗ hai phần tử này, tăng i, tăng count, ta được dãy số:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3 | 1 | 5 | 4 | 2 | 8 | 7 |

* i = 5, i + 1 = 6:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3 | 1 | 5 | 4 | 2 | 8 | 7 |

8 > 7, nghĩa là S[5] > S[6]: đổi chỗ hai phần tử này, tăng i, tăng count, ta được dãy số:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3 | 1 | 5 | 4 | 2 | 7 | 8 |

Với lần lặp đầu tiên, lần duyệt thứ nhất, ta đã dìm được phần tử 8 (cực đại) chìm xuống cuối dãy số và count = 5.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3 | 1 | 5 | 4 | 2 | 7 | 8 |

1. Đưa phần tử cực tiểu nâng lên đầu dãy số.

Khởi tạo i = 6, duyệt và so sánh các cặp phần tử kế cận i và i - 1, nếu i < i - 1 thì đổi chỗ chúng:

* i = 6, i - 1 = 5:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3 | 1 | 5 | 4 | 2 | 7 | 8 |

8 > 7, nghĩa là S[6] > S[5]: không thực hiện thao tác đổi chỗ, giảm i.

* i = 5, i - 1 = 4:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3 | 1 | 5 | 4 | 2 | 7 | 8 |

7 > 2, nghĩa là S[5] > S[4]: không thực hiện thao tác đổi chỗ, giảm i.

* i = 4, i - 1 = 3:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3 | 1 | 5 | 4 | 2 | 7 | 8 |

2 < 4, nghĩa là S[4] < S[3]: đổi chỗ hai phần tử này, giảm i, tăng count, ta được dãy số:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3 | 1 | 5 | 2 | 4 | 7 | 8 |

* i = 3, i - 1 = 2:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3 | 1 | 5 | 2 | 4 | 7 | 8 |

2 < 5, nghĩa là S[3] < S[2]: đổi chỗ hai phần tử này, giảm i, tăng count, ta được dãy số:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3 | 1 | 2 | 5 | 4 | 7 | 8 |

* i = 2, i - 1 = 1:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3 | 1 | 2 | 5 | 4 | 7 | 8 |

2 > 1, nghĩa là S[2] > S[1]: không thực hiện thao tác đổi chỗ, giảm i.

* i = 1, i - 1 = 0:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3 | 1 | 2 | 5 | 4 | 7 | 8 |

1 < 3, nghĩa là S[1] < S[0]: đổi chỗ hai phần tử này, giảm i, tăng count, ta được dãy số:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 | 2 | 5 | 4 | 7 | 8 |

Với lần lặp đầu tiên, lần duyệt thứ hai, ta đã dìm được phần tử 1 (cực tiểu) lên đầu dãy và count = 8.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 | 2 | 5 | 4 | 7 | 8 |

**Bước 2:** Lặp lại **Bước 1** với điều kiện dừng là khi count = 0 (xét điều kiện này mỗi khi kết thúc một vòng lặp (hai lần duyệt)):

Sau khi thực hiện các vòng lặp, ta lần lượt nhận được các dãy sau:

* Với lần lặp thứ hai, lần duyệt thứ nhất: count = 2:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 7 | 8 |

* Với lần lặp thứ hai, lần duyệt thứ hai: count = 2:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 7 | 8 |

Tiếp tục lần lặp thứ ba:

* Với lần lặp thứ ba, lần duyệt thứ nhất: count = 0:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 7 | 8 |

* Với lần lặp thứ ba, lần duyệt thứ hai: count = 0:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 7 | 8 |

Vì count = 0, dừng vòng lặp.

**Bước 3:** Trả về dãy cuối cùng:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 7 | 8 |

Thật vậy, dãy đã được sắp xếp theo thứ tự tăng dần đúng theo yêu cầu bài toán.

Đề xuất cải tiến: Với mỗi lần duyệt, ta đã đưa được một cực trị về cuối dãy hoặc đầu dãy, vì thế, phần tử đó sẽ không bao giờ bị đổi chỗ nữa. Vậy nên, với mỗi lần duyệt, chúng ta có thể giảm phạm vi trái/phải xuống để giảm số lần thực hiện vòng lặp, thuật toán được tối ưu hơn.

### Độ phức tạp

|  |  |  |  |
| --- | --- | --- | --- |
| Trường hợp | Tốt nhất | Trung bình | Xấu nhất |
| Độ phức tạp | O(n) | O(n2) | O(n2) |

Giải thích:

* **Best case:** Dãy đã thỏa điều kiện sắp xếp ngay từ vòng lặp đầu tiên, độ phức tạp là n - 1.
* **Worst case = Average case:** Số lần duyệt từ trái sang phải là n – 1, số lần duyệt từ phải sang trái là n – 1.

## Shell Sort

### Ý tưởng

Cho mảng gồm n phần tử, xét lần lượt hai phần tử cách nhau một khoảng a (a = n/2 hoặc n/4...), so sánh và đổi chỗ (nếu sai thứ tự); liên tục giảm khoảng cách giữa chúng cho đến khi sắp xếp xong.

Shell Sort là thuật toán tổng quát của thuật toán Insertion Sort.

### Thuật toán

Sắp xếp tăng dần mảng có n phần tử sau: (xét n = 8)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 23 | 38 | 12 | 4 | 9 | 5 | 3 | 1 |

**Bước 1:** Với n = 8, ta xét lần lượt hai phần tử cách nhau một khoảng bằng 4 (= n/2)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 23 | 38 | 12 | 4 | 9 | 5 | 3 | 1 |

Đầu tiên, phần tử thứ 0 được so sánh với phần tử thứ 4.

Nếu phần tử thứ 0 lớn hơn phần tử thứ 4, ta sẽ đổi chỗ hai phần tử này như sau: lưu giá trị của phần tử thứ 0 vào biến temp, giá trị của phần tử thứ 0 bằng giá trị phần tử thứ 4, giá trị phần tử thứ 4 bằng giá trị biến temp.

Ở ví dụ này, ta thấy 23 > 9 nên ta đổi chỗ hai phần tử này.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 9 | 38 | 12 | 4 | 23 | 5 | 3 | 1 |

Tiếp tục quá trình này với các phần tử còn lại.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 9 | 38 | 12 | 4 | 23 | 5 | 3 | 1 |
| 9 | 5 | 12 | 4 | 23 | 38 | 3 | 1 |
| 9 | 5 | 3 | 4 | 23 | 38 | 12 | 1 |
| 9 | 5 | 3 | 1 | 23 | 38 | 12 | 4 |

**Bước 2:** Giảm khoảng cách giữa hai phần tử ta xét, lúc này ta sẽ xét lần lượt hai phần tử cách nhau một khoảng bằng 2 (= n/4)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 9 | 5 | 3 | 1 | 23 | 38 | 12 | 4 |
| 3 | 5 | 9 | 1 | 23 | 38 | 12 | 4 |

Lúc này tất cả các cặp phần tử cách nhau một khoảng bằng 2 đều được so sánh với nhau.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 3 | 1 | 9 | 5 | 23 | 38 | 12 | 4 |
| 3 | 1 | 9 | 5 | 23 | 38 | 12 | 4 |

Tiếp tục quá trình này với các phần tử còn lại.

Kết quả sau khi chạy xong vòng 2:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 3 | 1 | 9 | 4 | 12 | 5 | 23 | 38 |

**Bước 3:** Giảm khoảng cách giữa hai phần tử ta xét một lần nữa, lúc này ta sẽ xét lần lượt hai phần tử cách nhau một khoảng bằng 1 (= n/8) và cũng là vòng chạy cuối cùng.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 3 | 1 | 9 | 4 | 12 | 5 | 23 | 38 |
| 1 | 3 | 9 | 4 | 12 | 5 | 23 | 38 |
| 1 | 3 | 9 | 4 | 12 | 5 | 23 | 38 |
| 1 | 3 | 4 | 9 | 12 | 5 | 23 | 38 |
| 1 | 3 | 4 | 9 | 12 | 5 | 23 | 38 |
| 1 | 3 | 4 | 5 | 9 | 12 | 23 | 38 |
| 1 | 3 | 4 | 5 | 9 | 12 | 23 | 38 |
| 1 | 3 | 4 | 5 | 9 | 12 | 23 | 38 |

Cuối cùng, ta thu được mảng đã được sắp xếp tăng dần.

### Độ phức tạp

Độ phức tạp thời gian:

|  |  |  |  |
| --- | --- | --- | --- |
| Trường hợp | Tốt nhất | Trung bình | Xấu nhất |
| Độ phức tạp | O(nlog n) | O(nlog n) | O(n2) |

Độ phức tạp không gian: O(1)

## Counting Sort

### Ý tưởng

Counting Sort là một thuật toán sắp xếp các số nguyên dương bằng cách đếm số lượng

các phần tử có cùng một giá trị khoá trong mảng hoặc danh sách. Độ phức tạp thời gian

phụ thuộc vào số lượng phần tử và chênh lệch giữa khoá lớn nhất và nhỏ nhất, do đó thường chỉ phù hợp khi biến thiên giá trị khoá không lớn hơn số lượng phần tử.

### Thuật toán

Xem xét dữ liệu trong phạm vi từ 0 đến 9.

Input:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 0 | 5 | 0 | 7 | 2 | 1 |

**Bước 1:** Tạo một mảng đếm với giá trị ban đầu là 0, để lưu lại số lần xuất hiện của các giá trị mà mảng cần xét. Sau đó, đếm số lần xuất hiện của các phần tử và lưu vào mảng

CountingArray:

Index: 0 1 2 3 4 5 6 7 8 9

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2 | 2 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |

**Bước 2:** Tính tổng tích lũy của mảng đếm. Bằng cách giá trị của phần tử sau bằng tổng của nó và phần tử trước nó.

Index: 0 1 2 3 4 5 6 7 8 9

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2 | 2 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |

|\_\_2+2\_\_\_|

Index: 0 1 2 3 4 5 6 7 8 9

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2 | 4 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |

Index: 0 1 2 3 4 5 6 7 8 9

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2 | 4 | 5 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |

Index: 0 1 2 3 4 5 6 7 8 9

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2 | 4 | 5 | 5 | 0 | 1 | 0 | 1 | 0 | 0 |

…

Làm tương tự ta được kết quả như sau:

Index: 0 1 2 3 4 5 6 7 8 9

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2 | 4 | 5 | 5 | 5 | 6 | 6 | 7 | 7 | 7 |

Mảng đếm đã sửa đổi cho biết vị trí của từng đối tượng trong trình tự đầu ra.

**Bước 3:** Duyệt mảng input và đặt các phần tử vào đúng vị trí ở Output, đồng thời giảm số lượng của nó ở mảng CountingArray đi 1.

+, Bước 3.1

Input:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 0 | 5 | 0 | 7 | 2 | 1 |

CountingArray:

Index: 0 1 2 3 4 5 6 7 8 9

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2 | 4 | 5 | 5 | 5 | 6 | 6 | 7 | 7 | 7 |

Output: 1 2 3 4 5 6 7

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  | 1 |  |  |  |

Cập nhật lại CountingArray:

CountingArray:

Index: 0 1 2 3 4 5 6 7 8 9

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2 | 3 | 5 | 5 | 5 | 6 | 6 | 7 | 7 | 7 |

+, Bước 3.2

Input:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 0 | 5 | 0 | 7 | 2 | 1 |

CountingArray:

Index: 0 1 2 3 4 5 6 7 8 9

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2 | 3 | 5 | 5 | 5 | 6 | 6 | 7 | 7 | 7 |

Output: 1 2 3 4 5 6 7

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 0 |  | 1 |  |  |  |

Cập nhật lại CountingArray:

CountingArray:

Index: 0 1 2 3 4 5 6 7 8 9

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 | 5 | 5 | 5 | 6 | 6 | 7 | 7 | 7 |

+, Bước 3.3

Input:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 0 | 5 | 0 | 7 | 2 | 1 |

CountingArray:

Index: 0 1 2 3 4 5 6 7 8 9

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 | 5 | 5 | 5 | 6 | 6 | 7 | 7 | 7 |

Output: 1 2 3 4 5 6 7

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 0 |  | 1 |  | 5 |  |

Cập nhật lại CountingArray:

CountingArray:

Index: 0 1 2 3 4 5 6 7 8 9

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 | 5 | 5 | 5 | 5 | 6 | 7 | 7 | 7 |

+, Bước 3.4

Index:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 0 | 5 | 0 | 7 | 2 | 1 |

CountingArray:

Index: 0 1 2 3 4 5 6 7 8 9

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 | 5 | 5 | 5 | 5 | 6 | 7 | 7 | 7 |

Output: 1 2 3 4 5 6 7

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 |  | 1 |  |  |  |

Cập nhật lại CountingArray:

CountingArray:

Index: 0 1 2 3 4 5 6 7 8 9

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 3 | 5 | 5 | 5 | 6 | 6 | 7 | 7 | 7 |

...

Làm tương tự ta được Output như sau:

Output: 1 2 3 4 5 6 7

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 1 | 1 | 2 | 5 | 7 |

Mảng đã được sắp xếp

### Độ phức tạp

Độ phức tạp về thời gian:

|  |  |  |  |
| --- | --- | --- | --- |
| Trường hợp | Tốt nhất | Trung bình | Xấu nhất |
| Độ phức tạp | O(n+k) | O(n+k) | O(n+k) |

với n là số phần tử trong mảng đầu vào và k là phạm vi đầu vào.

Không gian phụ trợ: O (n + k)

9.4. Cải tiến

Thay vì tạo mảng CountingArray với 10 phần tử từ 0 đến 9, ta có thể tạo mảng ít phần tử hơn, tiết kiệm bộ nhớ, hạn chế các phép gán và so sánh hơn đối với trường hợp dự liệu đầu vào với số lượng phần tử rất lớn.

Bằng cách, tìm phần tử lớn nhất trong mảng input, tạo mảng mới với [giá trị phần tử lớn nhất +1] phần. Như vậy, thay vì chúng ta đối chiếu đưa giá trị vào đúng vị trí rồi mới cập nhật giá trị của CountingArray, thì bây giờ chúng ta làm ngược lại ở giai đoạn này. Ví dụ:

Input:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 0 | 5 | 0 | 7 | 2 | 1 |

maxValue = 7

CountingArray: (coi như đã qua bước tính tổng tích lũy)

Index: 0 1 2 3 4 5 6 7

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 2 | 4 | 5 | 5 | 5 | 6 | 6 | 7 |

4 - 1 = 3

Output: 1 2 3 4 5 6 7

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | 1 |  |  |  |  |

Làm tương tự.

Trường hợp khác, chúng ta có thể không sử dụng mảng phụ để lưu kết quả mà lưu trực tiếp lên mảng input. Vì vậy, độ phức tạp của thuật toán về thời gian là O(1). Tuy nhiên, cách này không được dùng phổ biến vì độ phức tạp không gian cũng phụ thuộc vào giá trị của phạm vi k (phạm vi đầu vào) và trong trường hợp k lớn, việc không sử dụng mảng phụ có thể không hiệu quả.

## Radix Sort

### Ý tưởng

Là thuật toán không sử dụng phép so sánh (non-comparison) làm cơ sở mà sắp xếp theo phương pháp cơ số dựa trên nguyên tắc phân loại thư bưu điện. Thuật toán sẽ xét các số là chuỗi dữ liệu, phân loại các phần tử lần lượt theo chữ số từ phải sang trái (từ hàng đơn vị đến hàng chục, trăm…) hay có thể theo thứ tự ngược lại (phụ thuộc vào dữ liệu đầu vào), vào các hộp có nhãn là các kí số thuộc cơ số mà ta đang xét.

### Thuật toán

**Bước 1:** Tìm phần tử lớn nhất để xác định chiều dài dài nhất của các phần tử, hay là số lần sắp các phần tử vào các hộp chứa.

**Bước 2:** Dựa vào dữ liệu đầu vào, ta sẽ phân thành số các hộp chứa bằng cơ số ta đang xét, và các nhãn của các hộp là các kí tự của cơ số đó.

**Bước 3:** Áp dụng thuật toán Counting Sort tối thiểu m lần để phân các phần tử vào các hộp theo tham số exp. (Với exp ={1, 10, …, 10m-1} tượng trưng cho lớp hàng đơn vị, chục, trăm…)

Xét từng phần tử của mảng A, tách chữ số ở hàng exp của phần tử ai và bỏ vô hộp có nhãn tương ứng. Sau đó, sắp xếp lại các phần tử ở các hộp vào mảng ban đầu theo đúng trình tự.

Thực hiện với ví dụ sau:

Cho mảng A:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 534 | 12 | 4182 | 77 | 3 | 102 |

**Bước 1:** Tìm phần tử lớn nhất trong mảng

* Số lớn nhất của mảng: 4182
* Số chữ số lớn nhất của mảng: m = 4

**Bước 2:** Dữ liệu đầu vào là các số thập phân nên cơ số sẽ là 10. Ta khởi tạo 10 hộp chứa rỗng với các nhãn từ 0 đến 9. (với 0 đến 9 là các kí tự của cơ số 10)

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

Bước 3:

Phân theo chữ số hàng đơn vị:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 5 | 010**2** |  |  |  |  |  |  |  |  |  |  |
| 4 | 000**3** |  |  |  |  |  |  |  |  |  |  |
| 3 | 007**7** |  |  |  |  |  |  |  |  |  |  |
| 2 | 418**2** |  |  | 102 |  |  |  |  |  |  |  |
| 1 | 001**2** |  |  | 4182 |  |  |  |  |  |  |  |
| 0 | 053**4** |  |  | 12 | 3 | 534 |  |  | 77 |  |  |
| Chỉ số | A | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

Phân theo chữ số hàng chục:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 5 | 00**7**7 |  |  |  |  |  |  |  |  |  |  |
| 4 | 05**3**4 |  |  |  |  |  |  |  |  |  |  |
| 3 | 00**0**3 |  |  |  |  |  |  |  |  |  |  |
| 2 | 01**0**2 |  |  |  |  |  |  |  |  |  |  |
| 1 | 41**8**2 | 3 |  |  |  |  |  |  |  |  |  |
| 0 | 00**1**2 | 102 | 12 |  | 534 |  |  |  | 77 | 4182 |  |
| Chỉ số | A | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

Phân theo chữ số hàng trăm:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 5 | 4**1**82 |  |  |  |  |  |  |  |  |  |  |
| 4 | 0**0**77 |  |  |  |  |  |  |  |  |  |  |
| 3 | 0**5**34 |  |  |  |  |  |  |  |  |  |  |
| 2 | 0**0**12 | 77 |  |  |  |  |  |  |  |  |  |
| 1 | 0**0**03 | 12 | 4182 |  |  |  |  |  |  |  |  |
| 0 | 0**1**02 | 3 | 102 |  |  |  | 534 |  |  |  |  |
| Chỉ số | A | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

Phân theo chữ số hàng nghìn:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 5 | **0**534 |  |  |  |  |  |  |  |  |  |  |
| 4 | **4**182 | 534 |  |  |  |  |  |  |  |  |  |
| 3 | **0**102 | 102 |  |  |  |  |  |  |  |  |  |
| 2 | **0**077 | 77 |  |  |  |  |  |  |  |  |  |
| 1 | **0**012 | 12 |  |  |  |  |  |  |  |  |  |
| 0 | **0**003 | 3 |  |  |  | 4182 |  |  |  |  |  |
| Chỉ số | A | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

Cuối cùng ta được mảng đã sắp xếp:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| i | 0 | 1 | 2 | 3 | 4 | 5 |
| A[i] | 3 | 12 | 102 | 77 | 534 | 4182 |

### Độ phức tạp

|  |  |  |  |
| --- | --- | --- | --- |
| Trường hợp | Tốt nhất | Trung bình | Xấu nhất |
| Độ phức tạp | O(d\*(n+b)) | O(d\*(n+b)) | O(d\*(n+b)) |

Độ phức tạp không gian: O(n+b)

Với d: là số lượng chữ số lớn nhất cần xét

n: số lượng phần tử của mảng

b: là hệ cơ số ta xét

## Flash Sort

### Ý tưởng

Flash Sort được phát minh bởi Karl-Dietrich Neubert, là thuật toán dựa trên phân phối (distribution-based), phân chia các phần tử thành nhiều phân lớp, sắp xếp độc lập trên từng phân lớp đó, sau cùng gộp các phân lớp đã sắp xếp lại ta sẽ được một dãy có thứ tự. Ý tưởng của Flash Sort được xem là khá tương đồng với Bucket Sort nhưng được xem là một cải tiến hơn so với Bucket Sort về mặt bộ nhớ.

Flash Sort gồm có 3 giai đoạn:

* Phân loại các phần tử (Elements Classification)
* Phân bố các phần tử vào đúng các phân lớp (Elements Permutation)
* Sắp xếp các phần tử trong từng phân lớp theo đúng thứ tự (Elements Ordering).

### Thuật toán

**Giai đoạn 1:** Phân loại các phần tử (Elements Classification)

* **Bước 1:** Tìm số lượng phân lớp (bucket) tối ưu: .
* **Bước 2:** Khởi tạo mảng Buckets với số lượng Bucket bằng để đếm số lượng của các phần tử thuộc Buckets[k] đó.
* **Bước 3:** Tìm giá trị nhỏ nhất của mảng: minVal và vị trí của phần tử có giá trị lớn nhất: maxPos.
* **Bước 4:** Xác định chỉ số bucket của phần tử ai bằng công thức
* **Bước 5:** Xác định chỉ số của phần tử cuối cùng thuộc từng Buckets[k].

**Giai đoạn 2:** Phân bố các phần tử vào đúng các phân lớp (Elements Permutation)

Phân lớp k được xem là rỗng khi giá trị của Buckets[k] là vị trí phần tử cuối cùng của phân lớp đó trong mảng.

Phân lớp k được xem là đầy khi giá trị của Buckets[k] là vị trí phần tử đầu tiên của phân lớp đó trong mảng.

Ta sẽ chạy từng phần tử trên mảng, và xét xem phần tử đó đã ở đúng phân lớp hay chưa, nếu chưa ta đem về đúng phân lớp. Và khi đem một phần tử ở đâu đó đến một vị trí nào đó thì ta phải nhấc phần tử hiện tại đang chiếm chỗ ra, và tiếp tục với phần tử bị nhấc ra và đưa đến chỗ khác cho đến khi quay lại vị trí ban đầu thì hoàn tất vòng lặp.

Với tối đa n - 1 lần di chuyển thì các phần tử sẽ ở đúng phân lớp của nó.

* **Bước 1:** Đổi chỗ a[0] với a[maxPos] vì a[maxPos] thuộc phân lớp cuối cùng .
* **Bước 2:** Khởi tạo biến *Flash* giữ giá trị của phần tử cần di chuyển, biến *i* là biến chạy trên mảng, biến *countSwap* để đếm số lần dịch chuyển.

Nếu *countSwap* chưa vượt qua n - 1

2.1. Trong khi > thì:

Dịch chuyển qua phần tử tiếp theo và xét chỉ số bucket của phần tử đó vì đã thuộc đúng phân lớp.

2.2. Flash =

Trong khi

* Xác định lại
* (vị trí cuối cùng của phần tử thuộc phân lớp ) trừ đi 1 vì đã đưa về đúng phân lớp tại vị trí
* Đổi chỗ ()
* Tăng 1 cho *countSwap*

**Giai đoạn 3:** Sắp xếp các phần tử trong từng phân lớp theo đúng thứ tự (Elements Ordering). Dùng Insertion Sort trên mỗi phân lớp (bucket) để được mảng sắp xếp theo thứ tự. Thực hiện với ví dụ sau:

Sắp xếp mảng đã cho dưới đây:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| A[i] | 55 | 134 | 123 | 98 | 83 | 23 | 7 |

Giai đoạn 1:

**Bước 1:** Xác định số phân lớp: numberOfBuckets = 7\*0.43=3

**Bước 2:** Tạo mảng Buckets với số lượng là 3 phân lớp.

**Bước 3:** Xác định minVal và maxPos: minVal = 7 và maxPos = 1

**Bước 4:** Xác định phần tử ai thuộc về Bucket nào với công thức trên. Ta được bảng sau:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| A[i] | 55 | 134 | 123 | 98 | 83 | 23 | 7 |
|  | 0 | 2 | 1 | 1 | 1 | 0 | 0 |

Tổng hợp số lượng của mỗi phân lớp ta có: Buckets[0] = 3, Buckets[1] = 3, Buckets[2] = 1

**Bước 5:** Xác định chỉ số cuối cùng của phần tử trong mỗi phân lớp:

|  |  |  |  |
| --- | --- | --- | --- |
| k | 0 | 1 | 2 |
| Buckets[k] | 3 | 6 | 7 |

Giai đoạn 2:

* Bước 1:

Swap(a[0], a[maxPos])

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| A[i] | 134 | 55 | 123 | 98 | 83 | 23 | 7 |
|  | 2 | 0 | 1 | 1 | 1 | 0 | 0 |

* Bước 2:

+, countSwap = 0

Flash = a[i] = a[0] = 134 và = 2. Ta trừ đi Buckets[2] đi 1. Và cập nhật lại bảng vị trí của phần tử cuối cùng của các phân lớp như sau:

|  |  |  |  |
| --- | --- | --- | --- |
| k | 0 | 1 | 2 |
| Buckets[k] | 3 | 6 | 6 |

Swap (*Flash,* A[Buckets[2]])

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| A[i] | 7 | 55 | 123 | 98 | 83 | 23 | 134 |
|  | 0 | 0 | 1 | 1 | 1 | 0 | 2 |

+, countSwap = 1

Flash = a[i] = a[0] = 7 và = 0. Ta trừ đi Buckets[0] đi 1. Và cập nhật lại bảng vị trí của phần tử cuối cùng của các phân lớp như sau:

|  |  |  |  |
| --- | --- | --- | --- |
| k | 0 | 1 | 2 |
| Buckets[k] | 2 | 6 | 6 |

Swap (*Flash,* A[Buckets[0]])

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| A[i] | 123 | 55 | 7 | 98 | 83 | 23 | 134 |
|  | 1 | 0 | 0 | 1 | 1 | 0 | 2 |

+, countSwap = 2

Flash = a[i] = a[0] =123 và = 1. Ta trừ đi Buckets[1] đi 1. Và cập nhật lại bảng vị trí của phần tử cuối cùng của các phân lớp như sau:

|  |  |  |  |
| --- | --- | --- | --- |
| k | 0 | 1 | 2 |
| Buckets[k] | 2 | 5 | 6 |

Swap (*Flash,* A[Buckets[1]])

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| A[i] | 23 | 55 | 7 | 98 | 83 | 123 | 134 |
|  | 0 | 0 | 0 | 1 | 1 | 1 | 2 |

+, countSwap = 3

Flash = a[i] = a[0] = 23 và = 0. Ta trừ đi Buckets[0] đi 1. Và cập nhật lại bảng vị trí của phần tử cuối cùng của các phân lớp như sau:

|  |  |  |  |
| --- | --- | --- | --- |
| k | 0 | 1 | 2 |
| Buckets[k] | 1 | 5 | 6 |

Swap (*Flash,* A[Buckets[0]])

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| A[i] | 55 | 23 | 7 | 98 | 83 | 123 | 134 |
|  | 0 | 0 | 0 | 1 | 1 | 1 | 2 |

+, countSwap = 4

Flash = a[i] = a[0] = 55 và = 0. Ta trừ đi Buckets[0] đi 1. Và cập nhật lại bảng vị trí của phần tử cuối cùng của các phân lớp như sau:

|  |  |  |  |
| --- | --- | --- | --- |
| k | 0 | 1 | 2 |
| Buckets[k] | 0 | 5 | 6 |

Swap (*Flash,* A[Buckets[0]])

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| A[i] | 55 | 23 | 7 | 98 | 83 | 123 | 134 |
|  | 0 | 0 | 0 | 1 | 1 | 1 | 2 |

+, i = 0 = Buckets[0]: Phân lớp 0 đã đầy nên ta di chuyển qua phân tử khác cho đến khi tìm thấy phân tử đó chưa thuộc đúng lớp. à i=3

Flash = a[i] =a[3]=98 và = 1. Ta trừ đi Buckets[1] đi 1. Và cập nhật lại bảng vị trí của phần tử cuối cùng của các phân lớp như sau:

|  |  |  |  |
| --- | --- | --- | --- |
| k | 0 | 1 | 2 |
| Buckets[k] | 0 | 4 | 6 |

Swap (*Flash,* A[Buckets[1]])

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| A[i] | 23 | 55 | 7 | 83 | 98 | 123 | 134 |
|  | 0 | 0 | 0 | 1 | 1 | 1 | 2 |

+, countSwap = 6

Flash = a[i] =a[3]=83 và = 1. Ta trừ đi Buckets[1] đi 1. Và cập nhật lại bảng vị trí của phần tử cuối cùng của các phân lớp như sau:

|  |  |  |  |
| --- | --- | --- | --- |
| k | 0 | 1 | 2 |
| Buckets[k] | 0 | 3 | 6 |

Swap (*Flash,* A[Buckets[1]])

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| A[i] | 23 | 55 | 7 | 98 | 83 | 123 | 134 |
|  | 0 | 0 | 0 | 1 | 1 | 1 | 2 |

*+, countSwap = 7 > n - 1* nên ta đã hoàn thành 1 chu trình và có các phần tử đã về đúng phân lớp.

**Giai đoạn 3:** Tiến hành Insertion Sort trên từng phân lớp và ta có mảng đã được sắp xếp

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 7 | 23 | 55 | 83 | 98 | 123 | 134 |

### Độ phức tạp

|  |  |  |  |
| --- | --- | --- | --- |
| Trường hợp | Tốt nhất | Trung bình | Xấu nhất |
| Độ phức tạp | O(n) | O(n) | O(n2) |

Độ phức tạp không gian: O(n)

# KẾT QUẢ THỰC NGHIỆM

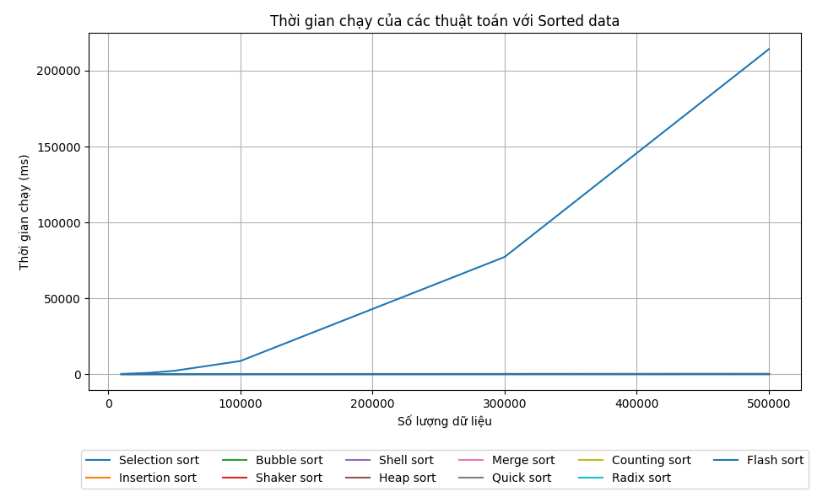
## Dữ liệu đã được sắp xếp tăng dần:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Data order: SORTED DATA | | | | | | |
| Data size | 10 000 | | 30 000 | | 50 000 | |
| Resulting static | Running time(ms) | Comparison | Running time(ms) | Comparison | Running time(ms) | Comparison |
| Selection Sort | 85.4356 | 100030003 | 776.195 | 900090003 | 2142.04 | 2500150003 |
| Insertion Sort | 0.0285 | 29998 | 0.0845 | 89998 | 0.1461 | 149998 |
| Bubble Sort | 0.0458 | 20001 | 0.05 | 60001 | 0.0859 | 100001 |
| Shaker Sort | 0.0336 | 40000 | 0.1896 | 120000 | 0.1808 | 200000 |
| Shell Sort | 0.4595 | 360042 | 1.1804 | 1170050 | 2.1112 | 2100049 |
| Heap Sort | 1.2865 | 670333 | 4.4073 | 2236652 | 8.2554 | 3925355 |
| Merge Sort | 3.0721 | 475242 | 10.7319 | 1559914 | 14.6398 | 2722826 |
| Quick Sort | 0.3063 | 233607 | 1.4672 | 747223 | 1.8176 | 1284455 |
| Counting Sort | 0.1534 | 60003 | 0.4766 | 180003 | 0.8049 | 300003 |
| Radix Sort | 0.6221 | 140056 | 2.1184 | 510070 | 3.7019 | 850070 |
| Flash Sort | 0.2317 | 127192 | 0.8464 | 381592 | 1.2767 | 635992 |

Bảng . Bảng thống kê thời gian chạy và số phép so sánh - dữ liệu sắp xếp tăng dần [1/2]

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Data order: SORTED DATA | | | | | | |
| Data size | 100 000 | | 300 000 | | 500 000 | |
| Resulting static | Running time(ms) | Comparison | Running time(ms) | Comparison | Running time(ms) | Comparison |
| Selection Sort | 8575.76 | 10000300003 | 77116.5 | 90000900003 | 214017 | 250001500003 |
| Insertion Sort | 0.3379 | 299998 | 0.8678 | 899998 | 1.3639 | 1499998 |
| Bubble Sort | 0.1769 | 200001 | 0.485 | 600001 | 0.8093 | 1000001 |
| Shaker Sort | 0.4652 | 400000 | 1.0125 | 1200000 | 1.6748 | 2000000 |
| Shell Sort | 4.4442 | 4500051 | 18.5674 | 15300061 | 25.481 | 25500058 |
| Heap Sort | 16.4896 | 8365084 | 53.8689 | 27413234 | 88.0972 | 47404890 |
| Merge Sort | 30.319 | 5745658 | 92.8859 | 18645946 | 153.067 | 32017850 |
| Quick Sort | 4.282 | 2668919 | 11.165 | 8475703 | 18.912 | 14475703 |
| Counting Sort | 1.6346 | 600003 | 4.6961 | 1800003 | 7.1266 | 3000003 |
| Radix Sort | 8.9105 | 1700070 | 26.9452 | 6000084 | 44.5611 | 10000084 |
| Flash Sort | 2.3781 | 1271992 | 7.287 | 3815992 | 11.8126 | 6359992 |

Bảng . Bảng thống kê thời gian chạy và số phép so sánh - dữ liệu sắp xếp tăng dần [2/2]

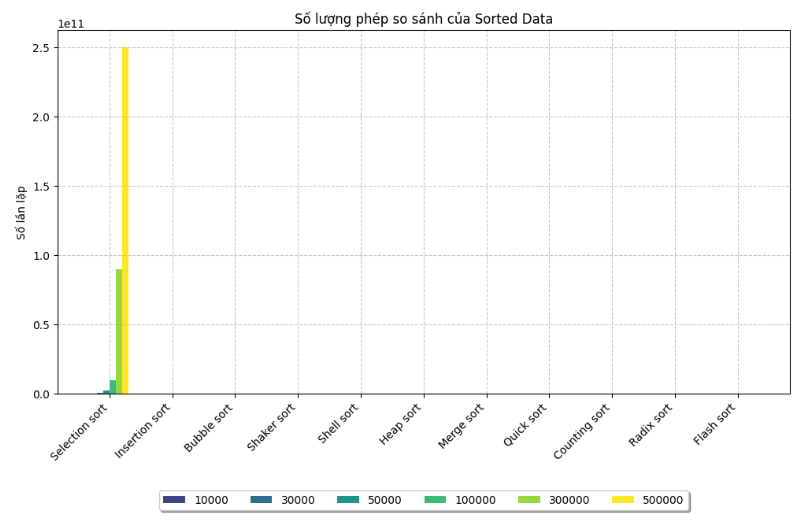


Hình . Biểu đồ đường thời gian chạy của các thuật toán trên kiểu dữ liệu sắp xếp tăng dần

Dựa vào biểu đồ ta có thể thấy:

Ta có thể thấy với dữ liệu càng lớn như từ 50000 phần tử trở lên, thời gian chạy của Selection Sort là lâu nhất.

Trong khi đó 3 thuật toán như Flash Sort, Counting Sort và Shaker Sort chạy với thời gian ít nhất, với Shaker Sort có thể đạt được đến 2 mili giây với dữ liệu 500000.



Hình 2. Biểu đồ cột số phép so sánh của các thuật toán trên kiểu dữ liệu sắp xếp tăng dần

Dựa vào biểu đồ ta có thể thấy:

Với mỗi số lượng input thì số phép so sánh của Selection Sort luôn lớn hơn các thuật toán còn lại. Và ta có thể thấy dữ liệu càng lớn thì số lần so sánh của thuật toán gần như cao gấp 2 đến 3 lần dữ liệu trước đó.

Thuật toán bubble sort có số lượng phép so sánh ít là do khi cài đặc nếu nhận thấy mảng không thay đổi ở bước nổi bọt đầu tiên thì chương trình sẽ ngừng lại do mảng đã được sắp xếp rồi nên số lượng phép so sánh khá là ít.

Và khi dữ liệu đã được sắp xếp số lần so sánh của các thuật toán nhiều đáng kể hơn so với các dữ liệu khác.

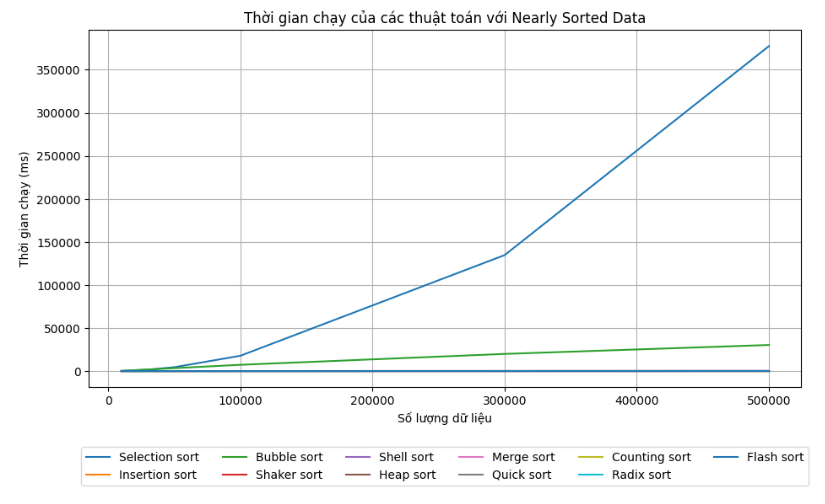
## Dữ liệu sắp xếp gần đúng:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Data order: NEARLY SORTED DATA | | | | | | |
| Data size | 10 000 | | 30 000 | | 50 000 | |
| Resulting static | Running time(ms) | Comparison | Running time(ms) | Comparison | Running time(ms) | Comparison |
| Selection Sort | 168.557 | 100030003 | 1615.87 | 900090003 | 4470.04 | 2500150003 |
| Insertion Sort | 0.3663 | 188546 | 1.157 | 623694 | 1.0637 | 632250 |
| Bubble Sort | 201.337 | 98332600 | 1802.06 | 898372600 | 3351.98 | 1643718305 |
| Shaker Sort | 0.8443 | 319993 | 2.3616 | 959993 | 4.0059 | 1599993 |
| Shell Sort | 0.8119 | 412721 | 3.1387 | 1333968 | 4.4867 | 2279600 |
| Heap Sort | 3.3829 | 669929 | 7.4083 | 2236544 | 13.8694 | 3925138 |
| Merge Sort | 3.0627 | 504144 | 18.2869 | 1677684 | 29.0129 | 2821340 |
| Quick Sort | 0.5061 | 235641 | 1.7936 | 759597 | 2.7255 | 1299280 |
| Counting Sort | 0.5653 | 60003 | 0.8418 | 180003 | 1.4561 | 300003 |
| Radix Sort | 1.6169 | 140056 | 4.5414 | 510070 | 6.6465 | 850070 |
| Flash Sort | 0.3107 | 127164 | 0.8865 | 381568 | 1.4582 | 635964 |

Bảng . Bảng thống kê thời gian chạy và số phép so sánh - dữ liệu sắp xếp gần đúng [1/2]

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Data order: NEARLY SORTED DATA | | | | | | |
| Data size | 100 000 | | 300 000 | | 500 000 | |
| Resulting static | Running time(ms) | Comparison | Running time(ms) | Comparison | Running time(ms) | Comparison |
| Selection Sort | 17745.1 | 10000300003 | 134723 | 90000900003 | 377527 | 250001500003 |
| Insertion Sort | 1.1714 | 694682 | 2.28 | 1249230 | 2.3052 | 1995766 |
| Bubble Sort | 7280.88 | 3564309825 | 19913.5 | 11235054897 | 30289.2 | 23569680897 |
| Shaker Sort | 7.5176 | 2799994 | 8.6251 | 8399994 | 29.9491 | 17999992 |
| Shell Sort | 9.5463 | 4672313 | 28.3349 | 15430770 | 50.0407 | 25665595 |
| Heap Sort | 26.8403 | 8364772 | 84.0732 | 27413259 | 144.181 | 47404770 |
| Merge Sort | 59.4037 | 5850936 | 165.829 | 18751832 | 276.28 | 32119423 |
| Quick Sort | 6.1581 | 2684159 | 18.2508 | 8491737 | 31.2409 | 14511636 |
| Counting Sort | 3.9407 | 600003 | 8.2402 | 1800003 | 14.22 | 3000003 |
| Radix Sort | 17.6917 | 1700070 | 41.3408 | 6000084 | 80.9729 | 10000084 |
| Flash Sort | 3.3708 | 1271964 | 9.5144 | 3815968 | 16.6182 | 6359962 |

Bảng . Bảng thống kê thời gian chạy và số phép so sánh - dữ liệu sắp xếp gần đúng [2/2]

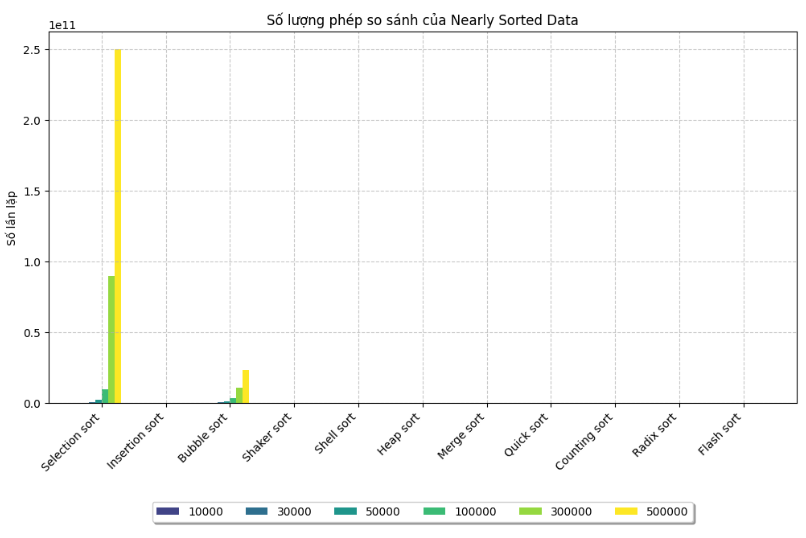


Hình . Biểu đồ đường thời gian chạy của các thuật toán trên kiểu dữ liệu sắp xếp gần đúng

Dựa vào biểu đồ đường và bảng số liệu, ta rút ra một số nhận xét sau đối với kiểu dữ liệu sắp xếp gần đúng:

Bubble Sort và Selection Sort là hai thuật toán sắp xếp chậm nhất. Thể hiện rõ nhất khi độ lớn dữ liệu đầu vào >= 50000

Insertion Sort, Counting Sort, Shaker Sort và Flash Sort là bốn thuật toán nhanh nhất.



Hình 4. Biểu đồ cột số phép so sánh của các thuật toán trên kiểu dữ liệu sắp xếp gần đúng

Dựa vào biểu đồ cột và bảng số liệu, ta rút ra một số nhận xét sau đối với kiểu dữ liệu sắp xếp gần đúng:

+, Bubble Sort và Selection Sort là hai thuật toán có số phép so sánh nhiều nhất.

+, Insertion Sort, Counting Sort, Shaker Sort và Flash Sort là bốn thuật toán dùng ít phép so sánh nhất.

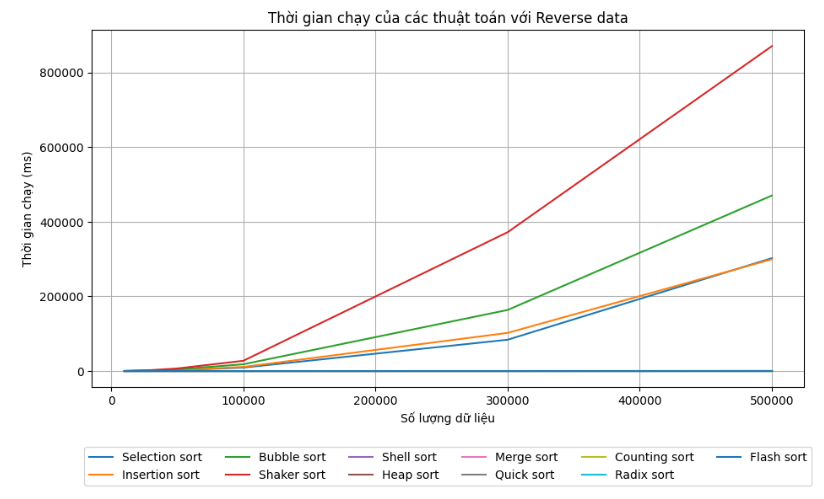
## Dữ liệu được sắp xếp ngược:

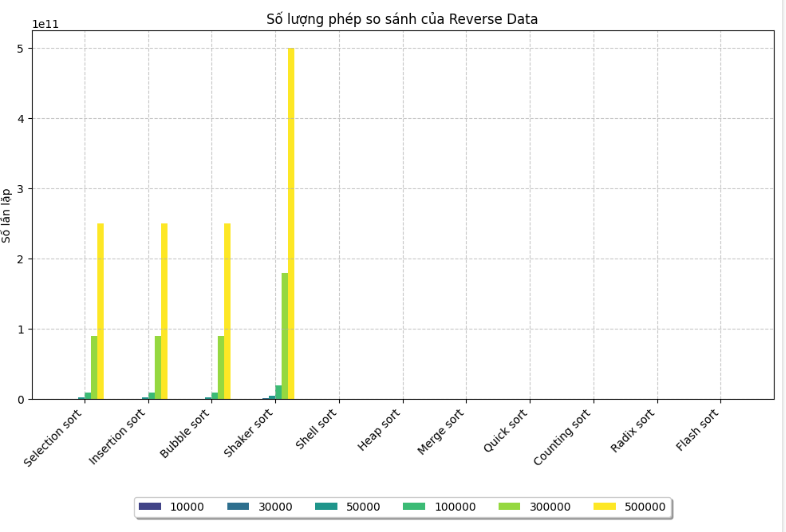
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Data order: REVERSE SORTED DATA | | | | | | |
| Data size | 10 000 | | 30 000 | | 50 000 | |
| Resulting static | Running time(ms) | Comparison | Running time(ms) | Comparison | Running time(ms) | Comparison |
| Selection Sort | 87.1047 | 100030003 | 787.46 | 900090003 | 2230.2 | 2500150003 |
| Insertion Sort | 102.692 | 100009999 | 947.21 | 900029999 | 2721.59 | 2500049999 |
| Bubble Sort | 176.685 | 100019998 | 1640.42 | 900059998 | 4547.02 | 2500099998 |
| Shaker Sort | 273.185 | 200035000 | 2474.93 | 1800105000 | 6929.18 | 5000175000 |
| Shell Sort | 0.5264 | 475175 | 1.7551 | 1554051 | 3.3464 | 2844628 |
| Heap Sort | 1.2065 | 606775 | 4.2555 | 2063328 | 7.3034 | 3612728 |
| Merge Sort | 2.8297 | 476441 | 8.7286 | 1573465 | 16.4556 | 2733945 |
| Quick Sort | 0.3589 | 248618 | 1.0671 | 792234 | 1.9158 | 1359466 |
| Counting Sort | 0.144 | 60003 | 0.4448 | 180003 | 0.8514 | 300003 |
| Radix Sort | 0.5573 | 140056 | 2.0627 | 510070 | 3.5896 | 850070 |
| Flash Sort | 0.2204 | 110701 | 0.7641 | 332101 | 1.1821 | 553501 |

Bảng . Bảng thống kê thời gian chạy và số phép so sánh - dữ liệu sắp xếp ngược [1/2]

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Data order: REVERSE SORTED DATA | | | | | | |
| Data size | 100 000 | | 300 000 | | 500 000 | |
| Resulting static | Running time(ms) | Comparison | Running time(ms) | Comparison | Running time(ms) | Comparison |
| Selection Sort | 9279.49 | 10000300003 | 83906.1 | 90000900003 | 302468 | 250001500003 |
| Insertion Sort | 11069.6 | 10000099999 | 102213 | 90000299999 | 299514 | 250000499999 |
| Bubble Sort | 18153.9 | 10000199998 | 163629 | 90000599998 | 470229 | 250000999998 |
| Shaker Sort | 27519.9 | 20000350000 | 371905 | 180001050000 | 870353 | 500001750000 |
| Shell Sort | 7.0055 | 6089190 | 40.8792 | 20001852 | 79.2743 | 33857581 |
| Heap Sort | 16.1797 | 7718947 | 90.0451 | 25569383 | 171.437 | 44483352 |
| Merge Sort | 29.7507 | 5767897 | 172.115 | 18708313 | 296.883 | 32336409 |
| Quick Sort | 4.0286 | 2818930 | 20.1892 | 8925714 | 42.0074 | 15225714 |
| Counting Sort | 1.4032 | 600003 | 9.1247 | 1800003 | 15.3776 | 3000003 |
| Radix Sort | 6.8921 | 1700070 | 56.534 | 6000084 | 91.2663 | 10000084 |
| Flash Sort | 2.22 | 1107001 | 9.0409 | 3321001 | 15.4854 | 5535001 |

Bảng . Bảng thống kê thời gian chạy và số phép so sánh - dữ liệu sắp xếp ngược [2/2]





Dựa vào biểu đồ trên ta có nhận xét sau:

+ Những thuật toán như Heap Sort và Merge Sort tỏ ra có hiệu suất tốt hơn rất nhiều so với các thuật toán cơ bản như Selection Sort, Insertion Sort và Bubble Sort khi chạy mảng có thứ tự đảo ngược: Nguyên do bởi vì các thuật toán sau sử dụng vòng lặp for với rất nhiều lần so sánh và gán, đặc biệt là khi mảng có thứ tự đảo ngược lại là Worst Case của thuật toán Selection Sort, Insertion Sort và Bubble Sort.

+ Những thuật toán như Shaker Sort được cho là có hiệu suất cao hơn Bubble Sort tuy nhiên số liệu cho thấy nó có thời gian chạy lớn hơn so với Bubble Sort: Nguyên do bởi vì thuật toán sử dụng chưa được tối ưu, số lần chạy vòng lặp gấp đôi nhưng phạm vi cần sắp xếp lại giữ nguyên khiến cho Shaker Sort mất nhiều thời gian hơn để chạy. Đề xuất cần giảm phạm vi sắp xếp ở hai biên của mảng sau mỗi vòng lặp để tối ưu hơn thuật toán này.

+ Những thuật toán Quick Sort, Radix Sort, Counting Sort, hay Flash Sort thể hiện hiệu suất đúng như cái tên của nó: rất hiệu quả và cần ít thời gian: Tuy nhiên những thuật toán này cần nhiều thời gian để tối ưu và sửa chữa (Ví dụ: Những phiên bản đầu tiên của thuật toán Quick Sort chúng em không thể chạy trên nhiều kiểu dữ liệu khác nhau).

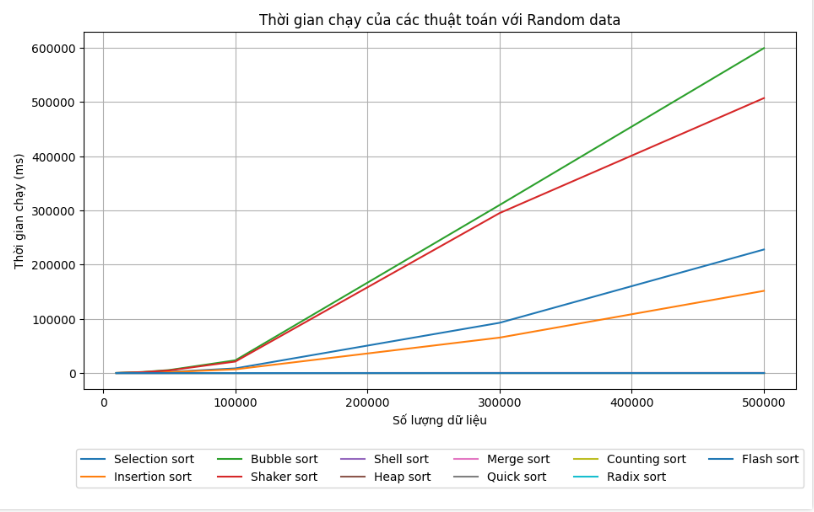
## Dữ liệu ngẫu nhiên:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Data order: RANDOMIZED DATA | | | | | | |
| Data size | 10 000 | | 30 000 | | 50 000 | |
| Resulting static | Running time(ms) | Comparison | Running time(ms) | Comparison | Running time(ms) | Comparison |
| Selection Sort | 95.1962 | 100030003 | 795.833 | 900090003 | 2120.41 | 2500150003 |
| Insertion Sort | 70.5768 | 50164649 | 621.519 | 448660836 | 1723.84 | 1247849336 |
| Bubble Sort | 197.053 | 99987960 | 1945.69 | 900021192 | 5586.63 | 2499933537 |
| Shaker Sort | 192.563 | 101357467 | 1734.09 | 902512480 | 4886.99 | 2497587513 |
| Shell Sort | 1.3564 | 637302 | 4.6975 | 2232387 | 8.966 | 4425509 |
| Heap Sort | 1.6236 | 638016 | 5.7786 | 2150020 | 9.5233 | 3772694 |
| Merge Sort | 3.6151 | 583631 | 11.5452 | 1937261 | 18.5477 | 3382968 |
| Quick Sort | 1.0282 | 345384 | 3.3346 | 1122331 | 5.6501 | 1909924 |
| Counting Sort | 0.225 | 60003 | 0.6576 | 180003 | 0.9048 | 265539 |
| Radix Sort | 0.5798 | 140056 | 2.0809 | 510070 | 3.7509 | 850070 |
| Flash Sort | 0.2399 | 99222 | 0.8071 | 290691 | 1.6092 | 495675 |

Bảng . Bảng thống kê thời gian chạy và số phép so sánh - kiểu dữ liệu ngẫu nhiên [1/2]

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Data order: RANDOMIZED DATA | | | | | | |
| Data size | 100 000 | | 300 000 | | 500 000 | |
| Resulting static | Running time(ms) | Comparison | Running time(ms) | Comparison | Running time(ms) | Comparison |
| Selection Sort | 8603.61 | 10000300003 | 92759.2 | 90000900003 | 228019 | 250001500003 |
| Insertion Sort | 6873.25 | 4996773979 | 65423.9 | 45048682532 | 151693 | 125058884449 |
| Bubble Sort | 23507.5 | 9999990237 | 310110 | 90000349000 | 599604 | 250000248312 |
| Shaker Sort | 21239.3 | 9980375050 | 295218 | 90213524823 | 507364 | 250237874882 |
| Shell Sort | 20.3628 | 10179766 | 94.3098 | 34481005 | 116.564 | 63950923 |
| Heap Sort | 20.8983 | 8044342 | 107.422 | 26485848 | 121.511 | 45969593 |
| Merge Sort | 40.3467 | 7166599 | 199.579 | 23382709 | 195.739 | 40382877 |
| Quick Sort | 11.3329 | 4019869 | 41.6879 | 12696525 | 57.4078 | 21171990 |
| Counting Sort | 1.5556 | 465539 | 6.3352 | 1265539 | 4.874 | 2065539 |
| Radix Sort | 7.2768 | 1700070 | 39.8043 | 5100070 | 37.4911 | 8500070 |
| Flash Sort | 2.5597 | 897368 | 13.2058 | 2809428 | 19.9816 | 4479288 |

Bảng . Bảng thống kê thời gian chạy và số phép so sánh - kiểu dữ liệu ngẫu nhiên [2/2]

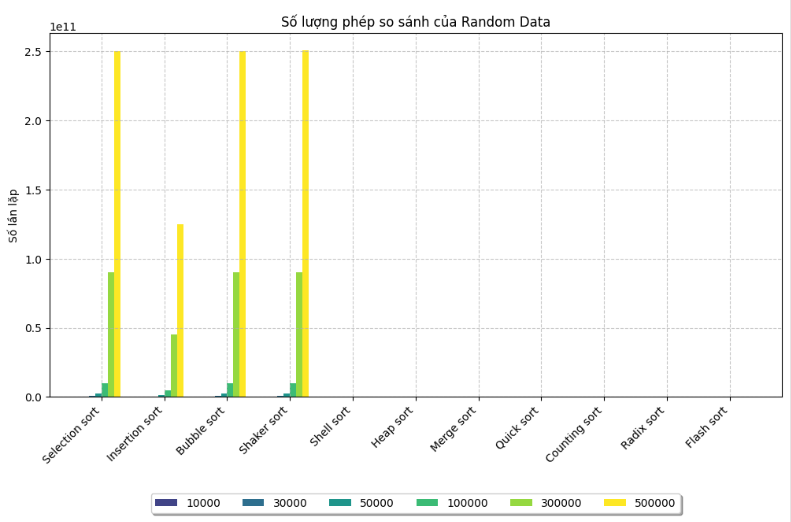


Dựa vào biểu đồ ta có các nhận xét sau:

+ Từ 10000 đến 50000 phần tử, biểu đồ không thể hiện rõ sự chênh lệch về thời gian của các thuật toán. Trong giai đoạn này hầu như máy tính vẫn có thể chạy mượt ra kết quả các thuật toán nhanh chóng.

+ Từ 50000 đến 100000 phần tử, biểu đồ bắt đầu có sự thay đổi khi thời gian của Shaker Sort và Bubble Sort có phần cần nhiều thời gian hơn để sắp xếp.

+ Từ 100000 đến 500000 phần tử, biểu đồ thể hiện rõ sự thay đổi thời gian của Bubble Sort và Shaker Sort. Cần nhiều thời gian nhất là bubble Sort, sau đó đến Shaker Sort, Selection Sort, Insertion Sort. Đặc biệt, từ 300000 đến 500000 phần tử, Bubble Sort và Shaker Sort gần như cần thêm gấp 3 lần thời gian để sắp xếp.



Dựa vào biểu đồ ta có nhận xét sau:

+, Biểu đồ thể hiện rõ nét sự chênh lệch về số phép so sánh của các thuật toán trong giai đoạn từ 100000 đến 500000 phần tử. Nhiều phép so sánh nhất là Shaker Sort và kế đến là Insertion Sort.

+, Trong đó, ở 300000 phần tử, Shaker Sort gần chạm mốc 1E+11 phép so sánh và Insertion Sort gần chạm mốc 5E+10. Ở 500000 phần tử, Shaker Sort tăng gấp hơn 2.5 lần phép so sánh.

# TỔNG QUAN VỀ CHƯƠNG TRÌNH

## Cách tổ chức source code

Thư mục SOURCE được tổ chức từ 3 thư mục:

Thư mục Algorithm: chứa 11 file .cpp và .h của 11 thuật toán

Thư mục DataGenerator: chứa file .cpp và .h của 4 kiểu khởi tạo dữ liệu và trong đó cũng cài đặc luôn các số lượng cần để chạy để thực thi chạy để so sánh.

Thư mục main: chứa 2 file .cpp. Lab3.cpp là của việc chạy các thuật toán so sánh với các loại dữ liệu khác nhau và số lượng dữ liệu khác nhau để có thể lấy thông số. a.cpp là file thực thi 5 CMD được đề bài yêu cầu và file a.exe để có thể chạy ở CMD

Để chạy được file a.exe ta cần mở command line ở thư mục đang chứa file .exe sau đó gõ: .\a.exe

Tiếp theo dòng lệnh trên là các cú pháp đã mặc định của các command.

## Các thư viện đặc biệt

Chương trình sử dụng các thư viện sau:

* Thư viện <vector> để khai báo biến vector.
* Thư viện <chrono> để đo thời gian thực thi của thuật toán.

## Thông số máy tính và IDE đã dùng

Toàn bộ quá trình chạy thử nghiệm 11 thuật toán sắp xếp để lấy dữ liệu được thực hiện trên desktop ASUS, 12th Gen Intel® CoreTM i7-12700H, RAM 16GB, ổ cứng SSD và chạy trên Visual Studio 2022.

## Chương trình Command

5 chế độ CMD được lập trình trong một chương trình và tận dụng những đặc điểm khác nhau của chúng khi nhập đầu vào mà chương trình sẽ trả về kết quả tương ứng.

CMD 1: ký tự argv[1] sẽ là -a và argv[3] sẽ không phải là số.

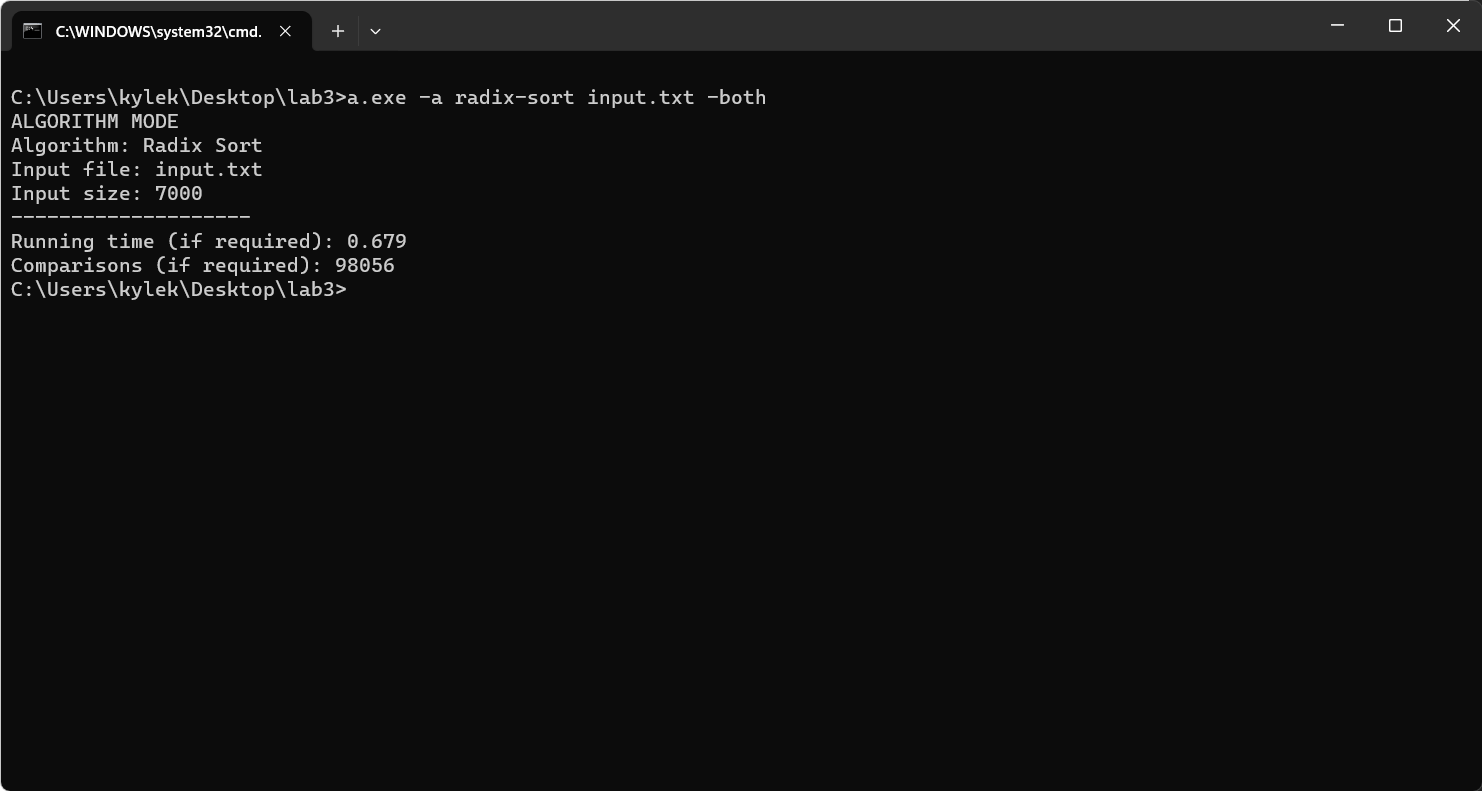
CMD 2: ký tự argv[1] sẽ là -a và argv[3] là số và là số lượng argv là 6.

CMD 3: ký tự argv[1] sẽ là -a và argv[3] là số và số lượng argv là 5.

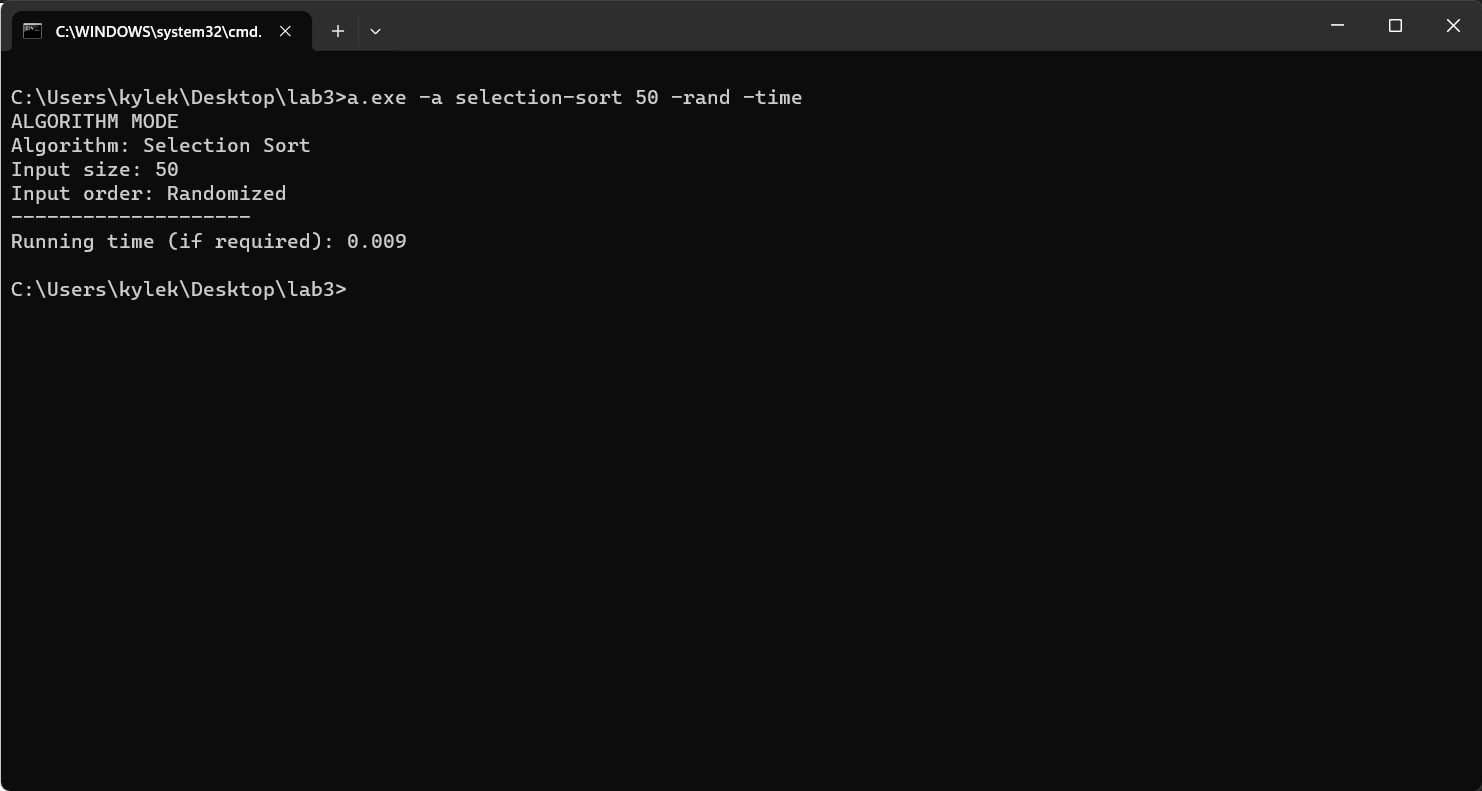
CMD 4: ký tự argv[1] sẽ là -c và số lượng argv là 5.

CMD 5: ký tự argv[1] sẽ là -c và số lượng argv là 6.

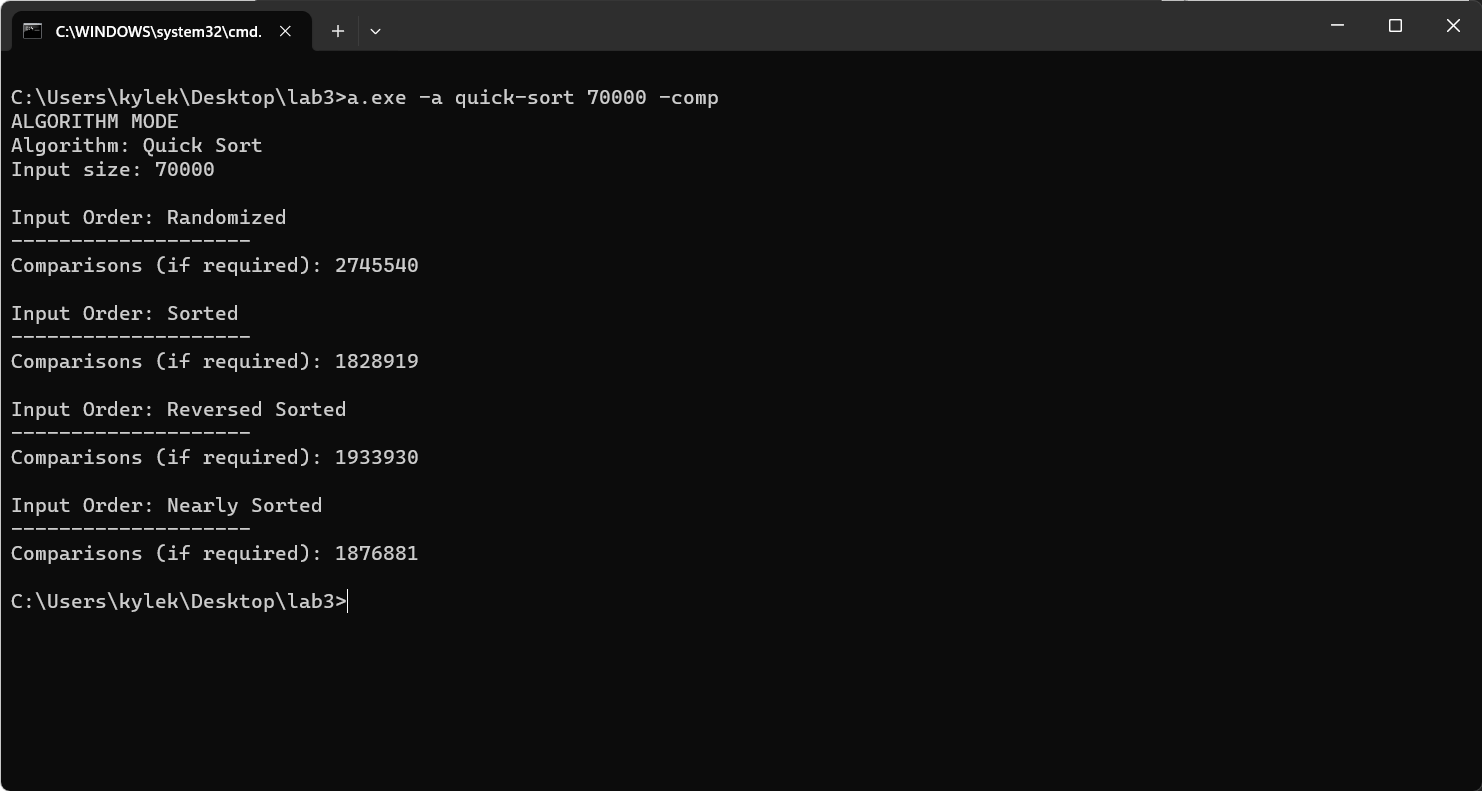
Các kết quả khi chạy chương trình:



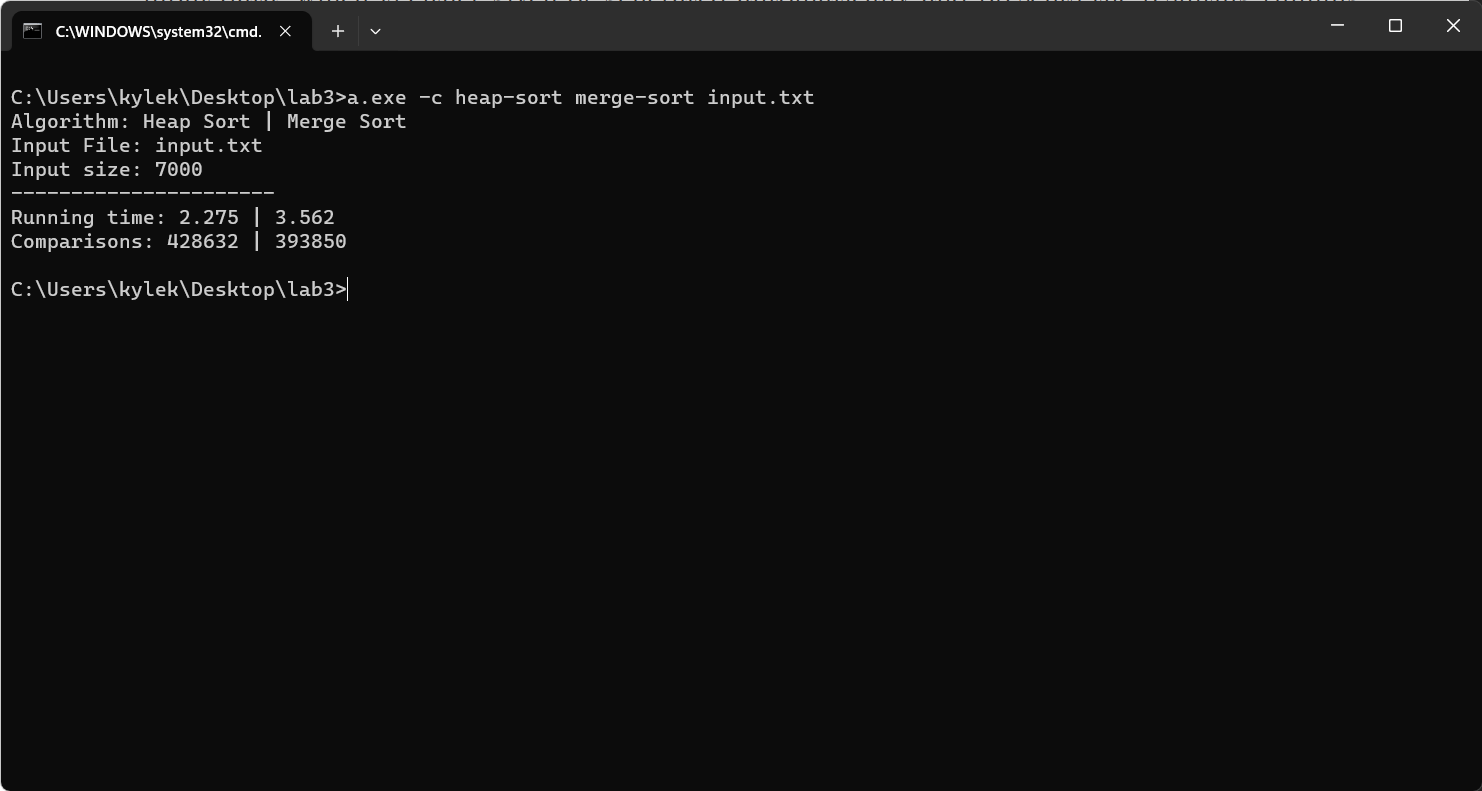
Hình 1: Kết quả của CMD 1



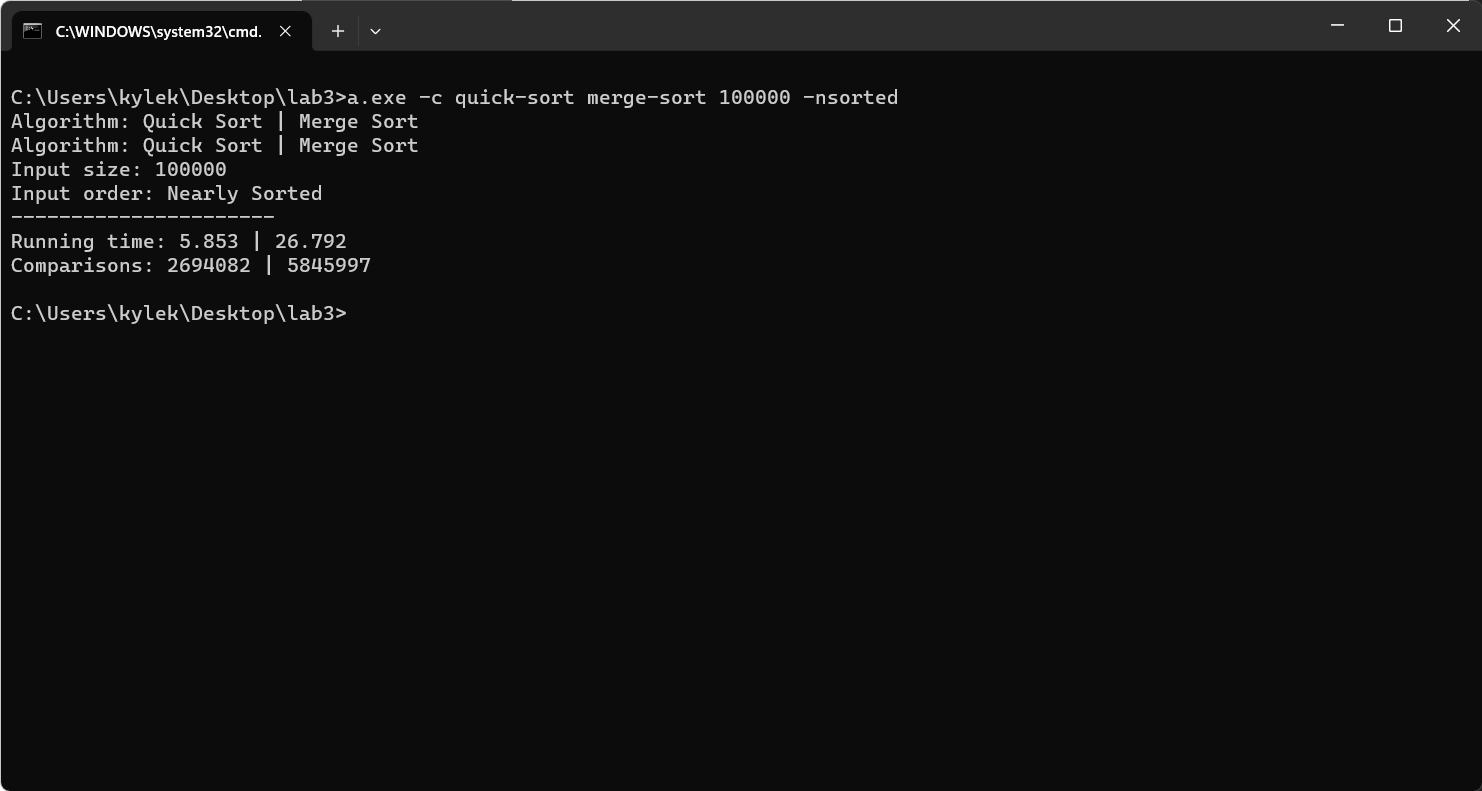
Hình 2: Kết quả của CMD 2



Hình 3: Kết quả của CMD 3



Hình 4: Kết quả của CMD 4



Hình 5: Kết quả của CMD 5

# TÀI LIỆU THAM KHẢO

|  |  |
| --- | --- |
| [1] | V. C. Nam, "Slides," [Online]. [Accessed 18 11 2023]. |
| [2] | "Luyện Code," [Online]. Available: https://blog.luyencode.net/wp-content/uploads/2018/07/thuat-toan-sap-xep-merge-Sort-minh-hoa-code-su-dung-c.png. [Accessed 25 11 2023]. |
| [3] | "[C++]. Thuật Toán Sắp Xếp Nhanh QuickSort | Phân Hoạch Lomuto Và Phân Hoạch Hoare," 28tech, [Online]. Available: https://www.youtube.com/watch?v=eT9Epyf0XLM&t=1223s. [Accessed 25 11 2023]. |
| [4] | "VNSCHOOl," [Online]. Available: http://www.vnschool.net/modules.php?name=News&file=article&sid=3855. [Accessed 25 11 2023]. |
| [5] | T. T. H. Nhi and Đ. A. Th.S Dương, Nhập môn Cấu trúc dữ liệu và Giải thuật, 2003. |
| [6] | "C Share To Learn More," [Online]. Available: https://cafedev.vn/thuat-toan-heapSort-gioi-thieu-chi-tiet-va-code-vi-du-tren-nhieu-ngon-ngu-lap-trinh/. [Accessed 18 11 2023]. |
| [7] | "C Share To Learn More," [Online]. Available: https://cafedev.vn/thuat-toan-bubble-Sort-gioi-thieu-chi-tiet-va-code-vi-du-tren-nhieu-ngon-ngu-lap-trinh/?fbclid=IwAR3dIZLYRV6T5YMnwIXuLhuGVzThb9Lrrfy6EttsACp9ax9WEtf-8wL54CA. [Accessed 18 11 2023]. |
| [8] | "C Share To Learn More," [Online]. Available: https://cafedev.vn/thuat-toan-counting-Sort-gioi-thieu-chi-tiet-va-code-vi-du-tren-nhieu-ngon-ngu-lap-trinh/. [Accessed 25 11 2023]. |
| [9] | "codelearn.io," [Online]. Available: https://codelearn.io/sharing/flash-Sort-thuat-toan-sap-xep-than-thanh. [Accessed 25 11 2023]. |
| [10] | "FREETUTS," [Online]. Available: https://freetuts.net/thuat-toan-sap-xep-chen-insertion-Sort-2930.html. [Accessed 223 11 2023]. |
| [11] | "GeeksforGeeks," [Online]. Available: https://www.geeksforgeeks.org/cpp-program-for-shellSort/. [Accessed 18 11 2023]. |
| [12] | "GeeksforGeeks," [Online]. Available: https://www.geeksforgeeks.org/radix-Sort/. [Accessed 25 11 2023]. |