

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN, ĐHQG-HCM
KHOA CÔNG NGHỆ THÔNG TIN



BÁO CÁO ĐỒ ÁN CÁC THUẬT TOÁN SẮP XẾP

MÔN HỌC: CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

GVHD: TRẦN HOÀNG QUÂN

Sinh viên thực hiện:

TRẦN TRÚC NGỌC	MSSV: 23120148
HỒ KHÔNG TUYẾT NHƯ	MSSV: 23120152
NGUYỄN ĐOÀN XUÂN THU	MSSV: 23120170
NGUYỄN ANH TUẤN	MSSV: 23120184
NGUYỄN BÁCH KHA VÂN	MSSV: 23120188

Thành phố Hồ Chí Minh, Ngày 24 Tháng 12 Năm 2024

MỤC LỤC

1. GIỚI THIỆU.....	4
2. THÔNG TIN.....	5
2.1. Tham số dòng lệnh.....	5
2.2. Khởi tạo dữ liệu.....	7
2.3. Tính toán độ phức tạp.....	7
2.4. Thông số kỹ thuật.....	7
3. CÁC THUẬT TOÁN KHẢO SÁT.....	9
3.1. Selection Sort.....	9
3.2. Insertion Sort.....	11
3.3. Bubble Sort.....	14
3.4. Shaker Sort.....	17
3.5. Shell Sort.....	20
3.6. Heap Sort.....	23
3.7. Merge Sort.....	26
3.8. Quick Sort.....	29
3.9. Counting Sort.....	33
3.10. Radix Sort.....	38
3.11. Flash Sort.....	41
4. KẾT QUẢ THỰC NGHIỆM VÀ NHẬN XÉT.....	47
4.1. Bộ dữ liệu có thứ tự ngẫu nhiên.....	47
4.2. Bộ dữ liệu gần được sắp xếp hoàn chỉnh.....	51
4.3. Bộ dữ liệu đã sắp xếp.....	54
4.4. Bộ dữ liệu sắp xếp ngược.....	58
4.5. Nhận xét tổng quan.....	62
5. CÁC LƯU Ý VỚI BÀI NỘP.....	64
5.1. Tổ chức dự án.....	64
5.2. Lưu ý trong mã nguồn.....	64
6. TÀI LIỆU THAM KHẢO.....	65
7. LỜI KẾT.....	68

MỤC LỤC HÌNH ẢNH

Hình 1: Cấu hình máy (Hình 1).....	8
Hình 2: Cấu hình máy (Hình 2).....	8
Hình 3: Mã giả Selection Sort.....	9
Hình 4: Ví dụ Selection Sort.....	10
Hình 5: Mã giả Insertion Sort.....	11
Hình 6: Ví dụ Insertion Sort.....	12
Hình 7: Mã giả Bubble Sort.....	15
Hình 8: Ví dụ Bubble Sort.....	16
Hình 9: Mã giả Shaker Sort (Hình 1).....	18
Hình 10: Mã giả Shaker Sort (Hình 2).....	18
Hình 11: Ví dụ Shaker Sort.....	19
Hình 12: Mã giả Shell Sort (Hình 1).....	21
Hình 13: Mã giả Shell Sort (Hình 2).....	21
Hình 14: Ví dụ Shell Sort.....	22
Hình 15: Mã giả Heap Sort.....	23
Hình 16: Ví dụ Heap Sort.....	24
Hình 17: Mã giả Merge Sort.....	26
Hình 18: Ví dụ minh họa Merge Sort.....	28
Hình 19: Mã giả Quick Sort (Hình 1).....	30
Hình 20: Mã giả Quick Sort (Hình 2).....	31
Hình 21: Ví dụ Quick Sort (Partition).....	32
Hình 22: Ví dụ Quick Sort.....	33
Hình 23: Mã giả Counting Sort.....	34
Hình 24: Ví dụ Counting Sort.....	36
Hình 25: Mã giả Radix Sort.....	38
Hình 26: Ví dụ Radix Sort.....	40
Hình 27: Mã giả Flash Sort (Hình 1).....	43
Hình 28: Mã giả Flash Sort (Hình 2).....	44
Hình 29: Ví dụ Flash Sort.....	44
Hình 30: Bảng dữ liệu ngẫu nhiên.....	47
Hình 31: Biểu đồ đường dữ liệu ngẫu nhiên.....	49
Hình 32: Biểu đồ cột dữ liệu ngẫu nhiên.....	50
Hình 33: Bảng dữ liệu gần sắp xếp.....	52
Hình 34: Biểu đồ đường dữ liệu gần sắp xếp.....	52
Hình 35: Biểu đồ cột dữ liệu gần sắp xếp.....	54
Hình 36: Bảng dữ liệu đã sắp xếp.....	55
Hình 37: Biểu đồ đường dữ liệu đã sắp xếp.....	56
Hình 38: Biểu đồ cột dữ liệu đã sắp xếp.....	58
Hình 39: Bảng dữ liệu sắp xếp ngược.....	60
Hình 40: Biểu đồ đường dữ liệu sắp xếp ngược.....	60
Hình 41: Biểu đồ cột dữ liệu sắp xếp ngược.....	62

1. GIỚI THIỆU

Trong khoa học máy tính, thuật toán sắp xếp đóng vai trò quan trọng trong việc tổ chức dữ liệu. Mục tiêu của thuật toán sắp xếp là sắp xếp các phần tử của một danh sách theo một thứ tự cụ thể, như tăng dần hoặc giảm dần. Việc sắp xếp hiệu quả không chỉ giúp tối ưu hóa các thuật toán khác mà còn tạo ra dữ liệu đầu ra rõ ràng, dễ hiểu.

Để đánh giá hiệu suất của thuật toán sắp xếp, ta thường xét hai chỉ số chính: số lượng phép so sánh và thời gian thực thi. Số lượng phép so sánh cho biết lượng công việc mà thuật toán phải thực hiện, trong khi thời gian thực thi thể hiện thời gian để hoàn thành quá trình sắp xếp.

Thời gian thực thi của một thuật toán phụ thuộc vào nhiều yếu tố, bao gồm cấu hình phần cứng, ngôn ngữ lập trình và đặc tính của dữ liệu đầu vào. Việc lựa chọn thuật toán sắp xếp phù hợp sẽ giúp tối ưu hóa hiệu suất và trải nghiệm người dùng.

Báo cáo bắt đầu bằng việc giới thiệu các thuật toán cơ bản, minh họa bằng mã giả và ví dụ cụ thể. Tiếp theo, báo cáo sẽ đi sâu vào phân tích chi tiết về các đặc điểm hiệu suất của từng thuật toán, dựa trên cả lý thuyết và kết quả thực nghiệm.

Thuật toán sắp xếp không chỉ là một khái niệm lý thuyết mà còn là công cụ cốt lõi để tối ưu hóa hiệu suất và nâng cao trải nghiệm người dùng trong các ứng dụng thực tế. Từ việc tối ưu hóa truy vấn cơ sở dữ liệu đến cải thiện thuật toán tìm kiếm, việc lựa chọn thuật toán sắp xếp phù hợp sẽ trực tiếp tác động đến tốc độ xử lý và chất lượng kết quả của hệ thống.

Mục tiêu của báo cáo này là phân tích sâu rộng các thuật toán sắp xếp. Chúng ta sẽ so sánh và đánh giá hiệu suất của từng thuật toán dựa trên các chỉ số đã nêu. Báo cáo sẽ bắt đầu bằng việc giới thiệu các thuật toán cơ bản, sau đó đi sâu vào phân tích chi tiết về hiệu suất của từng thuật toán, kết hợp cả lý thuyết và thực nghiệm.

2. THÔNG TIN

2.1. Tham số dòng lệnh

Chương trình sử dụng các đối số dòng lệnh (argc và argv[]) để xác định chế độ hoạt động và các thao tác cần thực hiện.

Đầu tiên, chương trình sẽ bắt đầu bằng cách kiểm tra số lượng tham số được truyền vào (argc).

Chương trình có hai chế độ chính:

- **-a (Chế độ thuật toán):** Dùng để chạy một thuật toán sắp xếp cụ thể trên một tệp tin đầu vào hoặc một mảng được tạo ra.
- **-c (Chế độ so sánh):** Dùng để so sánh hiệu suất của hai thuật toán sắp xếp trên cùng một dữ liệu đầu vào.

Chế độ -a (Chế độ thuật toán):

Trong chế độ này, chương trình sẽ chạy một thuật toán sắp xếp trên một mảng cho trước (có trong tệp) hoặc được sinh tự động, sau đó tùy vào yêu cầu lấy từ dòng lệnh mà in ra thời gian chạy và / hoặc số lượng phép so sánh đã thực hiện.

- **Nếu đầu vào là tệp tin:**
 - Chương trình đọc tệp tin đầu vào (argv[3]).
 - Thuật toán cần sử dụng được truyền qua argv[2].
 - Tham số đầu ra (chọn giữa -time, -comp, -both) được truyền qua argv[4].
 - Chương trình sẽ thực thi thuật toán sắp xếp trên dữ liệu, thu thập các thông số hiệu suất (thời gian, số lần so sánh đã thực hiện), và ghi dữ liệu đã sắp xếp vào tệp tin đầu ra (output.txt).
 - Tùy thuộc vào tham số argv[4], chương trình sẽ in ra thời gian chạy, số phép so sánh, hoặc cả hai.
- **Nếu đầu vào là một số:**
 - Chương trình sẽ tạo một mảng có kích thước n (được truyền qua argv[3]) với 4 loại dữ liệu là dữ liệu ngẫu nhiên, dữ liệu gần như đã sắp xếp, dữ liệu đã sắp xếp, và dữ liệu đảo ngược.
 - Chương trình sẽ lưu mảng vào các tệp tin khác nhau lần lượt là input_1.txt, input_2.txt, input_3.txt và input_4.txt.
 - Sau đó, thuật toán được chọn (lấy từ argv[2]) sẽ áp dụng lên mảng đã tạo và tính các thông số hiệu suất (thời gian, số phép so sánh).

- Chương trình hiển thị thời gian và số lần so sánh dựa trên các tùy chọn được cung cấp.

Chế độ -c (Chế độ so sánh):

Chế độ này cho phép so sánh hai thuật toán sắp xếp trên cùng một dữ liệu đầu vào.

- Hai thuật toán (argv[2] và argv[3]) sẽ được so sánh.
- Tệp tin đầu vào (argv[4]) sẽ được đọc, và từ dữ liệu đó sẽ sao chép qua ba mảng khác nhau.
- Sau đó, chương trình sẽ thực thi hai thuật toán này và đo thời gian chạy cũng như số lần so sánh cho mỗi thuật toán.
- Kết quả in ra bao gồm:
 - Thời gian chạy của mỗi thuật toán.
 - Số phép so sánh của mỗi thuật toán.

Các tham số dòng lệnh (command-line argument) được cung cấp sẵn:

- **Command 1:** Chạy một thuật toán trên một mảng cho trước.

Cú pháp: [Execution file] -a [Algorithm] [Given input] [Output parameter(s)]

Ghi lại mảng đã sắp xếp vào tệp “output.txt”.

- **Command 2:** Chạy một thuật toán với mảng được sinh tự động theo thứ tự và kích thước cho trước.

Cú pháp: [Execution file] -a [Algorithm] [Input size] [Input order] [Output parameter(s)]

Ghi lại đầu vào đã tạo vào tệp “input.txt”.

Viết mảng đã sắp xếp vào tệp “output.txt”.

- **Command 3:** Chạy một thuật toán trên tất cả các kích thước và thứ tự mảng.

Cú pháp: [Execution file] -a [Algorithm] [Input size] [Output parameter(s)]

Ghi mảng đã tạo vào các tệp tin: “input_1.txt”, “input_2.txt”, “input_3.txt”, và “input_4.txt”.

- **Command 4:** Chạy hai thuật toán trên một mảng cho trước.

Cú pháp: [Execution file] -c [Algorithm 1] [Algorithm 2] [Given input]

- **Command 5:** Chạy hai thuật toán trên mảng được sinh tự động.

Cú pháp: [Execution file] -c [Algorithm 1] [Algorithm 2] [Input size] [Input order]

Ghi mảng đã tạo vào tệp “input.txt”.

2.2. Khởi tạo dữ liệu

Nhóm em đã sử dụng các hàm thay cho sẵn trong **DataGenerator.cpp** để tạo ra mảng dữ liệu phục vụ cho việc kiểm tra, đo đặc và so sánh các thuật toán.

- Mảng ngẫu nhiên: Hàm GenerateRandomData().
- Mảng tăng dần: Hàm GenerateSortedData().
- Mảng giảm dần: Hàm GenerateReverseData().
- Mảng gần như sắp xếp: Hàm GenerateNearlySortedData().

Ngoài ra, hàm GenerateData dùng để phát sinh dữ liệu theo yêu cầu, giúp kiểm tra hiệu suất các thuật toán sắp xếp.

2.3. Tính toán độ phức tạp

Tính toán độ phức tạp thuật toán có hai phép đo chính: đo thời gian thực thi và đo số lượng phép so sánh.

- Để đo thời gian thực thi, chúng em sử dụng hàm high_resolution_clock::now() trong thư viện chrono. Chúng em sử dụng các biến start và end để ghi lại thời gian sau đó sử dụng lớp duration để tính toán thời gian thực thi của thuật toán. Đơn vị đo thời gian nhóm chúng em sử dụng là millisecs.
- Để đếm số lượng phép so sánh chúng em sử dụng biến đếm count_compare, nó được khởi tạo bằng 0 và chèn vào những nơi có phép so sánh để đếm số lượng.

2.4. Thông số kỹ thuật

Nhóm chúng em sử dụng cấu hình máy tính này để thực hiện đo thời gian và số phép so sánh của tất cả các thuật toán trong dự án này:

- **Model:** MacBook Air (M1, 2020).
- **Chip:** Apple M1.
- **Tổng số lõi:** 8 (4 lõi hiệu năng cao và 4 lõi tiết kiệm năng lượng).
- **RAM:** 8GB.
- **Màn hình:** Built-in Retina Display (13.3-inch, độ phân giải 2560 x 1600).
- **Dung lượng ổ cứng:**
 - + Tổng dung lượng: 245.11 GB.
 - + Dung lượng trống: 137.47 GB.
- **Hệ điều hành:** macOS Ventura (phiên bản 13.3).

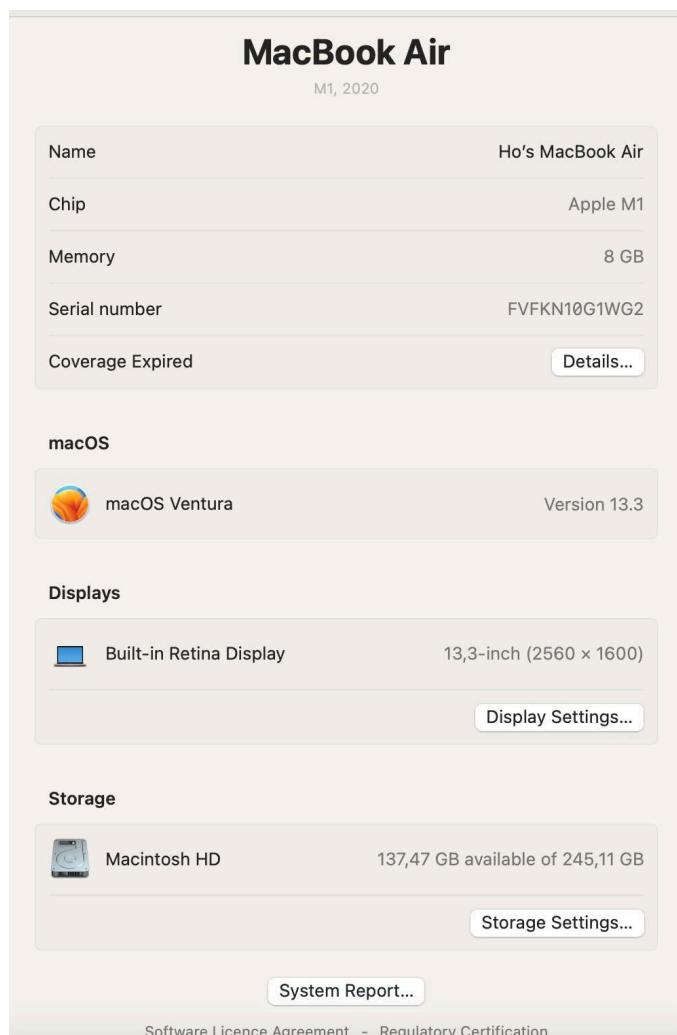
Hardware:**Hardware Overview:**

```

Model Name: MacBook Air
Model Identifier: MacBookAir10,1
Model Number: MGND3SA/A
Chip: Apple M1
Total Number of Cores: 8 (4 performance and 4 efficiency)
Memory: 8 GB
System Firmware Version: 8422.100.650
OS Loader Version: 8422.100.650
Serial Number (system): FVFKN10G1WG2
Hardware UUID: 33547BE0-F313-5E25-A1C3-9BF14FD97CF4
Provisioning UDID: 00008103-000E65221402201E
Activation Lock Status: Disabled

```

tuyetnhu@Hos-MacBook-Air ~ % └─

Hình 1: Cấu hình máy (Hình 1)**Hình 2: Cấu hình máy (Hình 2)**

3. CÁC THUẬT TOÁN KHẢO SÁT

3.1. Selection Sort

Ý tưởng chung của thuật toán:

Selection Sort thuộc nhóm các thuật toán sắp xếp so sánh tại chỗ (in-place comparison sorting). Thuật toán này được gọi là **selection sort** vì nó hoạt động bằng cách chọn phần tử nhỏ nhất trong mỗi lần duyệt (bước) của quá trình sắp xếp.

Cụ thể, để sắp xếp dữ liệu theo thứ tự tăng dần, phần tử đầu tiên sẽ được so sánh với tất cả các phần tử còn lại. Nếu phần tử đầu tiên lớn hơn phần tử nhỏ nhất, vị trí của hai phần tử sẽ được hoán đổi. Nhờ đó, sau lần duyệt đầu tiên, phần tử nhỏ nhất sẽ được đặt ở vị trí đầu tiên. Quá trình tương tự được lặp lại với phần tử thứ hai, rồi đến các phần tử tiếp theo, cho đến khi toàn bộ danh sách được sắp xếp.

Các bước của thuật toán:

SELECTION SORT (A,n)

1. for $i = 0$ to $n - 2$ do $i = 0$
2. $min = i$
3. for $j = i + 1$ to $n - 1$ do
4. if $A[j] < A[min]$ then
5. $min = j$
6. Swap($A[i], A[min]$)

Hình 3: Mã giả Selection Sort

Bước 1: Đặt phần tử đầu tiên làm giá trị nhỏ nhất.

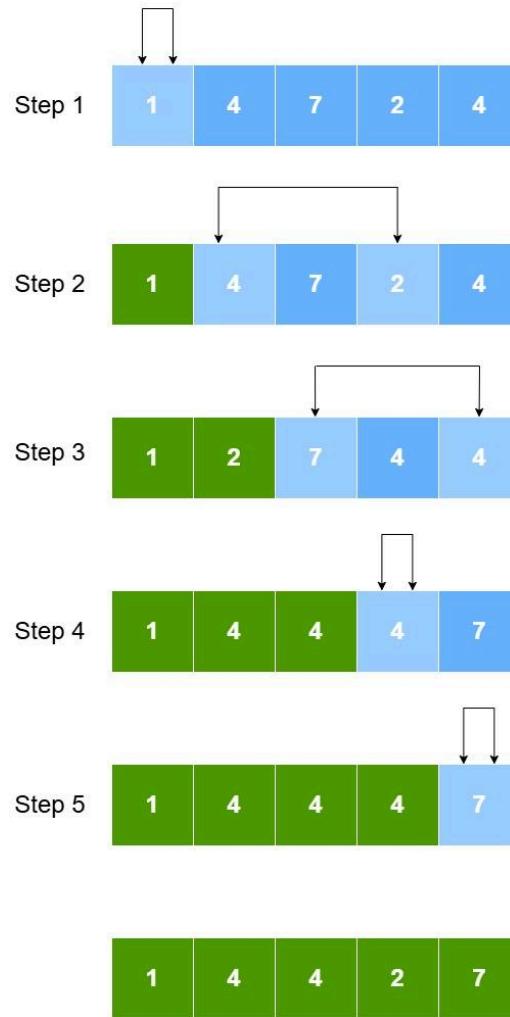
Bước 2: So sánh giá trị nhỏ nhất với phần tử thứ hai. Nếu phần tử thứ hai nhỏ hơn giá trị nhỏ nhất, gán giá trị nhỏ nhất là phần tử thứ hai.

Bước 3: So sánh giá trị nhỏ nhất với phần tử thứ ba. Nếu phần tử thứ ba nhỏ hơn, gán giá trị nhỏ nhất với phần tử thứ ba, nếu không thì không làm gì cả. Quá trình này tiếp tục cho đến phần tử cuối cùng.

Bước 4: Sau mỗi lần lặp, giá trị nhỏ nhất sẽ được đưa về vị trí đầu tiên của danh sách chưa được sắp xếp.

Bước 5: Ở mỗi lần lặp tiếp theo, chỉ số bắt đầu sẽ là phần tử đầu tiên của danh sách chưa được sắp xếp. Các bước từ 1 đến 3 được lặp lại cho đến khi tất cả các phần tử được đưa về đúng vị trí của chúng.

Ví dụ minh họa:

**Hình 4: Ví dụ Selection Sort**

Bước 1: Tìm phần tử nhỏ nhất trong mảng và hoán vị nó với phần tử đầu tiên. Lúc này, 1 tự hoán vị với chính nó.

Bước 2: Tìm phần tử nhỏ nhất trong mảng chưa được sắp xếp và hoán vị nó với phần tử thứ hai. Lúc này, 2 hoán vị với 4.

Bước 3: Lần lượt làm như vậy cho đến khi tất cả các phần tử được đưa về đúng vị trí.

Đánh giá độ phức tạp thuật toán:[1][2]

- **Phức tạp thời gian:**

Trường hợp tốt nhất: $O(n^2)$

Xảy ra khi mảng đã được sắp xếp sẵn theo thứ tự tăng dần.

Trường hợp trung bình: $O(n^2)$

Xảy ra khi các phần tử trong mảng có thứ tự lộn xộn.

Trường hợp xấu nhất: $O(n^2)$

Xảy ra khi chúng ta cần sắp xếp mảng theo thứ tự tăng dần nhưng mảng ban đầu lại được sắp xếp theo thứ tự giảm dần.

Độ phức tạp thời gian của Selection Sort giống nhau trong mọi trường hợp, vì ở mỗi bước, cần tìm phần tử nhỏ nhất và đưa nó vào đúng vị trí. Phần tử nhỏ nhất chỉ được xác định khi đã duyệt qua toàn bộ mảng.

- **Phức tạp không gian:**

Độ phức tạp không gian: $O(1)$ vì chỉ cần thêm một biến phụ để lưu chỉ số của phần tử nhỏ nhất trong mỗi lần lặp.

3.2. Insertion Sort

Ý tưởng chung của thuật toán:

Insertion Sort là thuật toán mô phỏng cách mà người chơi bài thường sắp xếp bài trên tay. Họ giữ các lá bài đã được chia theo thứ tự sắp xếp, và khi nhận thêm một lá bài mới, họ sẽ chèn nó vào đúng vị trí trong nhóm bài đã sắp xếp. Tương tự, để sắp xếp một mảng $x[n]$ theo thứ tự tăng dần, ta bắt đầu bằng cách xem phần tử đầu tiên ($x[0]$) là một mảng con đã được sắp xếp. Sau đó, lần lượt lấy từng phần tử từ $x[1]$ đến $x[n - 1]$ và chèn chúng vào đúng vị trí trong phần mảng đã sắp xếp. Quá trình này lặp lại cho đến khi toàn bộ mảng được sắp xếp.

Các bước của thuật toán:

INSERTION SORT(A)

1. for $j = 1$ to $A.length - 1$
2. key = $A[j]$
3. //Insert $A[j]$ into the sorted sequence $A[1..j - 1]$
4. $i = j - 1$
5. while $i > 0$ and $A[i] > key$
6. $A[i + 1] = A[i]$
7. $i = i - 1$
8. $A[i + 1] = key$

Hình 5: Mã giả Insertion Sort

Bước 1: Khởi tạo mảng con đã sắp xếp

Bắt đầu với phần tử đầu tiên của mảng (tại chỉ số 1 nếu đánh số từ 1 hoặc chỉ số 0 nếu đánh số từ 0). Xem phần tử này như một mảng con đã được sắp xếp, vì một phần tử đơn lẻ luôn ở đúng vị trí.

Bước 2: Chèn phần tử vào đúng vị trí trong mảng đã sắp xếp

Bước 2.1: Lấy phần tử tiếp theo trong mảng (ví dụ: $A[2]$ nếu bắt đầu từ chỉ số 1, hoặc $A[1]$ nếu bắt đầu từ chỉ số 0) và xem nó như phần tử cần được chèn (key).

Bước 2.2: So sánh key với các phần tử trong mảng con đã sắp xếp (bắt đầu từ cuối mảng con).

- Nếu key nhỏ hơn phần tử hiện tại (ví dụ: $A[j]$), dịch chuyển phần tử lớn hơn ($A[j]$) sang phải để tạo khoảng trống.
- Tiếp tục so sánh với các phần tử phía trước cho đến khi tìm được vị trí thích hợp.

Bước 2.3: Khi tìm được vị trí thích hợp (tức là không còn phần tử nào lớn hơn key), chèn key vào vị trí đó.

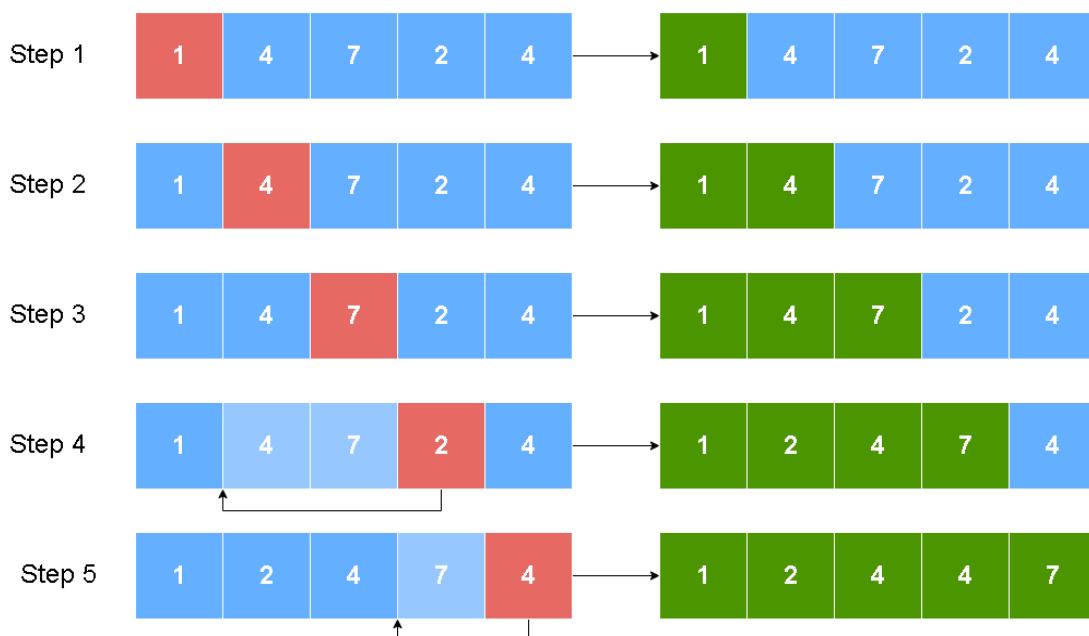
Bước 3: Tiếp tục với các phần tử còn lại

Lặp lại Bước 2 cho từng phần tử tiếp theo trong mảng (từ $A[3]$ đến $A[n]$ nếu đánh số từ 1, hoặc từ $A[2]$ đến $A[n - 1]$ nếu đánh số từ 0). Duy trì mảng con đã sắp xếp mở rộng từ trái sang phải.

Bước 4: Hoàn thành việc sắp xếp

Khi tất cả các phần tử trong mảng đã được xử lý và chèn vào đúng vị trí, toàn bộ mảng sẽ trở thành mảng sắp xếp theo thứ tự tăng dần.

Ví dụ minh họa:



Hình 6: Ví dụ Insertion Sort

Bước 1:

Tạo mảng ban đầu: [1, 4, 7, 2, 4].

Xem phần tử đầu tiên [1] đã được sắp xếp.

Bước 2:

Chọn phần tử thứ hai $key = 4$.

So sánh item với phần tử đầu tiên 1: do $4 > 1$, mảng không thay đổi.

Di chuyển key sang phần tử tiếp theo.

Bước 3:

Chọn $key = 7$.

So sánh key với các phần tử đã được sắp xếp trước đó (1 và 4): do $7 > 4$, mảng giữ nguyên.

Di chuyển key sang phần tử tiếp theo.

Bước 4:

Chọn $key = 2$.

So sánh key với các phần tử đã sắp xếp (1, 4, 7):

$2 < 7$: dịch chuyển 7 sang phải.

$2 < 4$: tiếp tục dịch chuyển 4 sang phải.

Chèn $key = 2$ vào đúng vị trí sau 1.

Mảng sau bước này: [1, 2, 4, 7, 4].

Bước 5:

Chọn $key = 4$ (phần tử cuối).

So sánh key với các phần tử đã sắp xếp (1, 2, 4, 7):

$4 < 7$: dịch chuyển 7 sang phải.

$4 = 4$: không cần dịch chuyển thêm.

Chèn key = 4 vào đúng vị trí sau 4.

Mảng cuối cùng sau khi sắp xếp: [1, 2, 4, 4, 7].

Đánh giá độ phức tạp thuật toán:

Phức tạp thời gian:

Trường hợp tốt nhất:

Khi mảng đã được sắp xếp sẵn, mỗi lần chèn một phần tử, thuật toán chỉ cần thực hiện một phép so sánh nên tổng số phép so sánh là $O(n)$ và không cần dịch chuyển phần tử nào vì mỗi phần tử ở đúng vị trí nên có $O(1)$ phép dịch chuyển. Trong trường hợp này, độ phức tạp thời gian là $O(n)$, với n là số lượng phần tử trong mảng.

Trường hợp trung bình:

Trong trường hợp trung bình, mỗi phần tử cần thực hiện khoảng $n/2$ phép so sánh và dịch chuyển $n/2$ phần tử để chèn vào đúng vị trí. Tổng số phép so sánh và dịch chuyển cho tất cả các phần tử trong mảng là $O(n^2)$. Vì vậy, độ phức tạp thời gian trong trường hợp trung bình của thuật toán Insertion Sort là $O(n^2)$.

Trường hợp xấu nhất:

Trong trường hợp xấu nhất của thuật toán sắp xếp chèn (Insertion Sort), khi mảng được sắp xếp theo thứ tự giảm dần, mỗi phần tử cần phải so sánh và dịch chuyển qua tất cả các phần tử đã sắp xếp trước đó. Cụ thể, để chèn phần tử thứ i vào vị trí đúng, cần thực hiện i phép so sánh và i phép dịch chuyển. Do đó, tổng số phép so sánh và phép dịch chuyển trong trường hợp xấu nhất là tổng của dãy số từ 1 đến $n - 1$, tức là $(n - 1) * n / 2$, dẫn đến độ phức tạp thời gian là $O(n^2)$.

Tóm lại: Trong trường hợp xấu và trung bình thì độ phức tạp của thuật toán sắp xếp chèn (Insertion Sort) là $O(n^2)$, và trong trường hợp tốt nhất có độ phức tạp là $O(n)$.

Phức tạp không gian:

Insertion Sort là một thuật toán sắp xếp tại chỗ (in-place sorting algorithm), nghĩa là nó không cần sử dụng bộ nhớ phụ (ngoài bộ nhớ để lưu trữ mảng ban đầu). Do đó, độ phức tạp không gian của thuật toán là $O(1)$.

Tóm lại: Độ phức tạp không gian của Insertion Sort là $O(1)$, vì thuật toán không yêu cầu bộ nhớ ngoài mảng ban đầu.

3.3. Bubble Sort

Ý tưởng chung của thuật toán:

Bubble Sort thuộc nhóm các thuật toán sắp xếp so sánh. Đây là kỹ thuật sắp xếp đơn giản nhất, với quá trình hoán đổi nhiều lần trong mỗi lượt, trong đó phần tử nhỏ nhất sẽ được “nổi bọt” lên đầu danh sách. Trong phương pháp sắp xếp này, các phần tử liền

kè trong danh sách cần sắp xếp sẽ được so sánh. Nếu phần tử phía trên lớn hơn phần tử ngay bên dưới, hai phần tử này sẽ được hoán đổi vị trí.

Các bước của thuật toán:

BUBBLE SORT

1. for $i = 0$ to $n - 1$ do
2. for $j = 0$ to $n - i - 1$ do
3. if $A[j] > A[j + 1]$ then
4. Swap($A[j], A[j + 1]$)

Hình 7: Mã giả Bubble Sort

Bước 1 (so sánh và hoán đổi):

Bắt đầu từ chỉ số đầu tiên, so sánh phần tử thứ nhất và phần tử thứ hai.

Nếu phần tử thứ nhất lớn hơn phần tử thứ hai, hoán đổi vị trí của chúng.

Tiếp theo, so sánh phần tử thứ hai và phần tử thứ ba. Nếu chúng không theo thứ tự, hoán đổi chúng.

Quá trình trên tiếp tục cho đến phần tử cuối cùng.

Các bước còn lại:

Quá trình tương tự được lặp lại cho các lượt tiếp theo.

Sau mỗi lượt, phần tử lớn nhất trong số các phần tử chưa được sắp xếp sẽ được đưa vào vị trí cuối cùng.

Trong mỗi lượt, việc so sánh chỉ diễn ra đến phần tử cuối cùng chưa được sắp xếp.

Mảng sẽ được sắp xếp khi tất cả các phần tử chưa được sắp xếp đã được đưa vào đúng vị trí của chúng.

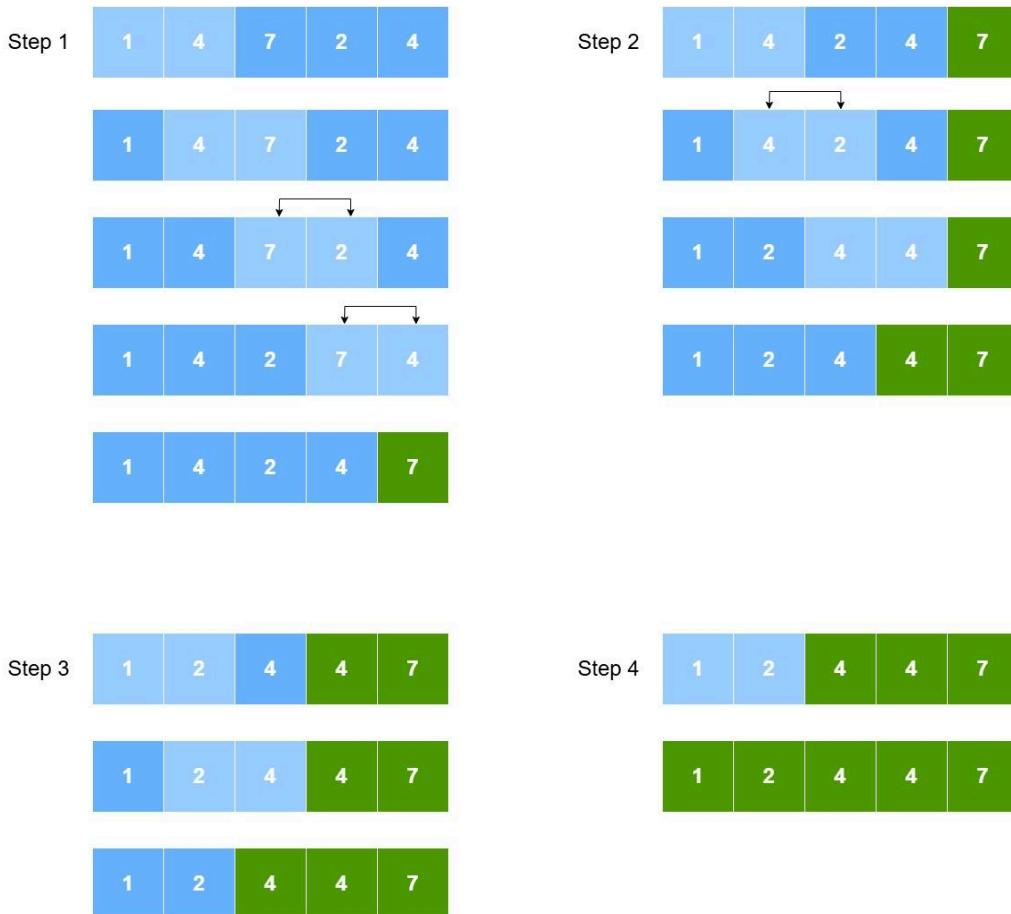
Tối ưu hóa thuật toán:

Trong thuật toán Bubble Sort thông thường, tất cả các phép so sánh đều được thực hiện ngay cả khi mảng đã được sắp xếp. Điều này làm tăng thời gian thực thi. Để giải quyết vấn đề này, chúng ta có thể sử dụng một biến phụ (hay cờ đánh dấu) **swapped**.

- Nếu xảy ra hoán đổi giữa các phần tử, **swapped = true**.
- Ngược lại, nếu không có hoán đổi nào xảy ra, **swapped = false**.

Sau mỗi vòng lặp, nếu giá trị của **swapped** là false, điều này có nghĩa là các phần tử đã được sắp xếp. Do đó, không cần thực hiện thêm các vòng lặp. Cách này giúp giảm thời gian thực thi và tối ưu hóa thuật toán Bubble Sort.

Ví dụ minh họa:



Hình 8: Ví dụ Bubble Sort

Bước 1: Ở lần duyệt đầu tiên, so sánh từng cặp phần tử liên tiếp

- So sánh 1 và 4 (không hoán vị), 4 và 7 (không hoán vị), 7 và 2 (hoán vị), 7 và 4 (hoán vị).
- Sau bước 1, phần tử lớn nhất là 7 được đưa về cuối.

Bước 2: Tiếp tục so sánh từng cặp phần tử (không xét phần tử 7 vì đã đúng vị trí)

- So sánh 1 và 4 (không hoán vị), 4 và 2 (hoán vị), 4 và 4 (không hoán vị).
- Sau bước 2, phần tử lớn thứ hai là 4 được đưa về đúng vị trí.

Bước 3: Tiếp tục so sánh từng cặp phần tử (không xét phần tử 4, 7)

- So sánh 1 và 2 (không hoán vị), 2 và 4 (không hoán vị).
- Sau bước 3, không có thay đổi nào xảy ra, phần tử lớn thứ ba là 4 đã đúng đúng vị trí.

Bước 4: Tiếp tục so sánh từng cặp phần tử (không xét hai phần tử 4, và phần tử 7)

- So sánh 1 và 2 (không hoán vị).

- Sau bước 4, không có thay đổi nào xảy ra, các phần tử đều đã được sắp xếp đúng vị trí.

Đánh giá độ phức tạp thuật toán:

Phức tạp thời gian:

Trường hợp tốt nhất: $O(n)$

Xảy ra khi mảng đã được sắp xếp sẵn theo thứ tự tăng dần.

Trường hợp trung bình: $O(n^2)$

Xảy ra khi các phần tử trong mảng có thứ tự lộn xộn.

Trường hợp xấu nhất: $O(n^2)$

Xảy ra khi chúng ta cần sắp xếp mảng theo thứ tự tăng dần nhưng mảng ban đầu lại được sắp xếp theo thứ tự giảm dần.

Phức tạp không gian:

Độ phức tạp không gian là $O(1)$ vì chỉ sử dụng thêm một biến phụ để hoán đổi giá trị. Trong thuật toán Bubble Sort tối ưu, có sử dụng thêm hai biến phụ (hoán đổi giá trị và cờ đánh dấu). Vì vậy, độ phức tạp không gian sẽ là $O(2)$.

3.4. Shaker Sort

Ý tưởng chung của thuật toán:

Shaker Sort là cải tiến của Bubble Sort mà trong đó việc duyệt mảng diễn ra theo cả hai chiều (từ trái sang phải và từ phải sang trái). Điều này giúp thuật toán cải thiện hiệu suất bằng cách đưa giá trị lớn nhất về cuối mảng và giá trị nhỏ nhất về đầu mảng trong mỗi lượt duyệt. Thuật toán sẽ ghi nhớ vị trí cuối cùng xảy ra đổi chỗ trong mỗi lượt duyệt, giúp giảm phạm vi xử lý ở các bước tiếp theo và tăng tốc độ sắp xếp.

Các bước của thuật toán:

```

procedure cocktailShakerSort(A : list of sortable items) is
    do
        swapped := false
        for each i in 0 to length(A) - 1 do:
            if A[i] > A[i + 1] then
                swap(A[i], A[i + 1])
                swapped := true
            end if
        end for
        if not swapped then
            break do-while loop
        end if
        swapped := false
        for each i in length(A) - 1 to 0 do:
            if A[i] > A[i + 1] then
                swap(A[i], A[i + 1])
            end if
        end for
    end do
end procedure

```

```

swapped := true
end if
end for
while swapped
end procedure

```

Hình 9: Mã giả Shaker Sort (Hình 1)

Từ mã giả trên cùng các thông tin và nguồn code nhóm chúng em đã tham khảo được. Chúng em đã đưa ra cách triển khai cuối cùng cho thuật toán Shaker Sort như mã giả bên dưới minh họa.

SHAKER SORT(A)

1. $left = 0, right = n - 1$
2. while $left < right$:
 3. for $i = left$ to $right - 1$:
 5. if $A[i] > A[i + 1]$: swap($A[i], A[i + 1]$); $k = i$
 6. $right = k$
 7. for $i = right$ to $left + 1$:
 8. if $A[i] < A[i - 1]$: swap($A[i], A[i - 1]$); $k = i$
 9. $left = k$

Hình 10: Mã giả Shaker Sort (Hình 2)**Bước 1: Khởi tạo**

$left = 0, right = n - 1, k = 0;$

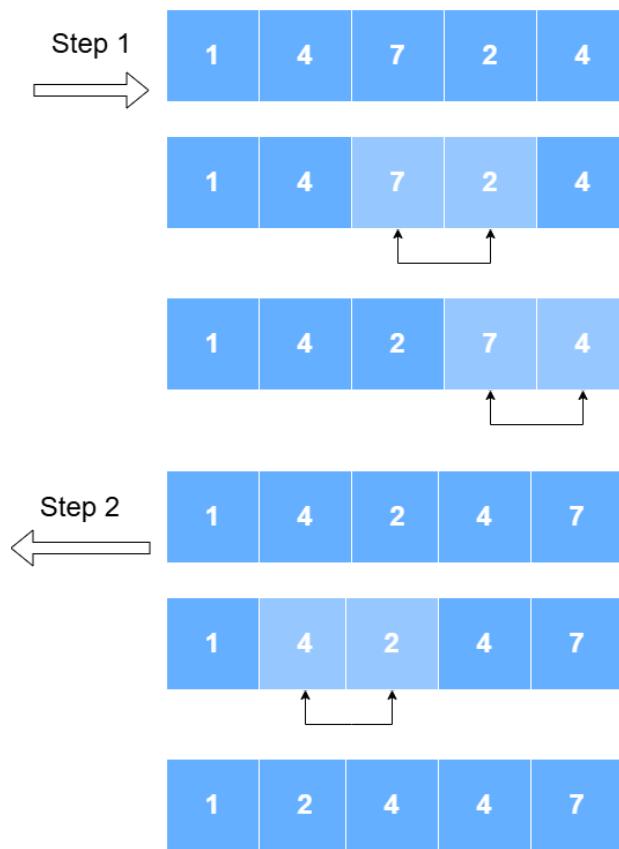
left, right, k lần lượt là vị trí đầu, vị trí cuối và vị trí gần nhất có sự thay đổi trong quá trình duyệt mảng

Bước 2: Vòng lặp chính: Điều kiện lặp ($left < right$)

- **Bước 2.1.** Duyệt từ đầu đến cuối dãy :
 - Tại mỗi bước, nếu phần tử hiện tại lớn hơn phần tử tiếp theo, ta sẽ hoán đổi hai phần tử (swap).
 - Cập nhật chỉ số k sau mỗi lần hoán đổi, k sẽ lưu chỉ số cuối cùng mà có sự hoán đổi.
- **Bước 2.2.** Gán $right = k$ để thu hẹp phạm vi duyệt xuống.
- **Bước 2.3.** Duyệt qua mảng từ chỉ số right đến left + 1
 - Tại mỗi bước, nếu phần tử hiện tại lớn hơn phần tử tiếp theo, ta sẽ hoán đổi hai phần tử (swap).
 - Cập nhật chỉ số k sau mỗi lần hoán đổi.
- **Bước 2.4.** Gán $left = k$ để thu hẹp phạm vi duyệt.

Bước 3: Vòng lặp ngoài kết thúc khi $left > right$.

Ví dụ minh họa:



Hình 11: Ví dụ Shaker Sort

Chi tiết cách sắp xếp:

Mảng ban đầu: [1, 4, 7, 2, 4].

Bước 1: Khởi tạo $left = 0$ (vị trí đầu mảng), $right = n - 1 = 4$ (vị trí cuối mảng), $k = 0$.

Bước 2: Vòng lặp chính (điều kiện lặp là $left < right$).

- **Bước 2.1:** Duyệt từ đầu đến cuối dãy (từ $left$ đến $right$):
 - **Mảng ban đầu:** [1, 4, 7, 2, 4]
 - So sánh $1 < 4 \rightarrow$ Không hoán đổi.
 - So sánh $4 < 7 \rightarrow$ Không hoán đổi.
 - So sánh $7 > 2 \rightarrow$ Hoán đổi \rightarrow [1, 4, 2, 7, 4], cập nhật $k = 3$.
 - So sánh $7 > 4 \rightarrow$ Hoán đổi \rightarrow [1, 4, 2, 4, 7], cập nhật $k = 4$.
- **Bước 2.2:** Gán $right = k$ ($right = 4$).
- **Bước 2.3:** Duyệt qua mảng từ phải sang trái (từ $right$ đến $left + 1$):
 - So sánh $4 > 2 \rightarrow$ Hoán đổi \rightarrow [1, 2, 4, 4, 7], cập nhật $k = 2$.

- So sánh 4 và 1 → Không hoán đổi.
- **Bước 2.4:** Gán $left = k$
 - $left = 2$. (Tại thời điểm này, $left = 2$ và $right = 4$, nên phải tiếp tục duyệt mảng cho đến khi $left$ vượt qua hoặc bằng $right$. Do lúc này mảng đã được sắp xếp, nên thêm 1 lần lặp thì $left = right = k$, vòng lặp sẽ kết thúc).

Bước 3: Kết thúc vòng lặp ngoài.

Đánh giá độ phức tạp thuật toán:

Phức tạp thời gian:

Trường hợp tốt nhất: $O(n)$

Tốt nhất là trong trường hợp mảng đã được sắp xếp, thuật toán chỉ cần duyệt qua mảng một lần

Trường hợp trung bình: $O(n^2)$

Xảy ra khi các phần tử trong mảng có thứ tự lộn xộn.

Trường hợp xấu nhất: $O(n^2)$

Trong trường hợp xấu nhất, mảng nghịch đảo, thuật toán sẽ phải duyệt qua toàn bộ mảng trong mỗi lần từ trái sang phải và từ phải sang trái. Mỗi lần duyệt có thể dẫn đến một hoán đổi, và quá trình này sẽ lặp lại cho đến khi mảng được sắp xếp hoàn toàn. Vì vậy, độ phức tạp thời gian là $O(n^2)$.

Phức tạp không gian: $O(1)$

Đoạn mã Shaker Sort sử dụng một số biến phụ ($left$, $right$, k) để theo dõi phạm vi sắp xếp cùng vị trí hoán đổi, và chỉ làm việc trực tiếp với mảng đầu vào mà không tạo ra mảng phụ. Vì vậy, độ phức tạp không gian của thuật toán là $O(1)$.

3.5. Shell Sort

Ý tưởng chung của thuật toán:

Ý tưởng của ShellSort là cho phép trao đổi các phần tử nằm cách xa nhau, nhằm giảm số lần di chuyển cần thiết so với Insertion Sort. Trong ShellSort, mảng được sắp xếp sao cho nó trở thành h-sorted với một giá trị lớn của h . Sau đó, giá trị h được giảm dần cho đến khi $h = 1$. Một mảng được gọi là h-sorted nếu tất cả các nhóm con gồm các phần tử cách nhau h vị trí đều đã được sắp xếp.

Các bước của thuật toán:

SHELL SORT(A,I,n,h,l)

1. for $I = n$ to $i/2$ (for pass)
2. for $I = h$ to n
3. for $j = 1$ to $j = j - h$ upto $j \geq h \&& k < A[j - h]$
4. assign $A[j - h]$ to $A[j]$

Hình 12: Mã giả Shell Sort (Hình 1)

Từ mã giả trên cùng các thông tin và nguồn code nhóm chúng em đã tham khảo được. Chúng em đã đưa ra cách triển khai cuối cùng cho thuật toán Shell Sort như mã giả bên dưới minh họa.

SHELL SORT(A, n, gap)

1. $gap = n/2$
2. for $gap = n/2$; $gap \geq 1$; $gap = n/2$
3. for $i = gap$ to $n - 1$
4. $temp = A[i]$
5. $j = i$
6. while $j \geq gap$ and $A[j - gap] > temp$
7. $A[j] = A[j - gap]$
8. $j = j - gap$
9. $A[j] = temp$

Hình 13: Mã giả Shell Sort (Hình 2)

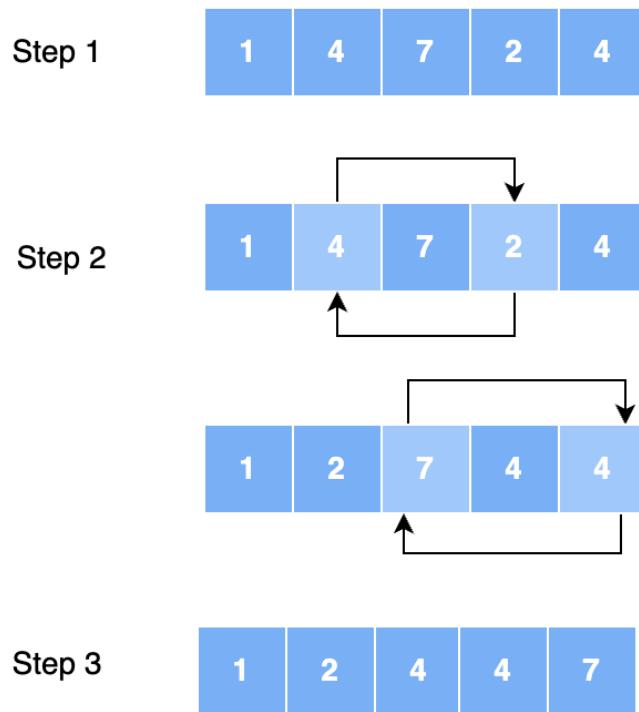
Bước 1: Khởi tạo giá trị của kích thước khoảng cách, ví dụ gap.

Bước 2: Chia danh sách thành các phần nhỏ hơn. Mỗi phần phải có khoảng cách bằng nhau đến gấp.

Bước 3: Sắp xếp các danh sách con này bằng cách sử dụng sắp xếp chèn.

Bước 4: Lặp lại bước 1 này cho đến khi danh sách được sắp xếp.

Ví dụ minh họa:

**Hình 14: Ví dụ Shell Sort**

Chi tiết cách làm:

Bước 1: Từ mảng a đề cho với số lượng phần tử là $n = 5$, ta tính khoảng cách $h = n/2 = 2$ (làm tròn xuống).

Bước 2: Lần lượt so sánh các phần tử cách nhau khoảng h với nhau.

So sánh $a[0]$ với $a[2]$: $1 < 7$ nên giữ nguyên vị trí.

So sánh $a[1]$ với $a[3]$: $4 > 2$ nên đổi chỗ $a[1]$ và $a[3]$.

So sánh $a[2]$ với $a[4]$: $7 > 4$ nên đổi chỗ $a[2]$ và $a[4]$.

Bước 3: Khi đã một lượt chạy qua hết mảng ta giảm $h = h/2 = 2/2 = 1$.

Lúc này ta so sánh các phần tử kế nhau với nhau: thấy đã đúng thứ tự nên giữ nguyên vị trí.

(Nếu mảng đề cho có nhiều phần tử thì mỗi lần chạy qua một lượt ta giảm h rồi thực hiện bước 2 giảm cho đến khi $h = 1$ rồi so sánh các phần tử cạnh nhau đến khi thu được mảng sắp xếp hoàn chỉnh thì dừng lại).

Đánh giá độ phức tạp thuật toán:

Phức tạp thời gian:

Tốt nhất: $\Omega(n \log n)$

Khi danh sách mảng đã cho đã được sắp xếp, tổng số phép so sánh của mỗi khoảng bằng với kích thước của mảng đã cho.

Trung bình: $O(n * \log n)$

Xấu nhất: $O(n^2)$

Khi khoảng cách không được chọn một cách tối ưu và mảng đầu vào có cấu trúc kém dẫn đến nhiều lần hoán đổi và so sánh.

Phức tạp không gian: $O(1)$

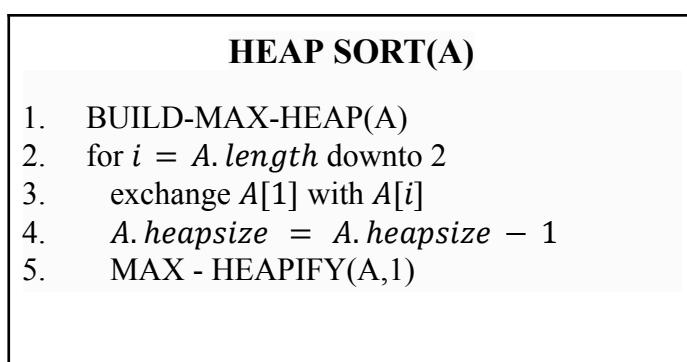
Do thuật toán chỉ sử dụng một lượng bộ nhớ không đổi và không phụ thuộc vào kích thước đầu vào.

3.6. Heap Sort

Ý tưởng chung của thuật toán:

Heap Sort là một thuật toán sắp xếp dựa trên so sánh (comparison-based sorting algorithm), sử dụng cấu trúc dữ liệu **heap nhị phân** để sắp xếp các phần tử một cách hiệu quả. Thuật toán này hoạt động bằng cách xây dựng một heap từ dữ liệu đầu vào, sau đó liên tục trích xuất phần tử lớn nhất (hoặc nhỏ nhất) từ heap và đưa nó vào đúng vị trí trong mảng đã sắp xếp.[3]

Các bước của thuật toán:[3]



Hình 15: Mã giả Heap Sort

Bước 1: Xây dựng Max-Heap từ mảng đầu vào

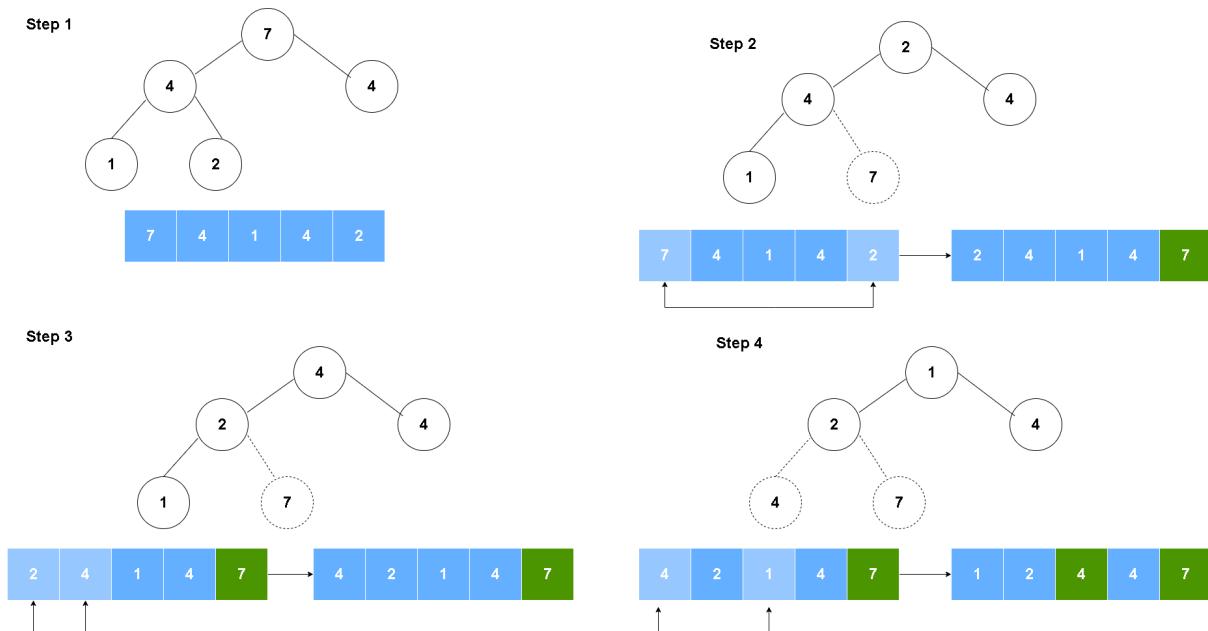
Đầu tiên, chúng ta chuyển mảng đầu vào thành một Max-Heap. Điều này có nghĩa là, đối với mỗi phần tử trong mảng, ta đảm bảo rằng nó lớn hơn hoặc bằng hai phần tử con của nó.

Bước 2: Trích xuất phần tử lớn nhất và sắp xếp

Sau khi mảng đã trở thành Max-Heap, phần tử lớn nhất (ở chỉ số 0) sẽ được chuyển về cuối mảng, sau đó heapify lại phần còn lại của mảng.

- Bước 2.1: Hoán đổi phần tử gốc với phần tử cuối cùng.
- Hoán đổi phần tử gốc (phần tử lớn nhất) với phần tử cuối cùng trong mảng (ở chỉ số i). Sau khi hoán đổi, phần tử lớn nhất đã được đặt vào cuối mảng, không còn là một phần của heap nữa.
- Bước 2.2: Giảm kích thước của heap.
- Sau khi hoán đổi, giảm kích thước của heap xuống một đơn vị (heapsize--), để không tính phần tử đã được hoán đổi vào trong heap nữa.
- Bước 2.3: Thực hiện lại thao tác "heapify".
- Gọi hàm maxHeapify(arr, 0) để đảm bảo rằng heap vẫn duy trì tính chất Max-Heap cho phần còn lại của mảng. Phần tử lớn nhất sẽ được đưa lên gốc của heap một lần nữa.
- Bước 2.4: Lặp lại bước trên.
- Tiếp tục lặp lại các bước hoán đổi, giảm kích thước heap, và heapify cho đến khi toàn bộ mảng được sắp xếp. Khi heap chỉ còn một phần tử, mảng đã được sắp xếp hoàn chỉnh.

Ví dụ minh họa:



Hình 16: Ví dụ Heap Sort

Chi tiết cách làm:

Bước 1: Xây dựng Max Heap

Biến đổi mảng ban đầu [1, 4, 7, 2, 4] thành **Max Heap**.

Sau khi "heapify", mảng trở thành [7, 4, 1, 4, 2], trong đó 7 là gốc (phần tử lớn nhất).

Bước 2: Hoán đổi gốc và phần tử cuối

Hoán đổi phần tử gốc (7) với phần tử cuối cùng (2).

Sau khi hoán đổi: [2, 4, 1, 4, 7].

Giảm kích thước heap đi 1 đơn vị (coi như phần tử 7 đã được sắp xếp).

Bước 3: "Heapify" lại phần còn lại

Gốc mới là 2. Sau khi "heapify", gốc trở thành 4.

Mảng sau khi "heapify": [4, 4, 1, 2, 7].

Bước 4: Lặp lại quá trình hoán đổi và "heapify"

Hoán đổi gốc (4) với phần tử cuối của heap (2).

Sau khi hoán đổi: [2, 4, 1, 4, 7].

Giảm kích thước heap đi 1 đơn vị (coi như phần tử 4 đã được sắp xếp).

Tiếp tục "heapify" phần còn lại để duy trì Max Heap.

Tiếp tục lặp lại cho đến khi heap chỉ còn 1 phần tử:

Mỗi lần hoán đổi phần tử lớn nhất (gốc) ra cuối mảng và "heapify" phần còn lại.

Mảng sau khi hoàn tất:

Mảng được sắp xếp thành [1, 2, 4, 4, 7].

Đánh giá độ phức tạp toán:

Phức tạp thời gian:

Trường hợp tốt nhất:

Khi mà mảng đã được xếp theo thứ tự tăng dần, tuy nhiên vẫn phải xây dựng cấu trúc heap và việc tái sắp xếp vẫn phải được diễn ra do đó độ phức tạp toán là $O(n \log n)$.

Trường hợp trung bình:

Khi mảng được sắp xếp một cách ngẫu nhiên, thuật toán phải xây dựng cấu trúc heap từ đầu và sau đó tiếp tục thực hiện thao tác “heapify” (tái cấu trúc heap) sau mỗi lần hoán đổi. Vì quá trình xây dựng heap ban đầu có độ phức tạp $O(n)$, và mỗi lần trích xuất phần tử (hoán đổi và tái cấu trúc heap) có độ phức tạp $O(\log n)$, nên tổng độ phức tạp cho trường hợp trung bình vẫn là $O(n \log n)$.

Trường hợp xấu nhất:

Trường hợp xấu nhất xảy ra khi mảng đã được sắp xếp theo thứ tự ngược (giảm dần). Trong trường hợp này, thuật toán vẫn phải thực hiện tất cả các thao tác “heapify” để duy trì cấu trúc heap sau mỗi lần hoán đổi. Do đó, độ phức tạp thời gian vẫn là $O(n \log n)$.

Tóm lại: trong trường hợp tệ nhất, trung bình và tốt nhất thì thuật toán sắp xếp vun đống (Heap sort) có độ phức tạp là $O(n \log n)$.

Phức tạp không gian:

Heap Sort là một thuật toán **in-place**, không sử dụng không gian phụ đáng kể (ngoài ngăn xếp đệ quy). Trong trường hợp triển khai đệ quy, độ sâu ngăn xếp tối đa là $O(\log n)$. Nếu triển khai dạng lặp, phức tạp không gian chỉ là $O(1)$.

3.7. Merge Sort

Ý tưởng chung của thuật toán:

Phân hoạch dãy ban đầu thành các dãy con. Sau khi phân hoạch xong, dãy ban đầu sẽ được tách ra thành 2 dãy phụ theo nguyên tắc phân phối đều luân phiên. Trộn từng cặp dãy con của hai dãy phụ thành một dãy con của dãy ban đầu, ta sẽ nhận lại dãy ban đầu nhưng với số lượng dãy con ít nhất giảm đi một nửa. Lặp lại quy trình trên sau một số bước, ta sẽ nhận được 1 dãy chỉ gồm 1 dãy con không giảm. Nghĩa là dãy ban đầu đã được sắp xếp.

Các bước của thuật toán:

MergeSort(A , B, C, n)

1. $k = 1$
2. while $k < n$ do
3. $p = pb = pc = 0$ //Chỉ số các chỉ số trên mảng A, B, C
4. while $p < n$
5. split A into B and C by principle of interleaving
6. mix B and C to A
7. $k *= 2$

Hình 17: Mã giả Merge Sort

Bước 1:

$k = 1$; // k là chiều dài của dãy con trong bước hiện hành.

Bước 2:

Tách dãy a_1, a_2, \dots, a_n thành 2 dãy b,c theo nguyên tắc luân phiên từng nhóm k phần tử:

$$b = a_1, \dots, a_k, a_{2k+1}, \dots, a_{3k}, \dots$$

$$c = a_{k+1}, \dots, a_{2k}, a_{3k+1}, \dots, a_{4k}, \dots$$

Bước 3:

Trộn từng cặp dãy con gồm k phần tử của 2 dãy b, c vào a.

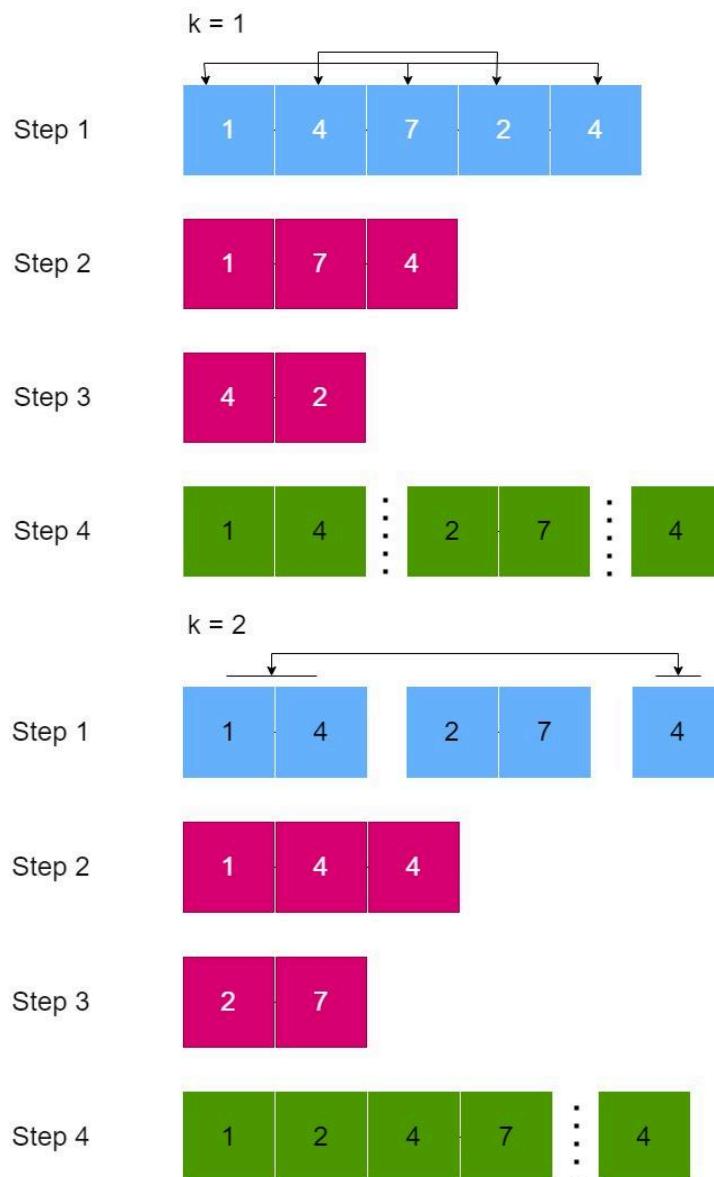
Bước 4:

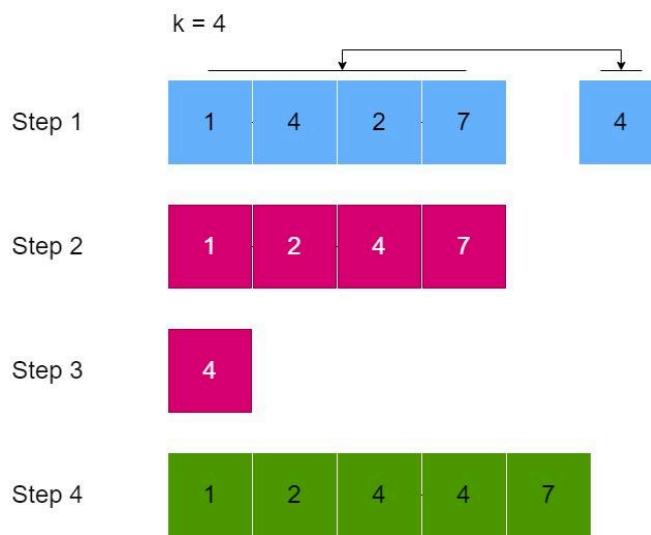
$$k = k * 2;$$

Nếu $k < n$ thì trở lại bước 2.

Ngược lại: Dừng.

Ví dụ minh họa:



**Hình 18: Ví dụ minh họa Merge Sort****Chi tiết cách bước sắp xếp:****Bước 1:** Chiều dài dãy con $k = 1$.

- Tách mảng $A[1,4,7,2,4]$ thành 2 mảng theo nguyên tắc luân phiên. Ta được 2 mảng:
 - + $B[1, 7, 4]$
 - + $C[4, 2]$

Bước 2: Trộn từng cặp dãy con gồm $k = 1$ phần tử của 2 mảng B,C vào A. Ta được:

- $A[1, 4, 2, 7, 4]$

Bước 3: Chiều dài dãy con $k = 2$

- Tách mảng $A[1, 4, 2, 7, 4]$ thành 2 mảng theo nguyên tắc luân phiên. Ta được 2 mảng:
 - + $B[1, 4, 4]$
 - + $C[2, 7]$

Bước 4: Trộn từng cặp dãy con gồm $k = 2$ phần tử của 2 mảng B,C vào A. Ta được:

- $A[1, 2, 4, 7, 4]$

Bước 5: Chiều dài dãy con là $k = 4$

- Tách mảng $A[1, 2, 4, 7, 4]$ thành 2 mảng theo nguyên tắc luân phiên. Ta được 2 mảng:
 - + $B[1, 2, 4, 7]$
 - + $C[4]$

Bước 6: Trộn từng cặp dãy con gồm $k = 4$ phần tử của 2 mảng B,C vào A. Ta được:

- $A[1, 2, 4, 4, 7]$
 \Rightarrow Mảng đã được sắp xếp.

Đánh giá độ phức tạp thuật toán:

Phức tạp thời gian:

- **Trường hợp tốt nhất:** $O(n \log n)$ khi mảng được sắp xếp hoặc gần được sắp xếp.
- **Trường hợp trung bình:** $O(n \log n)$ khi mảng được sắp xếp ngẫu nhiên.
- **Trường hợp xấu nhất:** $O(n^2)$ khi mảng được sắp xếp theo thứ tự ngược.

Phức tạp không gian: $O(n)$ cần có không gian bổ sung cho mảng tạm thời được sử dụng trong quá trình hợp nhất.

3.8. Quick Sort

Ý tưởng chung của thuật toán:

Quick Sort là một thuật toán sắp xếp tại chỗ (in-place), sử dụng chiến lược chia để trị (divide-and-conquer) và được thực hiện dưới dạng đệ quy sâu rộng. Thuật toán này còn được gọi là partition-exchange sort.

Chia (Divide):

- Chia danh sách thành hai phần bằng cách chọn một phần tử làm pivot (phần tử chốt).
- Một danh sách chứa tất cả các phần tử nhỏ hơn hoặc bằng phần tử chốt.
- Danh sách còn lại chứa tất cả các phần tử lớn hơn phần tử chốt.

Xử lý (Conquer):

- Tiếp tục thực hiện phân vùng trên hai danh sách vừa chia bằng cách áp dụng cùng một phương pháp (đệ quy).
- Quá trình này được lặp lại cho đến khi các danh sách trở nên nhỏ đến mức không cần sắp xếp thêm.

Kết hợp (Combine):

- Cuối cùng, các danh sách con đã được sắp xếp sẽ được ghép lại để tạo ra danh sách hoàn chỉnh đã được sắp xếp.

Các bước của thuật toán:

Quick sort (A, p, r)

1. If $p < r$
2. Then $q \leftarrow \text{partition}(A, p, r)$
3. Quick sort (A, p, q – 1)
4. Quick sort (A, q + 1, r)

Partition (A, p, r)

1. $x \leftarrow A[r]$
2. $i \leftarrow p - 1$
3. For $j \leftarrow p$ to $r - 1$
4. Do if $A[j] \leq x$
5. Then $i \leftarrow i + 1$
6. Exchange $A[i] \leftrightarrow A[j]$
7. Exchange $A[i + 1] \leftrightarrow A[r]$
8. Return $i + 1$

Hình 19: Mã giả Quick Sort (Hình 1)

Mã giả trên là Quick Sort với phần tử chốt ở cuối mảng. Đối với báo cáo này, chúng em đã triển khai thuật toán Quick Sort với phần tử chốt ở giữa mảng nên sẽ có thay đổi nhỏ về mã giả, dưới đây là mã giả minh họa Quick Sort với cách chọn phần tử chốt ở giữa mảng.

Quick sort (A, p, r)

1. If $p < r$
2. Then $q \leftarrow \text{partition}(A, p, r)$
3. Quick sort (A, p, q – 1)
4. Quick sort (A, q + 1, r)

Partition (A, p, r)

1. $x \leftarrow A[(p + r)/2]$
2. $i \leftarrow p$
3. $j \leftarrow r$
4. While $i \leq j$
5. While $A[i] < x$
6. $i \leftarrow i + 1$
7. While $A[j] > x$
8. $j \leftarrow j - 1$
9. If $i \leq j$
10. Exchange $A[i] \leftrightarrow A[j]$
11. $i \leftarrow i + 1$
12. $j \leftarrow j - 1$
13. Return i

Hình 20: Mã giả Quick Sort (Hình 2)**Bước 1:** Chọn phần tử chốt

Có nhiều biến thể của thuật toán Quick Sort dựa trên vị trí chọn phần tử chốt. Trong báo cáo này, chúng ta chọn phần tử giữa của mảng làm phần tử chốt.

Bước 2: Sắp xếp lại mảng

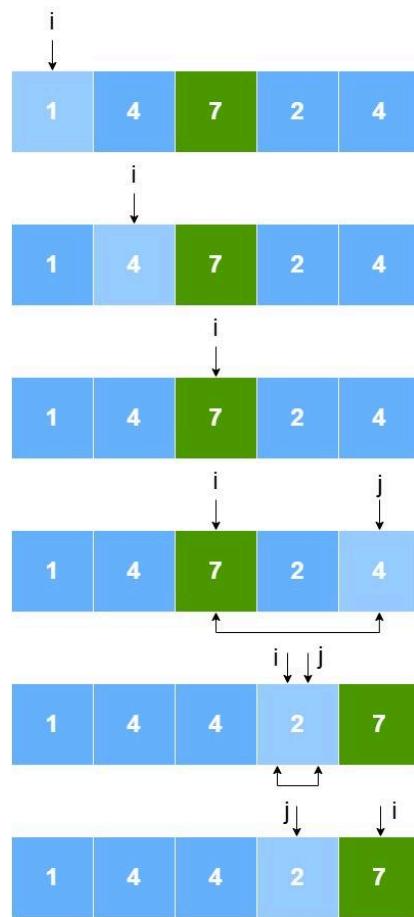
Các phần tử trong mảng được sắp xếp lại như sau (có hai con trỏ i và j được sử dụng):

- i bắt đầu từ đầu mảng, di chuyển sang phải để tìm phần tử không nhỏ hơn pivot.
- j bắt đầu từ cuối mảng, di chuyển sang trái để tìm phần tử không lớn hơn pivot.
- Khi cả i và j tìm được các phần tử vi phạm điều kiện, chúng được hoán đổi.
- Quá trình này tiếp tục cho đến khi hai con trỏ giao nhau.

Bước 3: Chia các mảng con

- Gọi đệ quy Quick Sort trên hai nửa mảng, tiếp tục lặp lại bước 1 và bước 2 cho đến khi mỗi mảng con chỉ còn một phần tử.
- Sau khi tất cả các mảng con được xử lý, mảng ban đầu sẽ được sắp xếp hoàn chỉnh.

Ví dụ minh họa:

**Hình 21: Ví dụ Quick Sort (Partition)**

Bước 1: Bắt đầu dịch chuyển con trỏ i từ đầu mảng để tìm vị trí nào vi phạm điều kiện (tức $A[i] >= 7$). Khi này $i = 2$ (ứng với số 7).

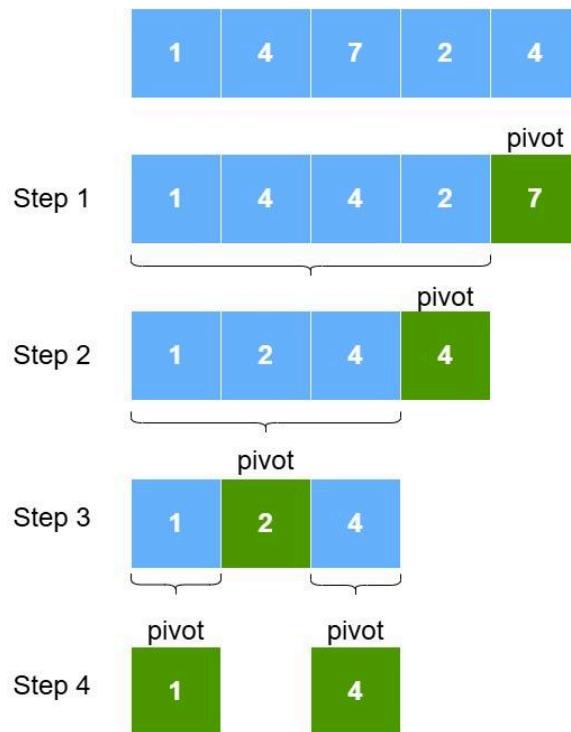
Bước 2: Bắt đầu dịch chuyển con trỏ j từ cuối mảng để tìm vị trí nào vi phạm điều kiện (tức $A[j] <= 7$). Khi này $j = 4$ (ứng với số 4).

Bước 3: Vì $i \leq j$, hoán vị hai phần tử ở này (số 7 và số 4). Đồng thời tăng $i = i + 1$, giảm $j = j - 1$.

Bước 4: Vì $i \leq j$, tiếp tục lặp lại bước 3.

Bước 5: Khi này $i > j$, mảng đã được phân chia thành mảng trái gồm [1, 4, 4, 2], phần tử chốt, và mảng phải lúc này rỗng.

Tiếp tục gọi đệ quy Quick Sort về hai mảng con trái và phải sau mỗi lần phân chia như hình minh họa bên dưới, ta thu được mảng đã được sắp xếp hoàn chỉnh.

**Hình 22: Ví dụ Quick Sort****Đánh giá độ phức tạp thuật toán:****Phức tạp thời gian:**

- **Trường hợp tốt nhất:** $O(n \log n)$.
Xảy ra khi phần tử chốt luôn là phần tử giữa hoặc gần với phần tử giữa.
Khi đó, mảng được chia thành hai mảng con có kích thước gần bằng nhau sau mỗi bước.
Điều này giảm tối đa số bước đệ quy, dẫn đến thời gian chạy tối ưu.
- **Trường hợp trung bình:** $O(n \log n)$.
Xảy ra khi phần tử chốt được chọn nằm ở các vị trí ngẫu nhiên trong mảng.
- **Trường hợp xấu nhất:** $O(n^2)$
Xảy ra khi phần tử chốt được chọn là phần tử lớn nhất hoặc nhỏ nhất trong mảng.
Khi đó, phần tử chốt nằm ở đầu hoặc cuối mảng đã được sắp xếp.
Một mảng con sẽ luôn rỗng, và mảng con còn lại chứa $n - 1$ phần tử.
Vì vậy, thuật toán Quick Sort chỉ được gọi cho mảng con này, dẫn đến số lần đệ quy tối đa.

Phức tạp không gian: $O(\log n)$.

3.9. Counting Sort

Ý tưởng chung của thuật toán:

Counting Sort không phải là một thuật toán sắp xếp dựa trên so sánh, vì chúng không sắp xếp các phần tử trong mảng bằng cách so sánh chúng với các phần tử khác. Ý tưởng cơ bản của Counting Sort là đếm xem mỗi giá trị xuất hiện bao nhiêu lần trong mảng, sau đó dùng thông tin này để sắp xếp các phần tử vào đúng vị trí của chúng trong mảng kết quả.

Các bước của thuật toán:

COUNTING SORT(A, B, k)

1. for $i = 0$ to k
2. $C[i] = 0$
3. for $j = 1$ to $A.length$
4. $C[A[j]] = C[A[j]] + 1$
5. for $i = 1$ to k
6. $C[i] = C[i] + C[i - 1]$
7. for $j = A.length$ down to 1
8. $B[C[A[j]]] = A[j]$
9. $C[A[j]] = C[A[j]] - 1$

Hình 23: Mã giả Counting Sort

Bước 1: Tìm phần tử lớn nhất (k):

Tìm phần tử lớn nhất trong mảng, gọi là k .

Bước 2: Khởi tạo mảng đếm:

Tạo mảng C có kích thước $k + 1$ và gán tất cả phần tử trong mảng này bằng 0. Mảng này dùng để lưu tần suất xuất hiện của mỗi giá trị trong mảng ban đầu.

Bước 3: Đếm tần suất các phần tử:

Duyệt qua mảng ban đầu và cập nhật tần suất xuất hiện của mỗi phần tử vào mảng C .

Ví dụ: nếu phần tử “3” xuất hiện 2 lần thì 2 được lưu trữ ở vị trí thứ 3 của mảng C . Nếu phần tử “5” không có trong mảng, thì số 0 sẽ được lưu trữ ở vị trí thứ 5.

Bước 4: Tính tổng tiền tố:

Lặp qua mảng C và thực hiện phép cộng dồn các phần tử. Điều này giúp mảng C lưu trữ vị trí chính xác của mỗi phần tử trong mảng đã sắp xếp.

Bước 5: Sắp xếp mảng:

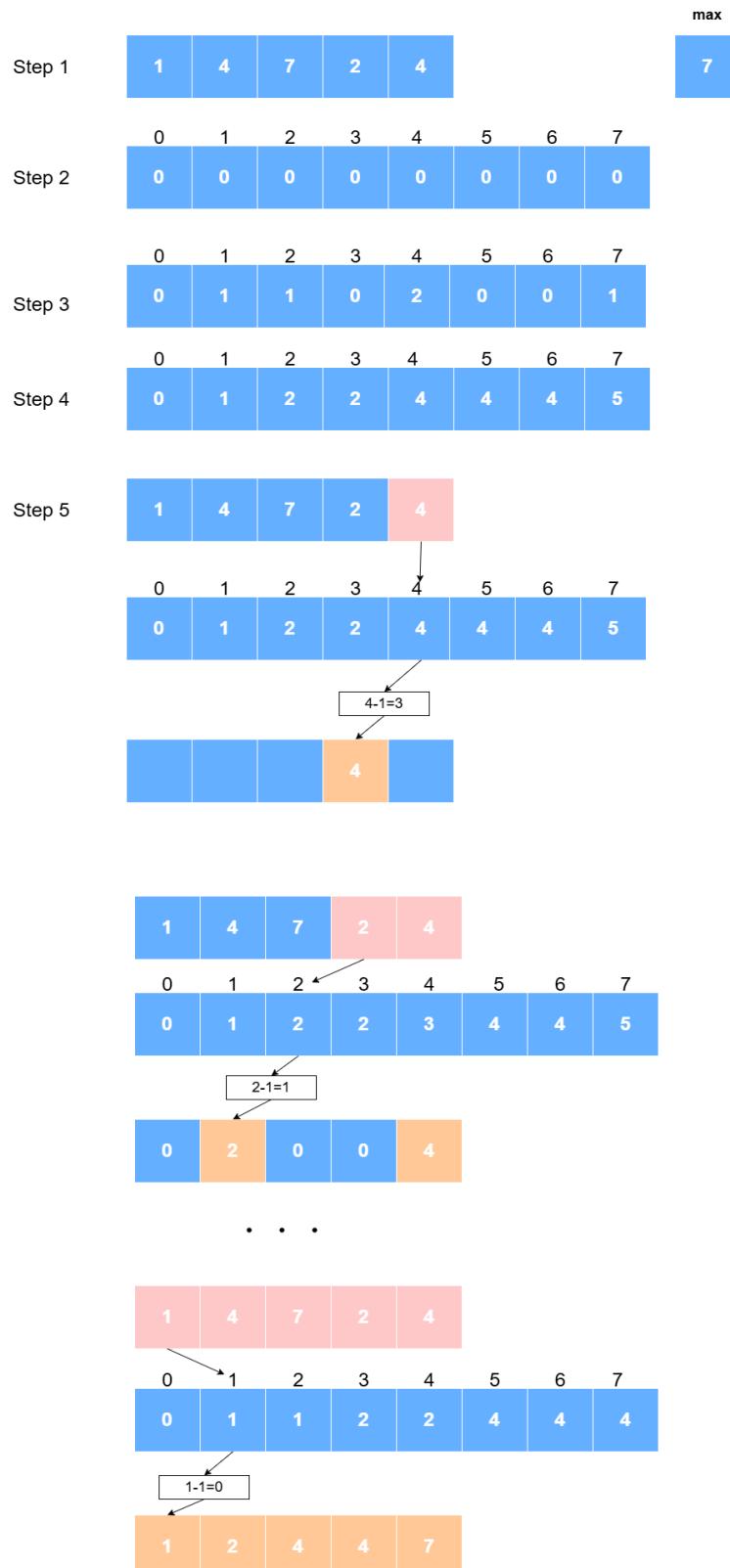
Tạo mảng B để lưu trữ mảng đã sắp xếp. Duyệt ngược qua mảng ban đầu (mảng A), sử dụng mảng C để xác định vị trí chính xác của mỗi phần tử trong mảng B.

Sau khi gán giá trị vào B, giảm giá trị trong mảng C để xử lý các phần tử trùng lặp.

Bước 6: Sao chép lại mảng đã sắp xếp:

Sao chép các phần tử trong mảng B trở lại mảng ban đầu (mảng A) để hoàn thành việc sắp xếp.

Ví dụ minh họa:



Hình 24: Ví dụ Counting Sort

Chi tiết cách sắp xếp:

Bước 1: Tìm phần tử lớn nhất trong mảng [1, 4, 7, 2, 4] đã cho. Giá trị lớn nhất đó là 7.

Bước 2: Khởi tạo mảng $C[]$ có độ dài $\max + 1 = 7 + 1 = 8$ với tất cả các phần tử là 0.

Bước 3: Duyệt qua mảng $[1, 4, 7, 2, 4]$, ta sẽ đếm số lần xuất hiện của từng giá trị:

- Số 1 xuất hiện 1 lần: $C[1] = 1$
- Số 4 xuất hiện 2 lần: $C[4] = 2$
- Số 7 xuất hiện 1 lần: $C[7] = 1$
- Số 2 xuất hiện 1 lần: $C[2] = 1$

Bước 4: Cập nhật lại mảng C bằng cách cộng dồn, ta sẽ thực hiện phép cộng dồn để cho biết số lượng các phần tử nhỏ hơn hoặc bằng của mỗi giá trị.

Bước 5: Duyệt dãy số ban đầu theo chiều từ cuối lên đầu dãy và sử dụng mảng C để xác định vị trí của mỗi phần tử trong mảng kết quả.

- Phần tử 4 sẽ đặt vào vị trí $C[4] - 1 = 3$. Sau khi đặt, ta giảm $C[4]$ đi 1: $C[4] = 3$.
- Phần tử 4 kế tiếp đặt vào vị trí $C[4] - 1 = 2$. Sau khi đặt, giảm $C[4]$ đi 1: $C[4] = 2$.
- Phần tử 7 đặt vào vị trí $C[7] - 1 = 4$. Sau khi đặt, ta giảm $C[7]$ đi 1: $C[7] = 4$.
- Phần tử 2 đặt vào vị trí $C[2] - 1 = 1$. Sau khi đặt, ta giảm $C[2]$ đi 1: $C[2] = 1$.
- Phần tử 1 đặt vào vị trí $C[1] - 1 = 0$. Sau khi đặt, ta giảm $C[1]$ đi 1: $C[1] = 0$.

Ta được mảng kết quả (mảng đã sắp xếp là): $B = [1, 2, 4, 4, 7]$.

Sao chép mảng B vào mảng ban đầu (mảng A), thì kết quả mảng A sau khi sắp xếp là:

$$A = [1, 2, 4, 4, 7].$$

Đánh giá độ phức tạp thuật toán:

Phức tạp thời gian:

Trường hợp tốt nhất: $O(n + k)$

Xảy ra khi mảng đã được sắp xếp sẵn theo thứ tự tăng dần.

Trường hợp trung bình: $O(n + k)$

Xảy ra khi các phần tử trong mảng có thứ tự lộn xộn.

Trường hợp xấu nhất: $O(n + k)$

Xảy ra khi mảng đã được sắp xếp sẵn theo thứ tự nghịch đảo.

Trong tất cả các trường hợp trên, độ phức tạp thời gian là như nhau vì dù các phần tử được sắp xếp như thế nào trong mảng, thuật toán vẫn sẽ duyệt qua $n + k$ lần.

Phức tạp không gian: $O(n + k)$

Trong đó:

- n là số phần tử trong mảng đầu vào.
- k là phần tử lớn nhất của mảng đầu vào.

3.10. Radix Sort

Ý tưởng chung của thuật toán:

Radix Sort là một thuật toán sắp xếp tuyển tính sắp xếp các phần tử bằng cách xử lý chúng theo từng chữ số. Đây là một thuật toán sắp xếp hiệu quả cho các số nguyên hoặc chuỗi có khóa có kích thước cố định.

Thay vì so sánh trực tiếp các phần tử, Radix Sort phân phối các phần tử vào các nhóm dựa trên giá trị của từng chữ số. Bằng cách sắp xếp lặp lại các phần tử theo chữ số có nghĩa của chúng, từ ít quan trọng nhất đến quan trọng nhất, Radix Sort đạt được thứ tự được sắp xếp cuối cùng.

Các bước của thuật toán:

RADIX-SORT(A,d)

1. for $i = 1$ to d
2. use a stable sort to sort array A on digit i

Hình 25: Mã giả Radix Sort

Bước 1:

Tìm phần tử lớn nhất trong mảng arr để xác định số chữ số tối đa m .

Đặt $exp = 1$ // biểu diễn chữ số hàng đơn vị.

Bước 2:

Thực hiện phân loại (Counting Sort)

- Tính toán tần suất xuất hiện của các chữ số tại vị trí đang xét. //chia arr[i] cho exp , rồi lấy phần dư khi chia 10.
- Tích lũy tần suất để xác định thứ tự sắp xếp.
- Sắp xếp lại các phần tử vào mảng tạm output[] theo thứ tự của các chữ số đang xét.

Sao chép mảng tạm output[] trở lại arr[]

- Sau mỗi lần phân loại, arr[] được cập nhật theo thứ tự dựa trên chữ số hiện tại.

Tăng exp lên bậc tiếp theo (nhân 10)

- Lặp lại quá trình cho đến khi không còn chữ số nào lớn hơn.

Bước 3:

Sau khi tất cả các chữ số được xử lý, mảng arr sẽ được sắp xếp hoàn toàn.

Ví dụ minh họa:

Step 1:

COMPARE UNITS											
7	2582										
6	4920										
5	1987										
4	2104										
3	3748										
2	251										
1	1246	4920	251	2582		2104		1246	1987	3748	
OS	A	0	1	2	3	4	5	6	7	8	9

Step 2:

COMPARE TENS											
6	1987										
5	1216										
4	2104										
3	2582										
2	251									1987	
1	4920	2104	1216	4920		3748	251			2582	
OS	A	0	1	2	3	4	5	6	7	8	9

Step 3:

COMPARE HUNDREDS											
6	2582										
5	251										
4	3748										
3	4920										
2	1216			251						4920	
1	2104		2104	1216		2582		3748	1987		
OS	A	0	1	2	3	4	5	6	7	8	9

Step 4:

COMPARE THOUSANDS											
6	1987										
5	3748										
4	2582										
3	251										
2	1216		1987	2582							
1	2104	251	1216	2104	3748	4920					
OS	A	0	1	2	3	4	5	6	7	8	9

Step 5:

7	4920										
6	3748										
5	2582										
4	2104										
3	1987										
2	1216										
1	251										
OS	A	0	1	2	3	4	5	6	7	8	9

Hình 26: Ví dụ Radix Sort**Chi tiết các bước sắp xếp:**

Bước 1: Kiểm tra hàng đơn vị của các số 1246, 251, 3748, 2104, 1987, 4920, 2582 để đưa vào các cột tương ứng có các số chạy từ 0 đến 9.

Bước 2: Tương tự bước 1 thực hiện cho hàng chục, trăm, nghìn.

Bước 3: Sau khi thực hiện các bước trên ta được một mảng được sắp xếp theo thứ tự tăng dần là 251, 1216, 1987, 2104, 2582, 3748, 4920.

Đánh giá độ phức tạp thuật toán:**Phức tạp thời gian:**

Trường hợp tốt nhất: $O(d * (n + b))$

Trường hợp trung bình: $O(d * (n + b))$

Trường hợp xấu nhất: $O(d * (n + b))$

Trong đó:

- d là số chữ số
- n là số phần tử
- b là cơ số của hệ thống số đang được sử dụng (trong thuật toán này b = 10)

Phức tạp không gian: $O(n + b)$

Trong đó:

- n là số phần tử

- b là cơ số của hệ thống số

3.11. Flash Sort

Ý tưởng chung của thuật toán:

Flash-Sort sử dụng một vecto $L(k)$ có độ dài m trong bước đầu tiên để phân loại các phần tử của mảng A . Sau đó, trong bước thứ hai, các số đếm kết quả được tích lũy và $L(k)$ trở đến ranh giới lớp. Sau đó, các phần tử được sắp xếp theo hoán vị *tại chỗ*. Trong quá trình hoán vị, $L(k)$ được giảm đi một bước đơn vị tại mỗi vị trí mới của một phần tử của lớp k trong mảng A . Một khía cạnh quan trọng của FlashSort là để xác định các phần tử dẫn đầu chu kỳ mới. Một chu kỳ kết thúc nếu vecto $L(k)$ trở đến vị trí của một phần tử bên dưới ranh giới lớp của lớp k . Phần tử dẫn đầu chu kỳ mới là phần tử nằm ở vị trí thấp nhất tuân thủ điều kiện bổ sung, tức là đối với phần tử này, $L(k)$ trở đến vị trí có $L(k(A(i))) \geq i$. Rõ ràng, ngoài mảng A có độ dài n chứa n phần tử cần sắp xếp, vecto phụ trợ duy nhất là vecto $L(k)$. Kích thước của vecto này bằng số lớp m , nhỏ hơn n , ví dụ m thường có thể được đặt thành $m = 0, 1, n$ trong trường hợp số dấu phẩy động. Cuối cùng, một số lượng nhỏ các phần tử có thể phân biệt được một phần được sắp xếp cục bộ trong các lớp của chúng bằng đệ quy hoặc bằng thuật toán sắp xếp thông thường đơn giản.

Các bước của thuật toán:

Bước 1: Phân lớp dữ liệu

- Tìm giá trị nhỏ nhất của các phần tử trong mảng(\minVal) và vị trí phần tử lớn nhất của các phần tử trong mảng(\max).
- Khởi tạo 1 mảng L có m phần tử (ứng với m lớp, trong source code lần này chọn số lớp bằng $0, 45n$).
- Đếm số lượng phần tử các lớp theo quy luật, phần tử $a[i]$ sẽ thuộc lớp: $k = \text{int}((m - 1) * (a[i] - \minVal) / (\max - \minVal))$.
- Tính vị trí kết thúc của phân lớp thứ j theo công thức:

$$L[j] = L[j] + L[j - 1] \quad (j tăng từ 1 đến m - 1).$$

Bước 2: Hoán vị toàn cục

- Đầu tiên: $\text{swap}(a[\max], a[0])$.
- Với tối đa $n - 1$ lần swap, n phần tử trong mảng sẽ về đúng phân lớp của mình.
- Khi $j > L[k] - 1$ nghĩa là khi đó phần tử $a[j]$ đã nằm đúng vị trí phân lớp của nó do đó ta bỏ qua và tiếp tục tăng j lên để xét các phần tử tiếp theo.
- Mỗi lần đưa 1 phần tử về đúng phân lớp của nó ta giảm vị trí cuối cùng của phân lớp đó xuống (đồng thời tăng biến đếm số lần swap lên 1 đơn

vị), quá trình này được thực hiện cho tới khi $L[k] = j$, điều này cũng tương đương với việc phân lớp k đã đầy.

Bước 3: Sử dụng hàm Insertion Sort để sắp xếp lại từng phân lớp và mảng hoàn chỉnh.

FLASHSORT(A,n)

```

1. dimension A(1),l(1)
2. ===== class formation =====
3. anmin = A(1)
4. nmax = 1
5. do i = 1,n
6.   if( A(i).lt.anmin) anmin=A(i)
7.   if( A(i).gt.A(nmax)) nmax = i
8. end do
9. if (anmin.eq.A(nmax)) return
10. c1 = (m - 1) / (A(nmax) - anmin)
11. do k=1,m
12.   l(k)=0
13. end do
14. do i = 1, n
15.   k = 1 + int(c1 * (A(i) - anmin))
16.   l(k) = l(k) + 1
17. end do
18. do k = 2,m
19.   l(k) = l(k) + l(k - 1)
20. end do
21. hold = a(nmax)
22. A(nmax) = A(1)
23. A(1) = hold
24. ===== permutation =====
25. nmove=0
26. j = 1
27. k = m
28. do while (nmove.lt.n - 1)
29.   do while (j.gt.l(k))
30.     j = j + 1
31.     k = 1 + int(c1 * (A(j) - anmin))
32.   end do
33.   flash=A(j)
34.   do while (.not.(j.eq.l(k) + 1))
35.     k = 1 + int(c1 * (flash - anmin))
36.     hold=A(l(k))
37.     A(l(k)) = flash

```

```

38.     flash = hold
39.     l(k) = l(k) - 1
40.     nmove = nmove + 1
41.   end do
42. end do
43. ===== straight insertion =====
44. do i = n - 2, 1, - 1
45.   if (A(i + 1).lt.A(i)) then
46.     hold = a(i)
47.     j = i
48.     do while (A(j + 1).lt.hold)
49.       A(j) = A(j + 1)
50.       j = j + 1
51.     end do
52.     A(j) = hold
53.   endif
54. end do
55. ===== return,end flash1 =====
56. return
57. end

```

Hình 27: Mã giả Flash Sort (Hình 1)

Từ mã giả trên cùng với các thông tin và nguồn code mà nhóm chúng em đã tham khảo được. Chúng em đã đưa ra cách triển khai cuối cùng cho thuật toán Flash Sort như mã giả bên dưới minh họa.

FLASH SORT(A, n)

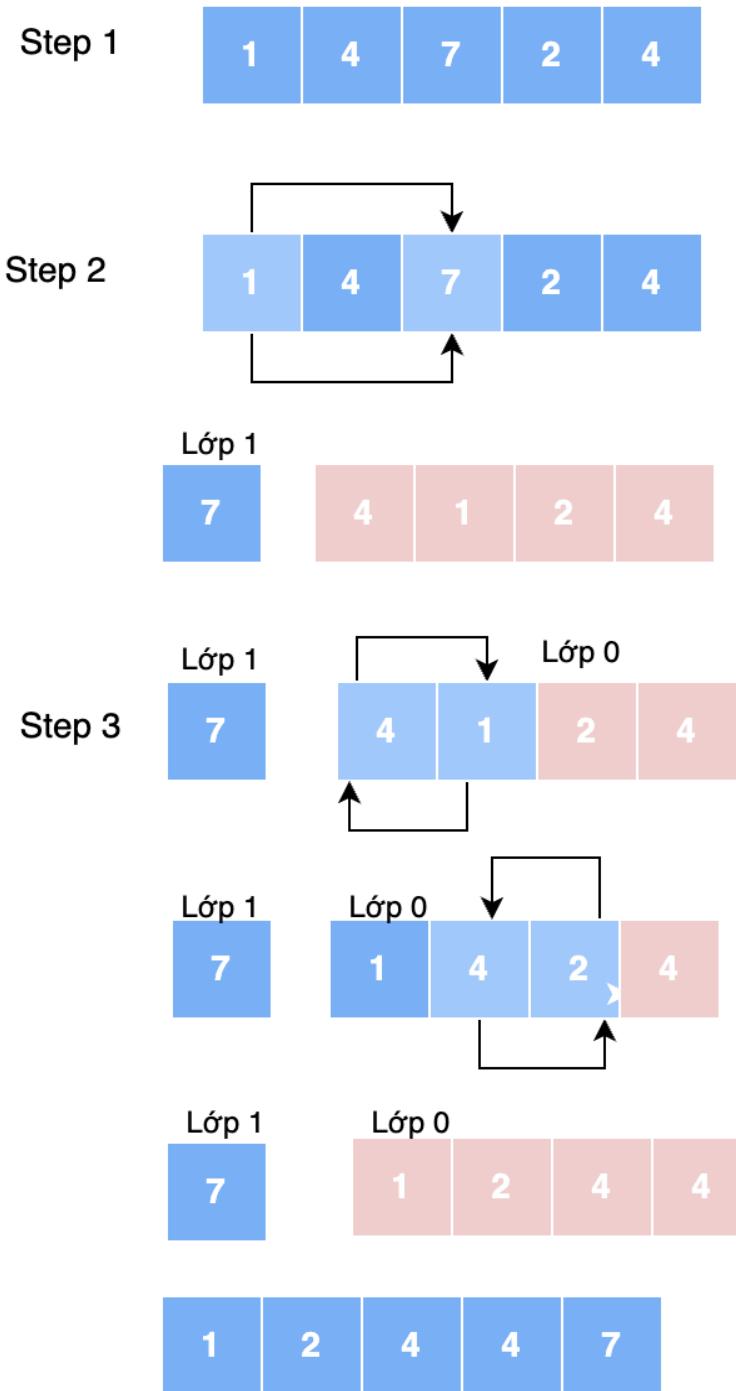
```

1. min = FindMin(A)
2. max = FindMax(A)
3. m = 0.45 * n
4. Create L[0..m-1] as empty lists
5. for i = 0 to n - 1
6.   k = ((m - 1) * (A[i] - min)) / (max - min)
7.   L[k] = L[k] + 1
8. L[0] = L[0] - 1
9. for j = 1 to m - 1
10.  L[j] = L[j] + L[j - 1]
11. Swap(A[0], A[max_index])
12. for i = 1 to n - 1
13.   k = ((m - 1) * (A[i] - min)) / (max - min)
14.   Swap(A[i], A[L[k]])
15.   L[k] = L[k] - 1
16. for each group k:
17.   InsertionSort(L[k])

```

Hình 28: Mô hình Flash Sort (Hình 2)

Ví dụ minh họa:

**Hình 29: Ví dụ Flash Sort**

Chi tiết cách sắp xếp:

Bước 1: Phân lớp dữ liệu

- Tìm minVal và maxVal:
 - $\minVal = 1, \maxVal = 7.$
- Chia thành các lớp:
 - Số lớp $m = 0.45 * n = 0.45 * 5 = 2.25 = 2$ (làm tròn xuống).
 - Tính số lớp của từng phần tử bằng công thức:
 - $k = \text{int}((m - 1) * (a[i] - \minVal) / (\maxVal - \minVal))$ (làm tròn xuống)
 - $a[0] = 1: \text{thuộc lớp } 0$
 - $a[0] = 1: \text{thuộc lớp } 0$
 - Tính vị trí kết thúc của mỗi lớp:
 - Lớp 0 có 4 phần tử, nên $L[0] = 4$ (vị trí cuối của lớp 0 là 3).
 - Lớp 1 có 1 phần tử, $L[1] = L[0] + 1 = 4 + 1 = 5$ (vị trí cuối của lớp 1 là 4).

Bước 2: Hoán vị toàn cục

- Đưa maxVal về đầu:
 - Hoán đổi vị trí $a[0]$ với $a[\maxVal]$.
- Đưa các phần tử về đúng lớp của nó.
 - Lớp 0: [4, 1, 2, 4].
 - Lớp 1: [7].

Bước 3: Sử dụng Insertion Sort.

- Lớp 0 : [4, 1, 2, 4].
 - Chèn 1 lên trước 4: [1, 4, 2, 4]
 - Chèn 2 lên trước 4: [1, 2, 3, 4]
 - Kết quả: [1, 2, 4, 4].
- Lớp 1: [7]: Không cần sắp xếp.
- Ghép lớp 0 và lớp 1 lại: [1, 2, 4, 4, 7].

Đánh giá độ phức tạp thuật toán:**Phức tạp thời gian:**

Tốt nhất: $O(n)$

Trung bình: $O(n + m)$

Trong đó:

- n là số phần tử.
- m là kích thước của mảng phụ trợ (thường bằng $0,45n$).

Xấu nhất: $O(n^2)$

Phức tạp không gian: $O(m)$

Trong đó:

- m là kích thước của mảng phụ trợ (thường bằng $0,45n$).

4. KẾT QUẢ THỰC NGHIỆM VÀ NHẬN XÉT

4.1. Bộ dữ liệu có thứ tự ngẫu nhiên

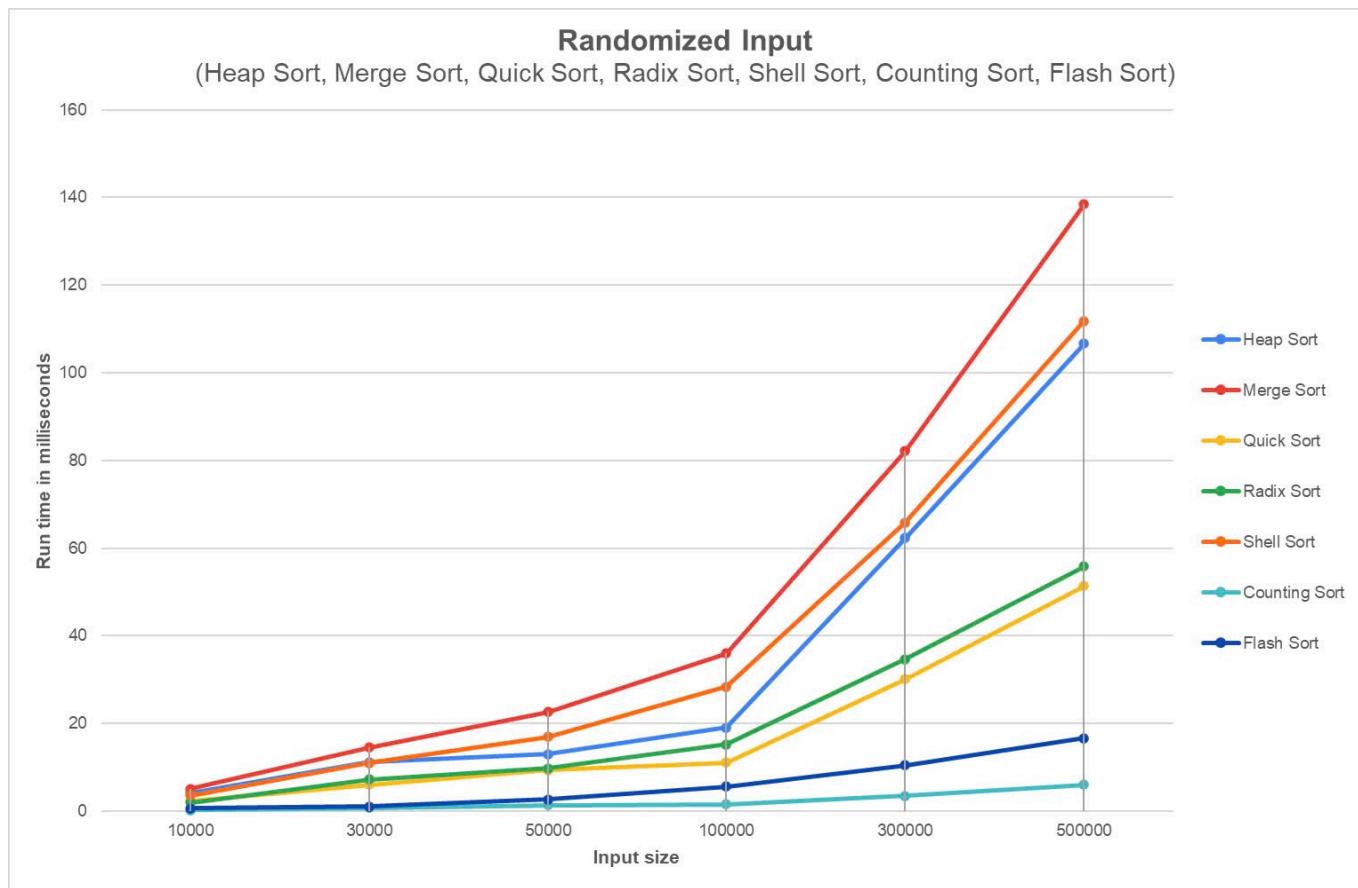
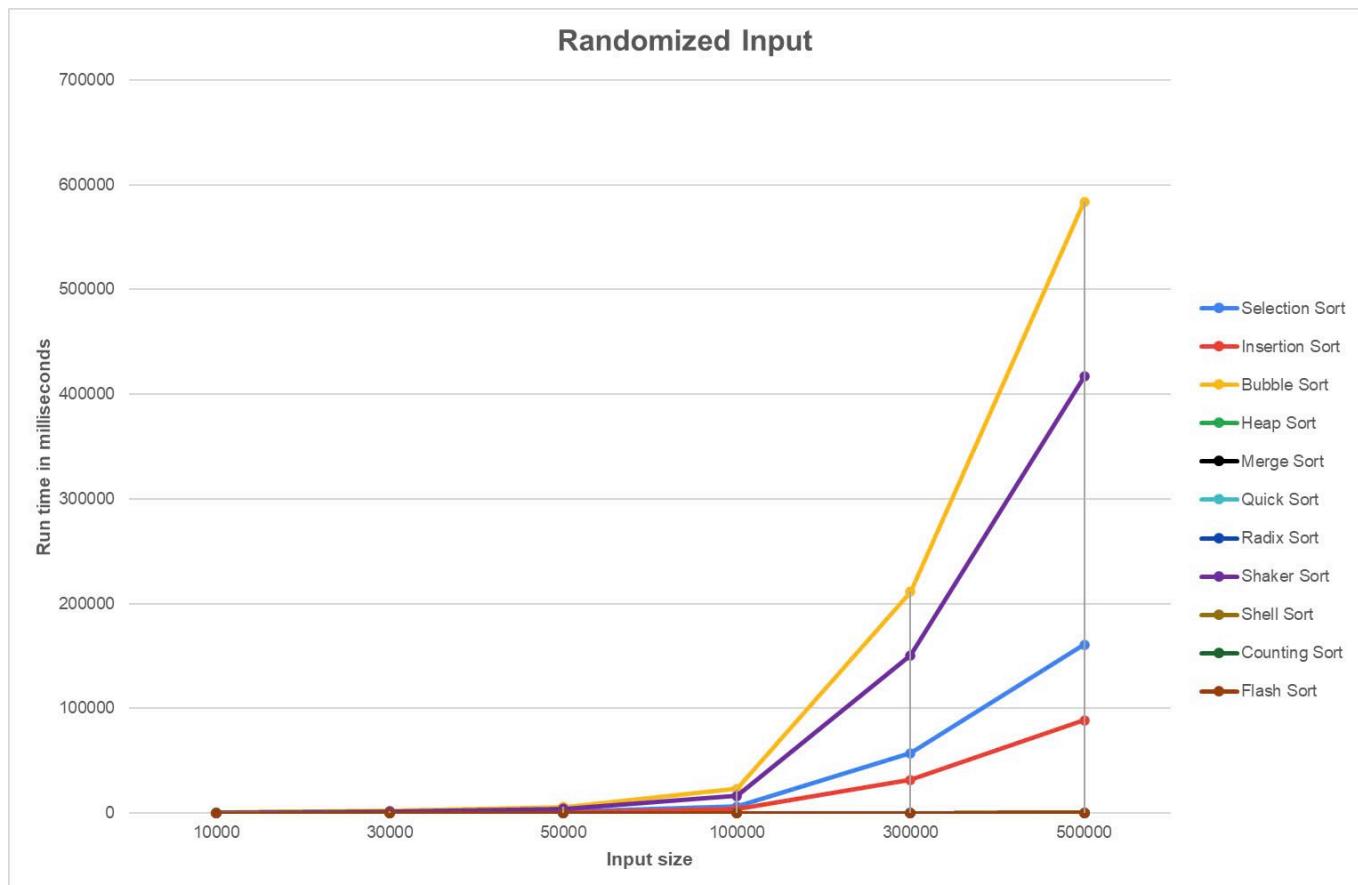
Bảng dữ liệu:

Data order: RANDOMIZED DATA						
Data size	10000		30000		50000	
Resulting statics	Running time	Comparisons	Running time	Comparisons	Running time	Comparisons
<i>Selection Sort</i>	94,2546	100009999	594,107	900029999	1601,6	2500049999
<i>Insertion Sort</i>	58,8756	49949602	332,142	447680980	882,017	1252782840
<i>Bubble Sort</i>	227,904	99991101	2004,56	900054817	5638,73	2500080401
<i>Heap Sort</i>	4,21829	637663	11,21	2150311	13,0416	3771761
<i>Merge Sort</i>	5,11233	846914	14,5817	2759256	22,6846	4896580
<i>Quick Sort</i>	2,31196	14003	5,99817	44451	9,3335	64967
<i>Radix Sort</i>	1,99404	140056	7,29517	510070	9,83587	850070
<i>Shaker Sort</i>	183,138	66845353	1454,5	597375567	4087,52	1671837371
<i>Shell Sort</i>	3,64496	252630	11,0482	914011	16,9364	1851085
<i>Counting Sort</i>	0,254125	50004	0,8035	150004	1,40033	250004
<i>Flash Sort</i>	0,685042	68609	1,12271	207811	2,65325	369509

Data order: RANDOMIZED DATA						
Data size	100000		300000		500000	
Resulting statics	Running time	Comparisons	Running time	Comparisons	Running time	Comparisons
<i>Selection Sort</i>	6407,44	10000099999	57240,6	90000299999	160680	250000499999
<i>Insertion Sort</i>	3499,71	4984456275	31757,3	45047657562	88606,6	125042814858
<i>Bubble Sort</i>	23119,5	10000136497	211561	89999467905	583748	249997532957
<i>Heap Sort</i>	19,0753	8043948	62,2792	26489370	106,644	45970506
<i>Merge Sort</i>	35,9916	10391856	82,1203	34511193	138,41	58064932
<i>Quick Sort</i>	11,0582	149529	30,1105	417841	51,3767	628523
<i>Radix Sort</i>	15,2837	1700070	34,6558	6000084	55,831	10000084
<i>Shaker Sort</i>	16618,5	6639513238	150641	60065271560	417036	166767156595
<i>Shell Sort</i>	28,429	4281806	65,8115	14879922	111,799	27670877
<i>Counting Sort</i>	1,55017	500003	3,53792	1500003	6,01154	2500004
<i>Flash Sort</i>	5,64363	736938	10,508	2011391	16,6483	3516827

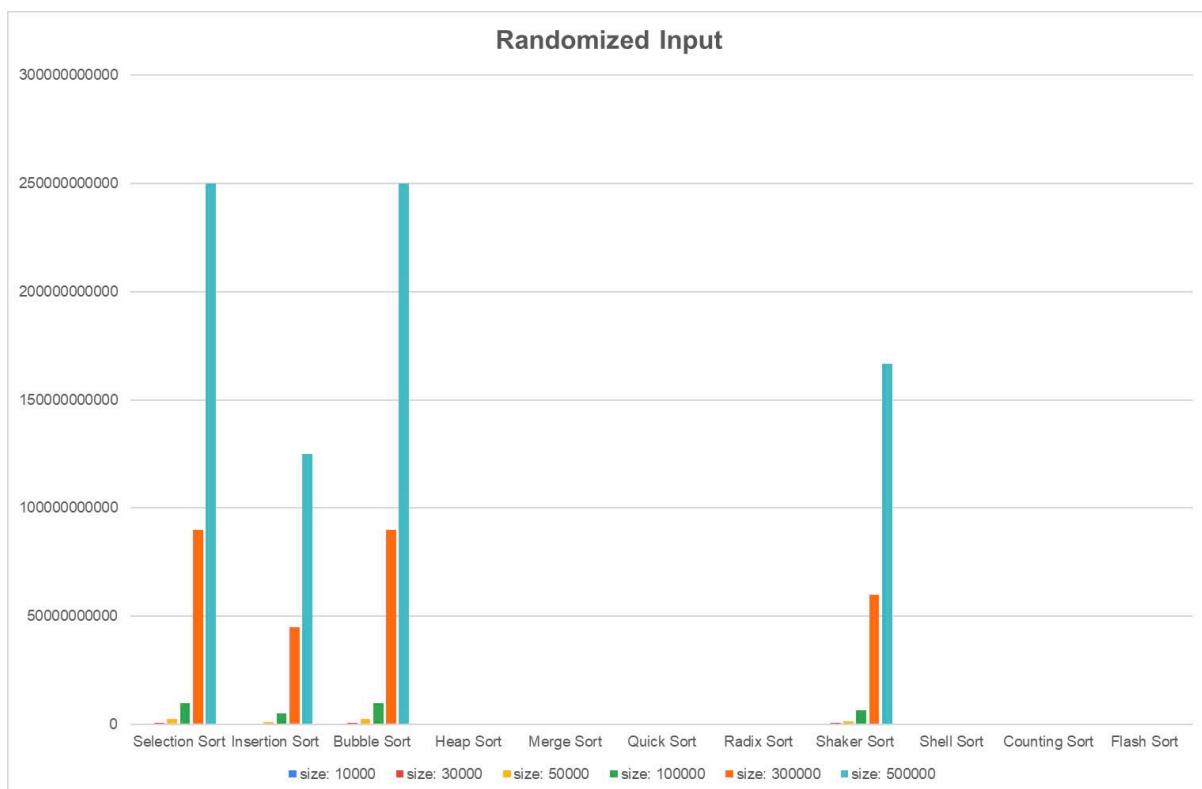
Hình 30: Bảng dữ liệu ngẫu nhiên

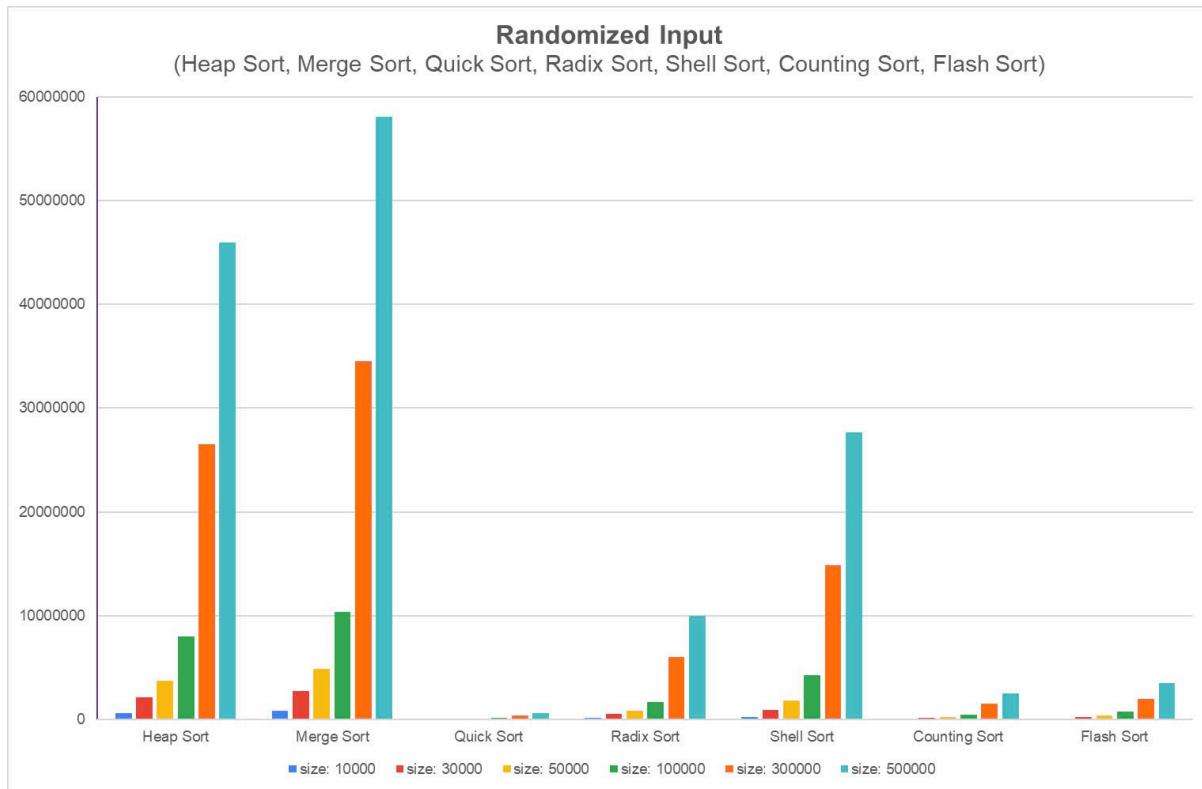
Biểu đồ đường:



Hình 31: Biểu đồ đường dữ liệu ngẫu nhiên**Nhận xét:**

- Bubble Sort và Shaker Sort: Cả hai thuật toán này đều có hiệu suất kém, cho thấy sự kém hiệu quả rõ rệt khi làm việc với các tập dữ liệu lớn. Thời gian thực thi của chúng tăng mạnh khi kích thước đầu vào tăng.
- Selection Sort và Insertion Sort: Các thuật toán này hoạt động tốt hơn Bubble Sort và Shaker Sort, nhưng vẫn có sự tăng trưởng nhanh trong thời gian thực thi. Thời gian chạy của chúng tăng rõ rệt với các tập dữ liệu lớn, mặc dù các thuật toán này có lợi thế là ít hoán đổi hơn.
- Counting Sort: Nhìn chung, là thuật toán có hiệu suất tốt nhất, với thời gian thực thi luôn thấp trên tất cả các kích thước đầu vào. Thuật toán này đặc biệt hiệu quả nhờ vào tính chất không so sánh trực tiếp giữa các phần tử.
- Flash Sort: Hiệu suất gần bằng Counting Sort.
- Radix Sort, Quick Sort, Shell Sort, Merge Sort và Heap Sort: Đều là những thuật toán có hiệu suất tốt, có thời gian thực thi tương đối thấp ngay cả khi kích thước đầu vào tăng.

Biểu đồ cột:



Hình 32: Biểu đồ cột dữ liệu ngẫu nhiên

Nhận xét:

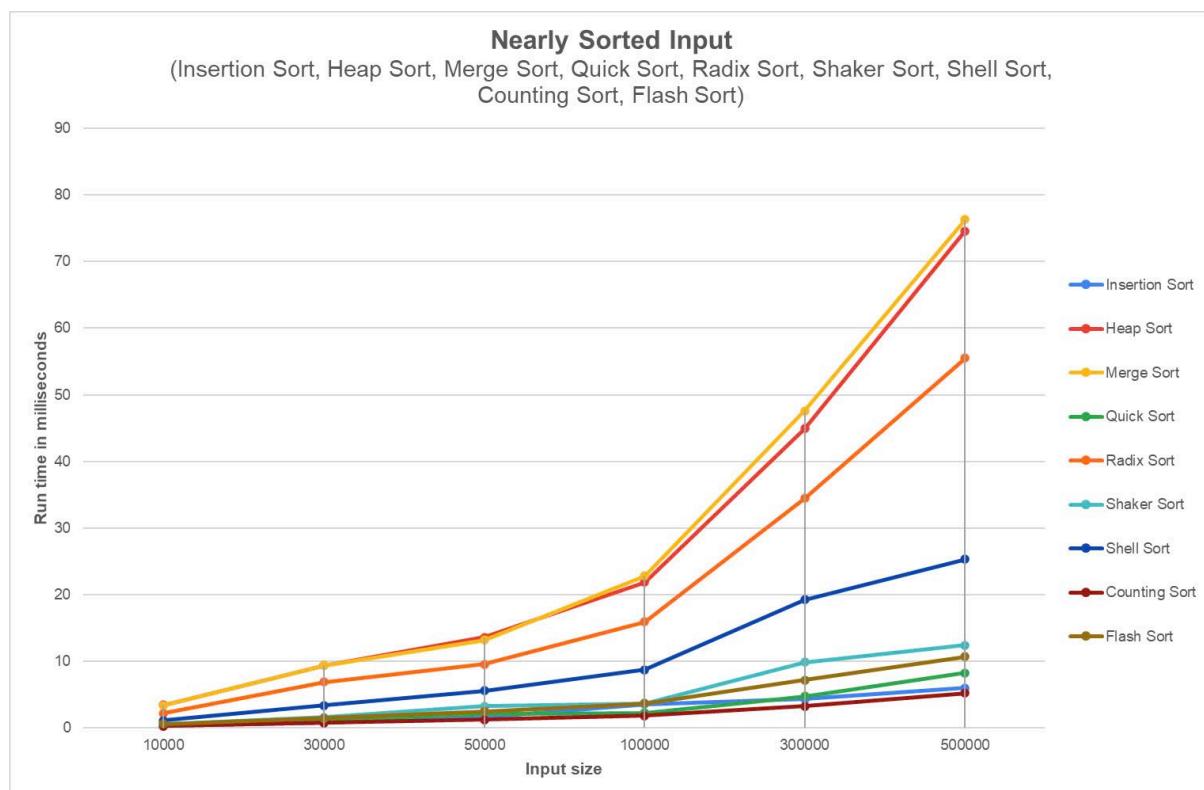
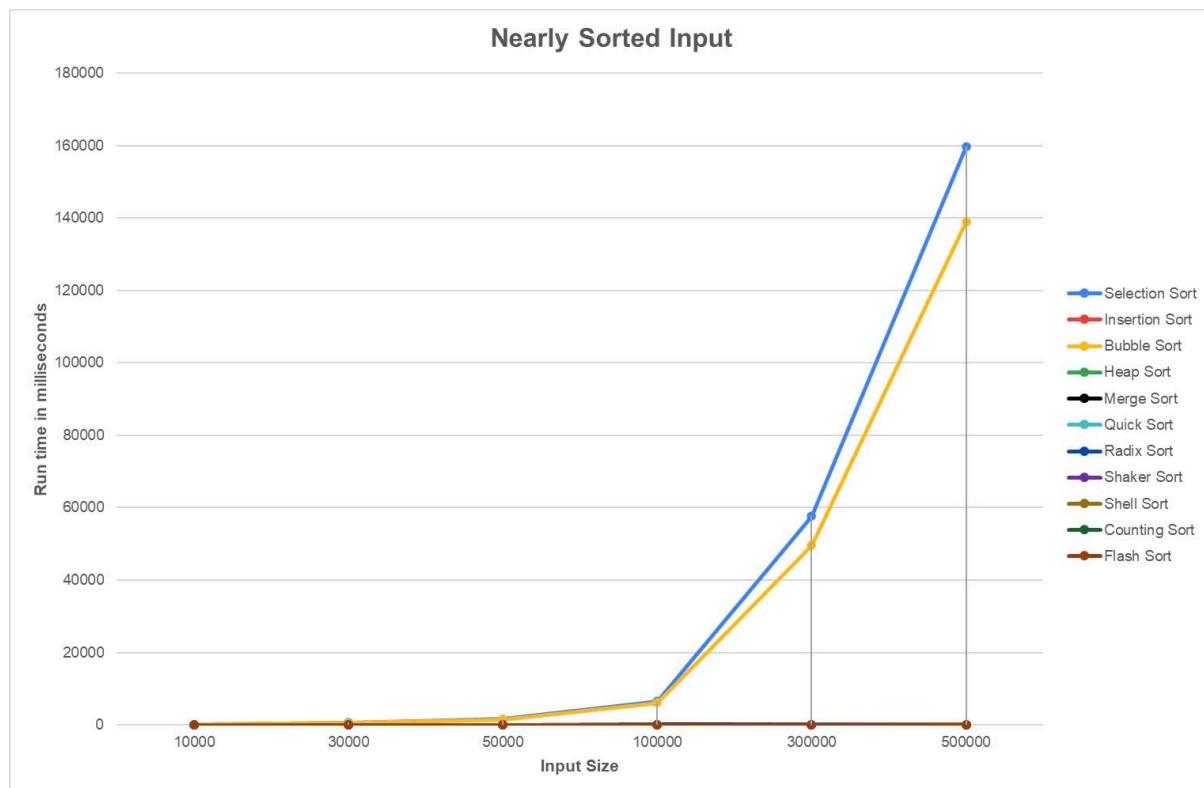
- Selection Sort, Bubble Sort, Shaker Sort: Các thuật toán này có số lượng phép so sánh đã thực hiện cao nhất trong tất cả các kích thước mảng. Vì số lượng phép so sánh tăng mạnh khi kích thước mảng lớn hơn, do đó chúng không phù hợp với các tập dữ liệu lớn. Các thuật toán này đều có độ phức tạp $O(n^2)$, điều này thể hiện rõ trong việc hiệu suất kém khi kích thước mảng tăng.
- Insertion Sort: Tương tự như ba thuật toán trên nhưng với số lượng so sánh ít hơn một chút. Do độ phức tạp $O(n^2)$ nên sự tăng trưởng trong số lượng so sánh vẫn tương đối đáng kể.
- Counting Sort và Radix Sort: Những thuật toán sắp xếp này sử dụng rất ít phép so sánh vì không sắp xếp bằng cách so sánh trực tiếp hai phần tử mà chủ yếu dựa vào việc đếm số lần xuất hiện của các phần tử (Counting Sort) và đặt vị trí các phần tử dựa trên giá trị chữ số tương ứng trong vòng lặp của chúng (Radix Sort).
- Quick Sort: Thuật toán có số phép so sánh ít nhất vì nó sử dụng phương pháp chia để trị, phân tách mảng thành các phần nhỏ. Mỗi lần phân tách, chỉ cần so sánh các phần tử với pivot, sau đó áp dụng đệ quy cho các phân đoạn nhỏ hơn.

4.2. Bộ dữ liệu gần được sắp xếp hoàn chỉnh

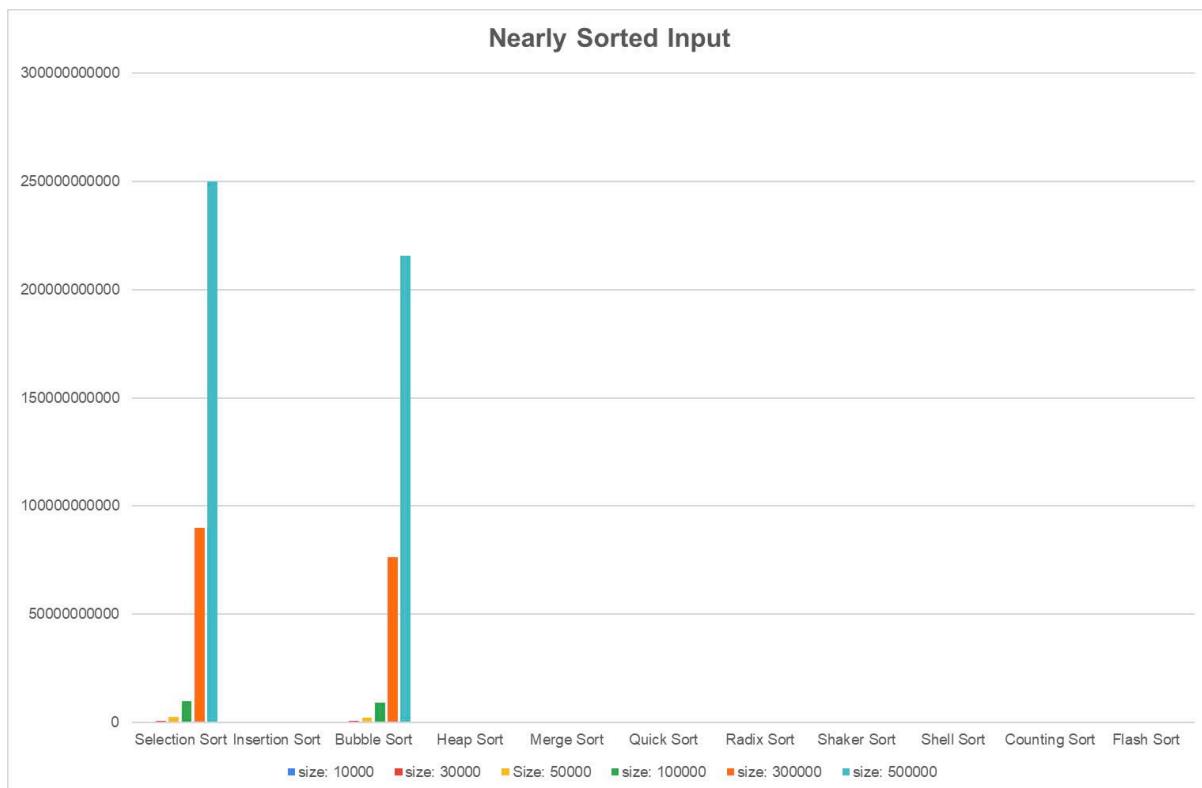
Bảng dữ liệu:

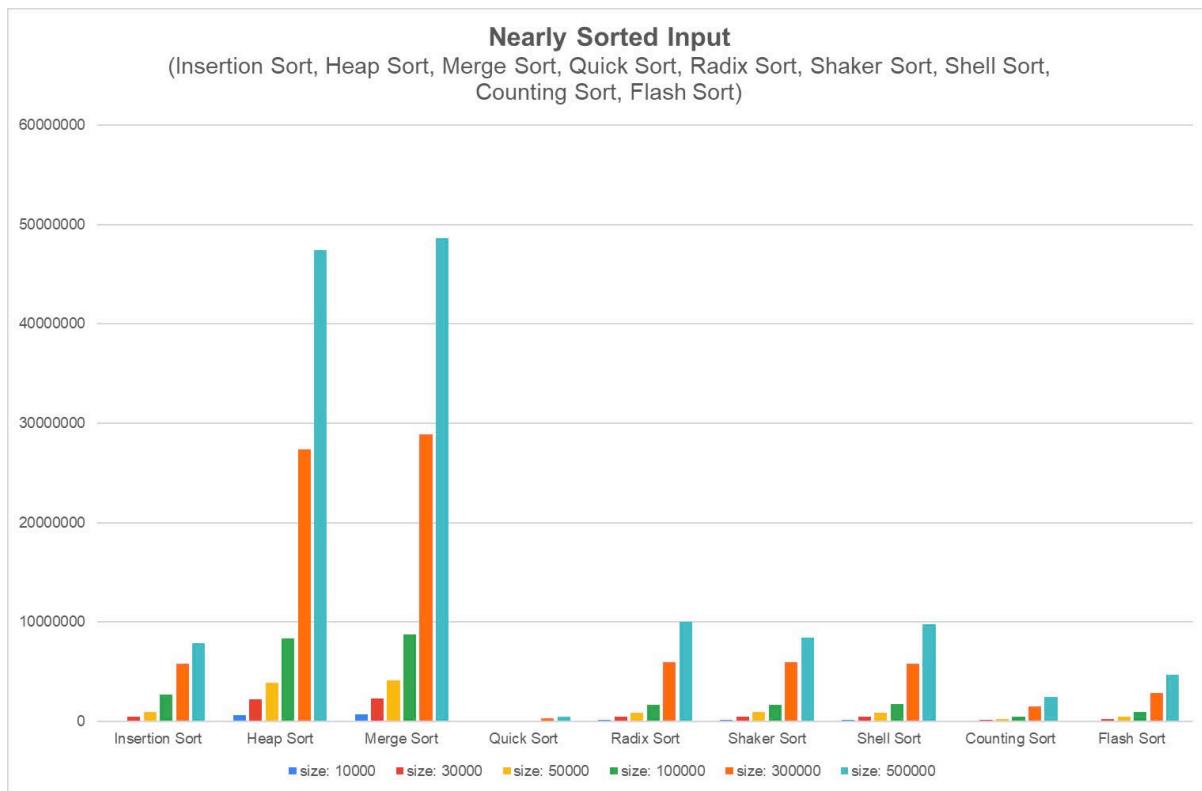
Data order: NEARLY SORTED DATA						
Data size	10000		30000		50000	
Resulting statics	Running time	Comparisons	Running time	Comparisons	Running time	Comparisons
<i>Selection Sort</i>	90,2682	100009999	598,997	900029999	1609,39	2500049999
<i>Insertion Sort</i>	0,281625	124378	0,814208	445182	1,56212	936374
<i>Bubble Sort</i>	84,2137	82863837	483,578	722264445	1520,39	2361708305
<i>Heap Sort</i>	3,40379	669851	9,37633	2236656	13,6161	3925141
<i>Merge Sort</i>	3,387	736406	9,36554	2340477	13,162	4170904
<i>Quick Sort</i>	0,379542	10010	1,15175	30018	2,12246	50018
<i>Radix Sort</i>	2,19417	140056	6,86212	510070	9,55271	850070
<i>Shaker Sort</i>	0,557833	142326	1,56025	464277	3,27904	970206
<i>Shell Sort</i>	1,12546	136794	3,356	455718	5,55663	853690
<i>Counting Sort</i>	0,302833	50004	0,744167	150004	1,25475	250004
<i>Flash Sort</i>	0,598917	94469	1,49804	283468	2,48346	472468

Data order: NEARLY SORTED DATA						
Data size	100000		300000		500000	
Resulting statics	Running time	Comparisons	Running time	Comparisons	Running time	Comparisons
<i>Selection Sort</i>	6467,71	10000099999	57643,9	90000299999	159683	250000499999
<i>Insertion Sort</i>	1,28733	1669902	4,32625	5814238	5,98071	7910742
<i>Bubble Sort</i>	6050,08	9309982017	49512,8	76476072976	138828	215441446397
<i>Heap Sort</i>	13,5137	8364777	44,8941	27417701	74,5846	47428412
<i>Merge Sort</i>	22,8106	8791575	47,6079	28918146	76,311	48637012
<i>Quick Sort</i>	2,25475	100014	4,75687	300018	8,2285	500014
<i>Radix Sort</i>	15,9165	1700070	34,4798	6000084	55,4977	10000084
<i>Shaker Sort</i>	3,66583	1706493	9,86817	5978635	12,4357	8465413
<i>Shell Sort</i>	8,74971	1763315	19,2376	5833191	25,3129	9826880
<i>Counting Sort</i>	1,80937	500004	3,25108	1500004	5,21417	2500004
<i>Flash Sort</i>	3,69333	944964	7,19113	2834966	10,6708	4724966

Hình 33: Bảng dữ liệu gần sắp xếp**Hình 34: Biểu đồ đường dữ liệu gần sắp xếp****Nhận xét:**

- Đối với dữ liệu gần như đã sắp xếp, thuật toán Selection Sort và Bubble Sort (với độ phức tạp $O(n^2)$) là chậm nhất.
- Ngoại trừ Selection Sort và Bubble Sort, với kiểu dữ liệu đầu vào này, các thuật toán khác hoạt động rất hiệu quả ngay cả khi xử lý dữ liệu có kích thước lớn.
- Do có nhiều đường biểu đồ chồng lên nhau, không thể xác định được thuật toán nào chạy tối ưu (nhanh nhất). Vì vậy, chúng ta chỉ có thể dựa vào bảng dữ liệu để kết luận rằng thuật toán chạy nhanh nhất là Counting Sort. Điều này bởi vì với kiểu sắp xếp này, hầu hết các phần tử đã ở vị trí đúng, nên không cần tìm vị trí chèn cho nhiều phần tử.
- Thuật toán nhanh thứ hai là Insertion Sort với độ phức tạp thời gian ở trường hợp tốt nhất là $O(n)$ vì bộ dữ liệu gần như đã được sắp xếp, sau đó là Quick Sort với độ phức tạp thời gian trong trường hợp tốt nhất là $O(n \log n)$.





Hình 35: Biểu đồ cột dữ liệu gần sắp xếp

Nhận xét:

- Selection Sort được triển khai với hai vòng lặp lồng nhau và không được tối ưu hóa cho kiểu dữ liệu đã được sắp xếp. Do đó, khi làm việc với dữ liệu đầu vào gần như đã được sắp xếp, số lượng so sánh là rất lớn.
- Thuật toán Bubble Sort cũng có số lượng so sánh khá lớn. Do các cải tiến trong mã nguồn (cờ đánh dấu swapped), số lượng so sánh đã giảm so với Selection sort.
- Ngoại trừ hai thuật toán trên, các thuật toán sắp xếp khác có thể xử lý hiệu quả các kích thước đầu vào lớn mà không cần so sánh quá nhiều.
- Quick Sort là thuật toán có số lượng phép so sánh ít nhất vì nó thực hiện việc chia để trị, phân tách mảng thành các phần nhỏ. Mỗi lần phân tách, chỉ cần so sánh các phần tử với pivot, sau đó áp dụng đệ quy cho các phân đoạn nhỏ hơn.
- Counting Sort là thuật toán có số lượng so sánh ít thứ hai sau Quick Sort. Vì đây là phương pháp sắp xếp không so sánh mà đếm số lần xuất hiện thay vì so sánh các phần tử, số lượng so sánh có thể giảm đáng kể.

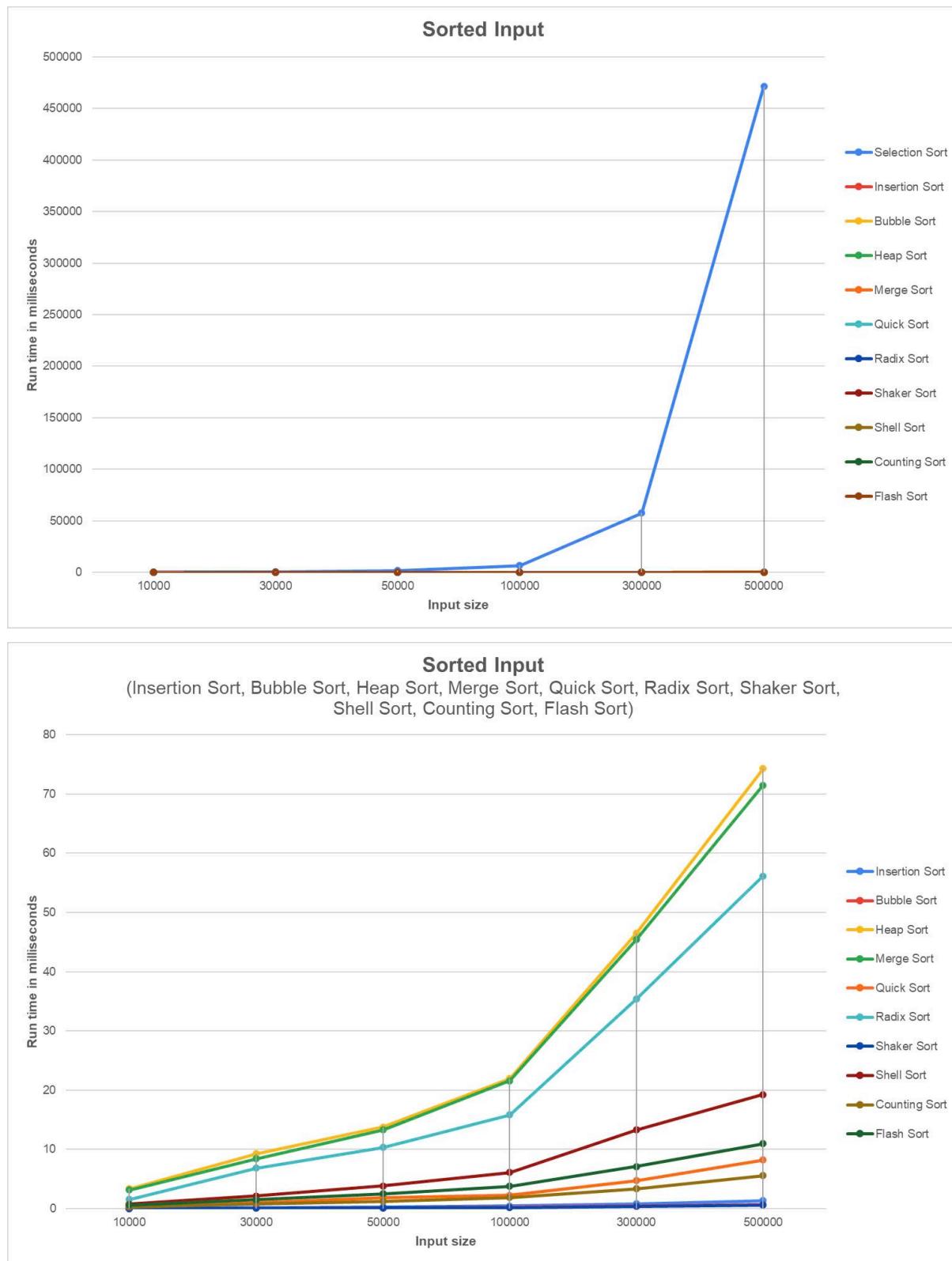
4.3. Bộ dữ liệu đã sắp xếp

Bảng dữ liệu:

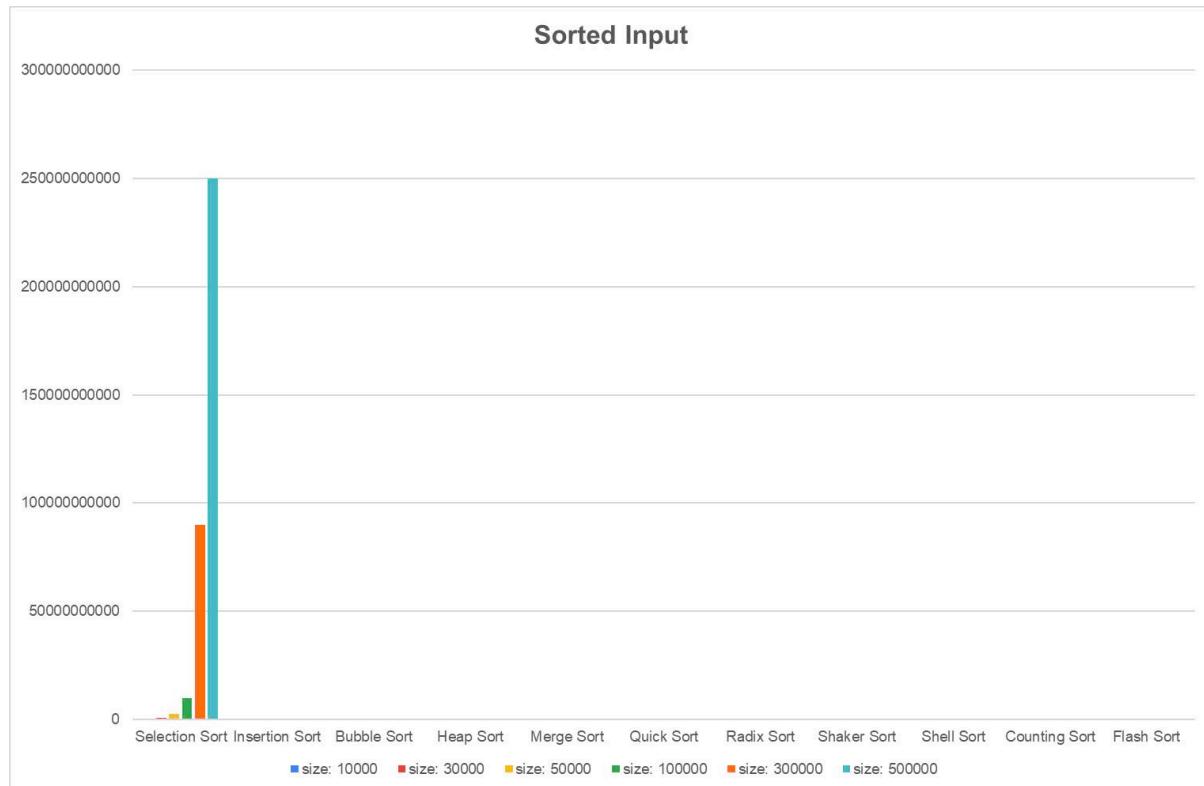
Data order: SORTED DATA						
Data size	10000		30000		50000	
Resulting statics	Running time	Comparisons	Running time	Comparisons	Running time	Comparisons
Selection Sort	94,2785	100009999	593,187	900029999	1588,99	2500049999
Insertion Sort	0,075958	29998	0,187125	89998	0,311834	149998
Bubble Sort	0,031292	20001	0,097875	60001	0,156041	100001
Heap Sort	3,35717	670329	9,29958	2236648	13,7753	3925351
Merge Sort	3,13725	691508	8,44592	2214561	13,3202	3938509
Quick Sort	0,310416	10006	0,988875	30006	1,86371	50006
Radix Sort	1,55771	140056	6,85633	510070	10,3205	850070
Shaker Sort	0,038542	20002	0,093792	60002	0,156291	100002
Shell Sort	0,796333	120005	2,12737	390007	3,84629	700006
Counting Sort	0,309792	50004	0,823292	150004	1,26921	250004
Flash Sort	0,616583	94491	1,50537	283491	2,51896	472491

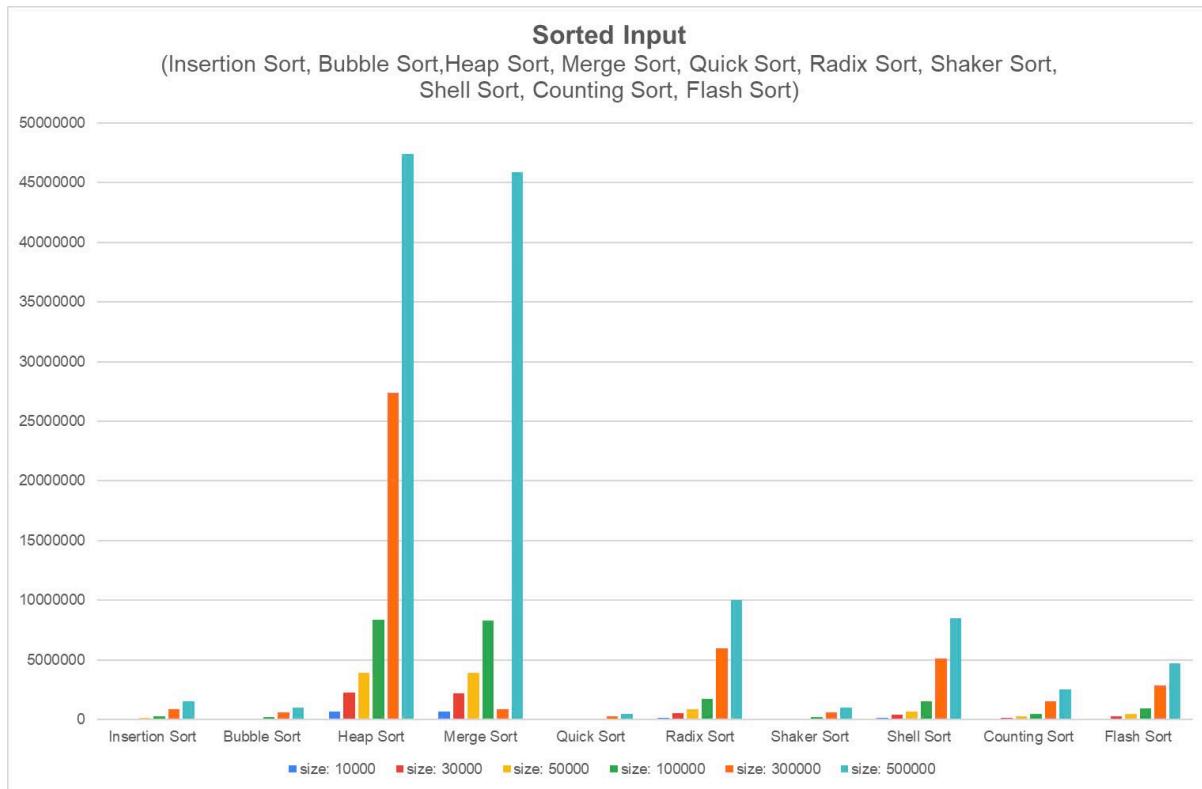
Data order: SORTED DATA						
Data size	100000		300000		500000	
Resulting statics	Running time	Comparisons	Running time	Comparisons	Running time	Comparisons
Selection Sort	6462,21	10000099999	57630,3	90000299999	471578	250000499999
Insertion Sort	0,470625	299998	0,839542	899998	1,34242	1499998
Bubble Sort	0,23925	200001	0,466	600001	0,649167	1000001
Heap Sort	21,986	8365080	46,4627	27413230	74,2782	47404886
Merge Sort	21,6033	8326959	45,4157	899998	71,4727	45883672
Quick Sort	2,25458	100006	4,73321	300006	8,21967	500006
Radix Sort	15,8335	1700070	35,4083	6000084	56,1518	10000084
Shaker Sort	0,204875	200002	0,383417	600002	0,63625	1000002
Shell Sort	6,13025	1500006	13,2863	5100008	19,2627	8500007
Counting Sort	1,87129	500004	3,36175	1500004	5,55917	2500004
Flash Sort	3,76671	944991	7,12767	2834991	10,9967	4724991

Hình 36: Bảng dữ liệu đã sắp xếp

**Hình 37: Biểu đồ đường dữ liệu đã sắp xếp****Nhận xét:**

- Nhìn chung ở bảng tổng quát của tất cả các thuật toán, hầu hết các đường đồ thị đều nằm ngang, điều đó cho thấy các thuật toán được tối ưu rất tốt cho mảng đã được sắp xếp, ngay cả khi thuật toán có độ phức tạp thời gian là $O(n^2)$.
- Đôi với dữ liệu đã sắp xếp, Selection Sort là thuật toán chậm nhất. Độ phức tạp của Selection Sort là $O(n^2)$ trong mọi trường hợp, độ phức tạp này là do Selection Sort được triển khai với hai vòng lặp lồng nhau và không được tối ưu hóa cho kiểu dữ liệu đã được sắp xếp.
- Mặc dù Bubble Sort cũng chạy với hai vòng lặp lồng nhau, nhưng thuật toán này đã được tối ưu hóa với cờ đánh dấu swapped cho mảng đầu vào đã sắp xếp, vì vậy độ phức tạp thời gian của nó được giảm xuống $O(n)$.
- Ngoại trừ Selection Sort, các thuật toán sắp xếp khác hoạt động tương đối nhanh trên mảng đã được sắp xếp. Khi quan sát thời gian chạy của các thuật toán còn lại ở hình ..., Shaker Sort, Insertion Sort và Counting Sort là ba thuật toán cho thời gian chạy tốt nhất với bộ dữ liệu này.



**Hình 38: Biểu đồ cột dữ liệu đã sắp xếp****Nhận xét:**

Nhìn tổng thể tất cả các cột ở hình ..., có thể thấy ngoại trừ Selection Sort, số phép so sánh của các thuật toán sắp xếp khác là không đáng kể.

Selection Sort vẫn là thuật toán tệ nhất đối với mảng đã được sắp xếp vì thuật toán thực hiện một số lượng lớn phép so sánh. Thuật toán này có số phép so sánh lớn như vậy vì được triển khai với hai vòng lặp lồng nhau và không được tối ưu hóa cho dữ liệu đầu vào đã sắp xếp. Do đó, bất kể dạng mảng đầu vào như thế nào, thuật toán vẫn sẽ duyệt qua toàn bộ các vòng lặp.

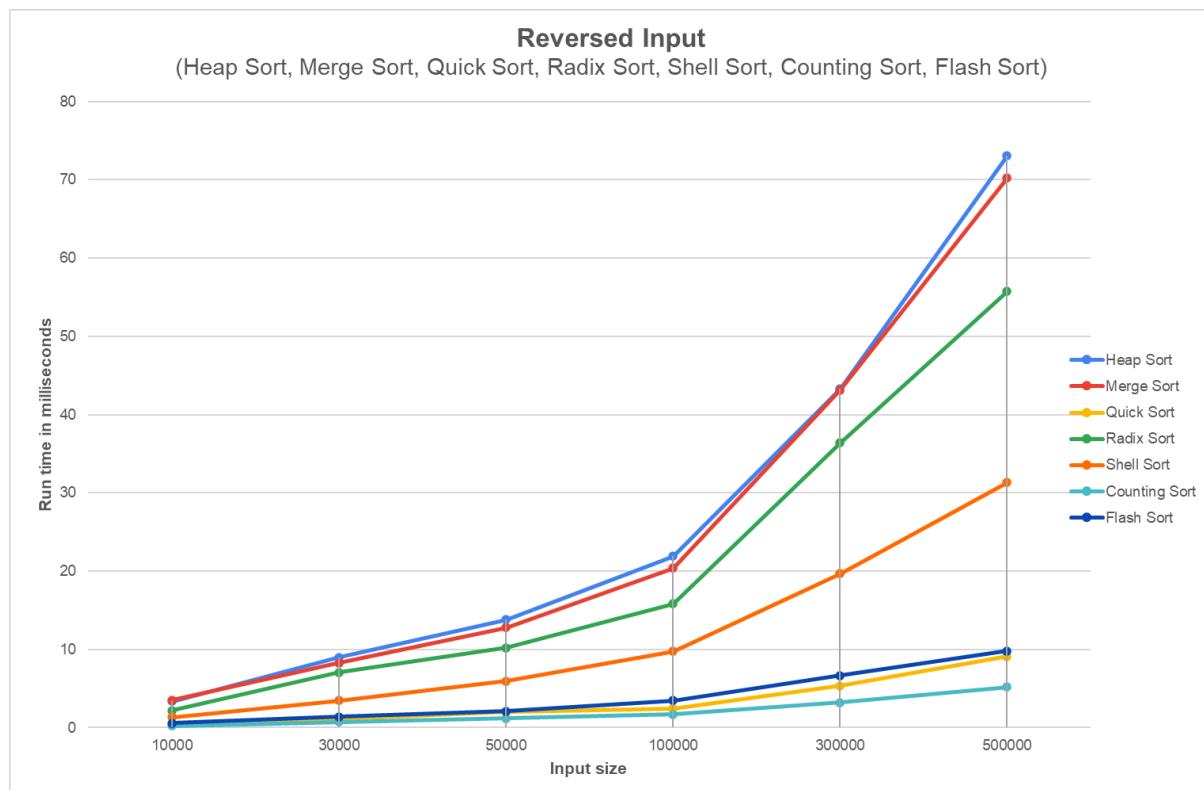
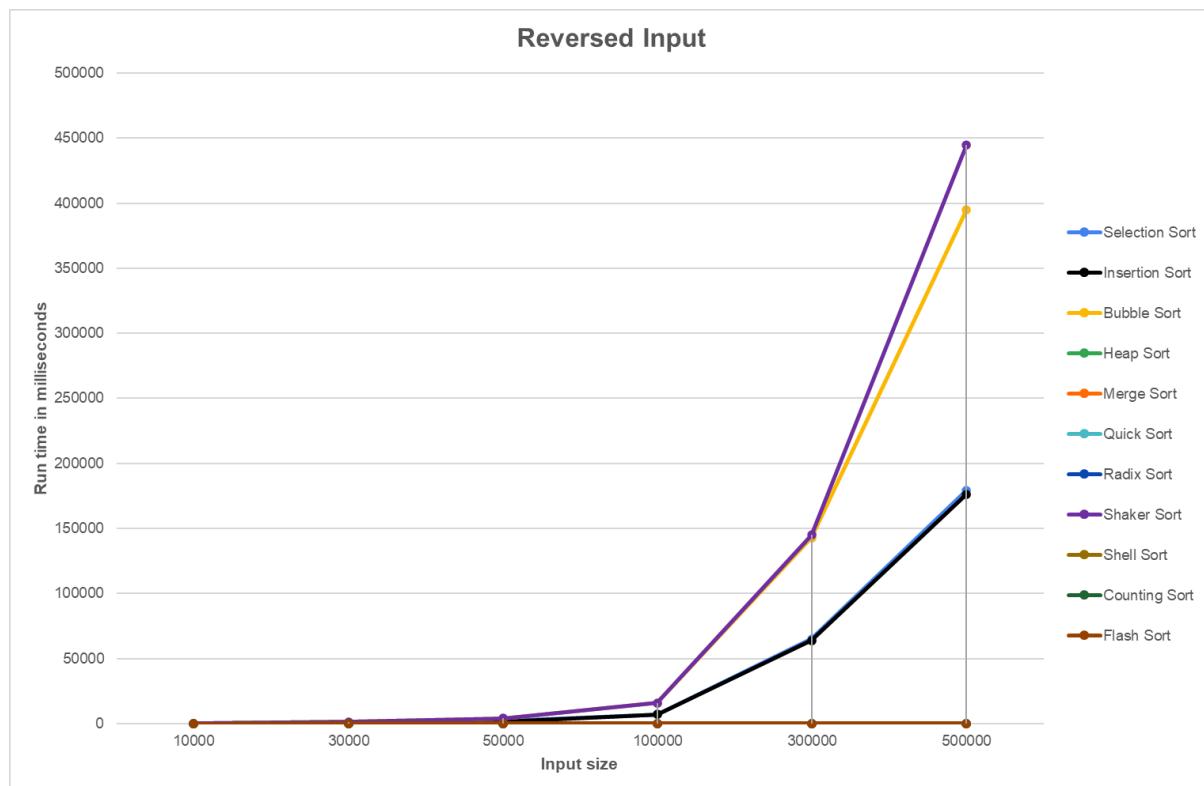
Khi quan sát số phép so sánh của các thuật toán còn lại ở hình ..., Quick Sort, Shaker Sort, Bubble Sort là ba thuật toán có số phép so sánh nhỏ nhất so với các thuật toán khác. Trong đó Heap Sort và Merge Sort có số phép so sánh tương đối cao trong nhóm còn lại này.

4.4. Bộ dữ liệu sắp xếp ngược

Bảng dữ liệu:

Data order: REVERSED DATA						
Data size	10000		30000		50000	
Resulting statics	Running time	Comparisons	Running time	Comparisons	Running time	Comparisons
Selection Sort	104,074	100009999	657,546	900029999	1791,7	2500049999
Insertion Sort	99,1823	100009999	637,42	900029999	1750,45	2500049999
Bubble Sort	186,554	100019998	1416,53	900059998	3906,11	2500099998
Heap Sort	3,34863	606771	8,98162	2063324	13,3368	3612724
Merge Sort	3,51167	670196	8,25512	2186817	12,7613	3844093
Quick Sort	0,416083	20003	1,1515	60003	1,98092	100003
Radix Sort	2,19008	140056	7,07221	510070	10,196	850070
Shaker Sort	187,566	100005001	1428,87	900015001	3943,41	2500025001
Shell Sort	1,34433	172578	3,479	567016	5,95512	1047305
Counting Sort	0,212291	50004	0,745917	150004	1,17708	250004
Flash Sort	0,561708	79249	1,38933	237749	2,10971	396249

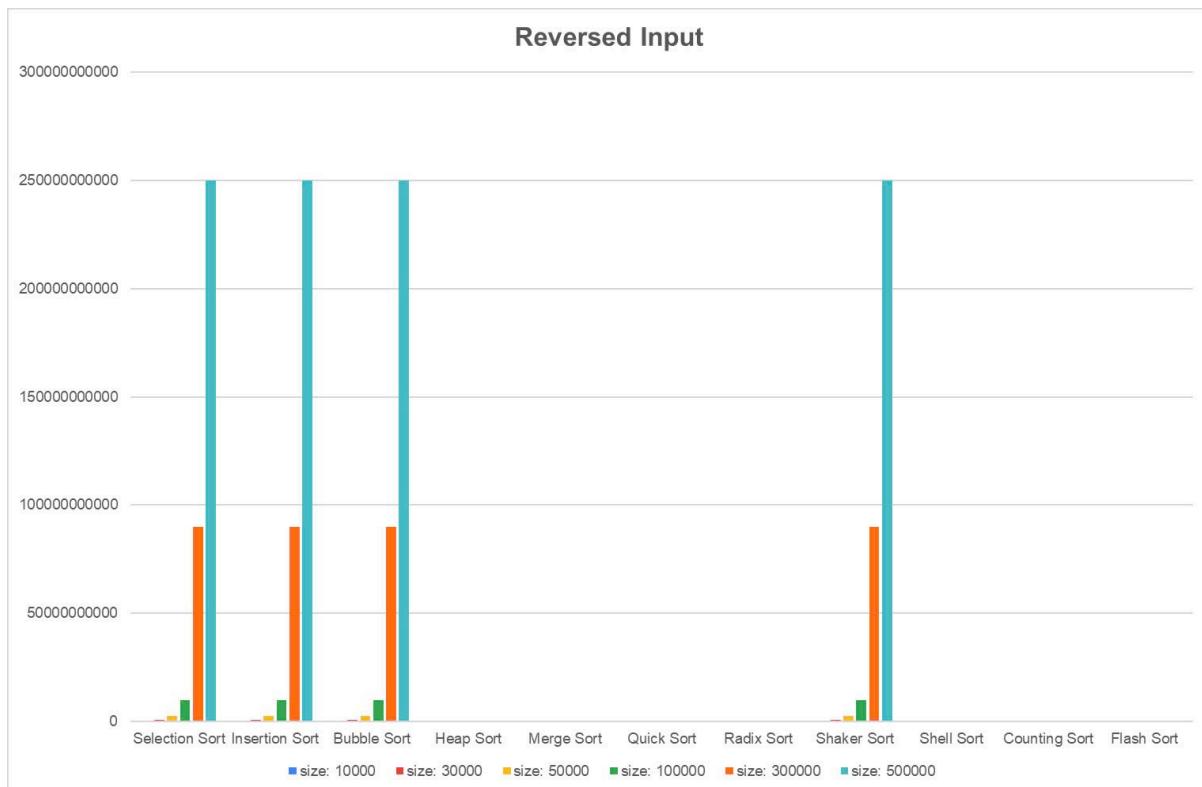
Data order: REVERSED DATA						
Data size	100000		300000		500000	
Resulting statics	Running time	Comparisons	Running time	Comparisons	Running time	Comparisons
Selection Sort	7221,57	10000099999	65246,1	90000299999	179327	250000499999
Insertion Sort	7041,62	10000099999	63774	90000299999	176277	250000499999
Bubble Sort	15827	10000199998	142871	90000599998	394962	250000999998
Heap Sort	13,818	7718943	43,2604	25569379	73,0413	44483348
Merge Sort	20,3942	8138127	43,124	2684750	70,2209	45567832
Quick Sort	2,45417	200003	5,36267	600003	9,09146	1000003
Radix Sort	15,8388	1700070	36,4056	6000084	55,7058	1000084
Shaker Sort	15966,7	10000050001	145067	90000150001	444948	250000250001
Shell Sort	9,75979	2244585	19,6884	7300919	31,3055	12428778
Counting Sort	1,67342	500004	3,23246	1500004	5,19958	2500004
Flash Sort	3,48183	792499	6,66737	2377499	9,82083	3962499

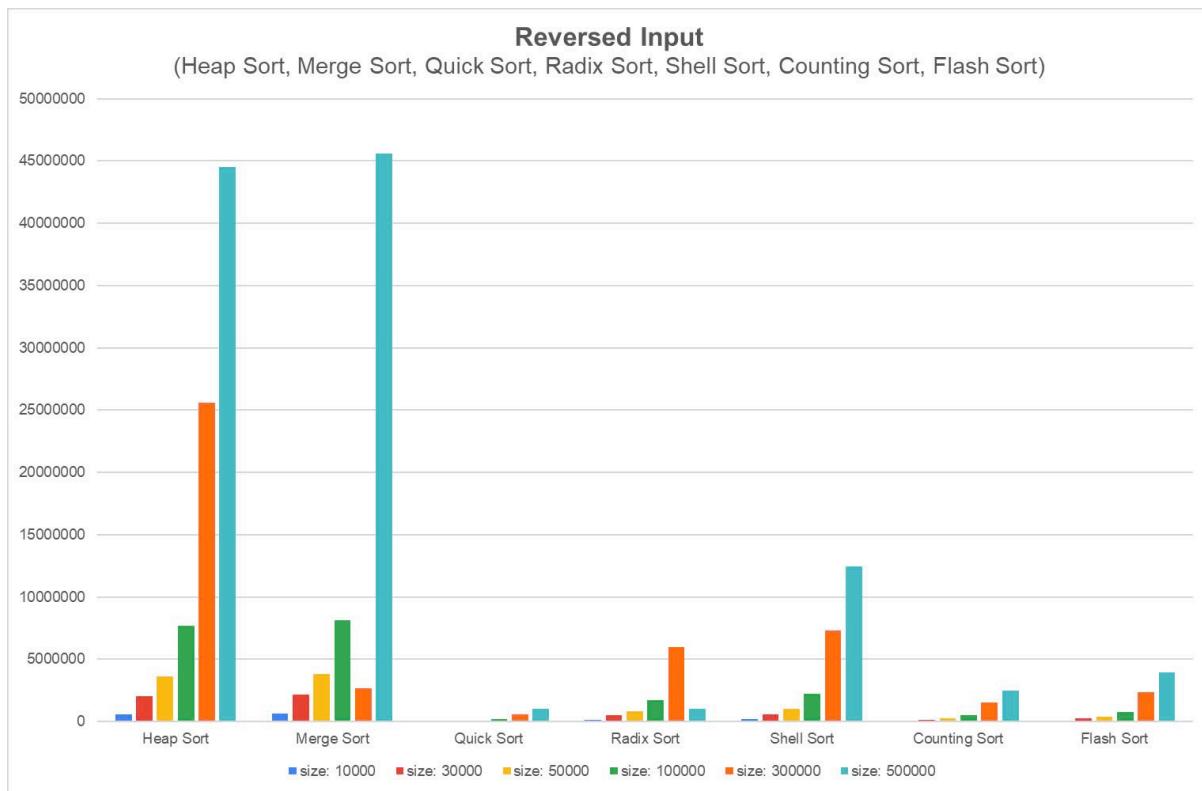
Hình 39: Bảng dữ liệu sắp xếp ngược**Hình 40: Biểu đồ đường dữ liệu sắp xếp ngược**

Nhận xét:

- Bubble Sort và Shaker Sort: Luôn thể hiện hiệu suất tệ nhất trên tất cả các kích thước dữ liệu. Điều này phù hợp với độ phức tạp thời gian $O(n^2)$ dự kiến trong trường hợp xấu nhất, xảy ra khi dữ liệu đã được sắp xếp theo thứ tự ngược.
- Selection Sort và Insertion Sort: Mặc dù cũng có độ phức tạp thời gian $O(n^2)$, nhưng chúng thể hiện hiệu suất tốt hơn đáng kể so với Bubble Sort và Shaker Sort. Điều này cho thấy rằng việc triển khai của chúng có thể được tối ưu hóa hoặc các hệ số hằng trong độ phức tạp thời gian của chúng thấp hơn.
- Counting Sort: Thể hiện hiệu suất vượt trội, đặc biệt với các kích thước dữ liệu nhỏ. Tuy nhiên, yêu cầu bộ nhớ của nó tăng đáng kể với các bộ dữ liệu có kích thước lớn, vì nó cần phân bổ một mảng để đếm số lần xuất hiện của mỗi phần tử.
- Radix Sort: Cân bằng tốt giữa hiệu suất và sử dụng bộ nhớ. Thuật toán mở rộng tốt với kích thước dữ liệu lớn và có thời gian chạy nhanh.
- Merge Sort, Heap Sort và Quick Sort: Các thuật toán này thường duy trì hiệu suất tương đối ổn định trên các kích thước dữ liệu khác nhau. Vì vậy, chúng được xem là các thuật toán sắp xếp phổ biến do khả năng thích ứng cao này.

Nhìn chung, Merge Sort, Heap Sort và Quick Sort được sử dụng phổ biến hơn do khả năng thích ứng với các thứ tự mảng khác nhau, nhưng khi xét trong dữ liệu đảo ngược, Bubble Sort là chậm nhất và Counting Sort là nhanh nhất.





Hình 41: Biểu đồ cột dữ liệu sắp xếp ngược

Nhận xét:

Selection Sort, Insertion Sort và Bubble Sort có số lượng so sánh cao nhất. Mặc dù Shaker Sort có ít phép so sánh hơn so với các thuật toán $O(n^2)$ khác trong các bộ dữ liệu phía trên, nhưng số lượng phép so sánh của nó trong bộ dữ liệu đảo ngược lại tương đối tương đồng với ba thuật toán trên, vì đây là trường hợp xấu nhất của Shaker Sort.

Counting Sort có số lượng phép so sánh ít nhất, Flash Sort và Radix Sort cũng cho số phép so sánh gần như tương đương.

Mặc dù, dữ liệu đảo ngược thường là trường hợp xấu nhất của Quick Sort và số lượng phép so sánh phải là rất lớn. Tuy nhiên, nhờ việc chọn phần tử giữa làm pivot, chúng ta gần như có thể tránh được điều này, đây là một trong những cách cải tiến của nhóm đối với thuật toán này.

4.5. Nhận xét tổng quan

Trong phần này, nhóm sẽ tóm tắt về độ phức tạp thời gian của các thuật toán sắp xếp (trong trường hợp trung bình), sau đó đưa ra ý kiến về các thuật toán nào là ổn định và không ổn định.

- $O(n^2)$:

- Các thuật toán có độ phức tạp thời gian $O(n^2)$ bao gồm Selection Sort, Insertion Sort, Bubble Sort và Shaker Sort. Mặc dù các thuật toán này mất khá nhiều thời gian xử lý khi số lượng phần tử đầu vào lớn, nhưng nếu cần một thuật toán sắp xếp đơn giản và chính xác, chúng sẽ là sự lựa chọn nên được ưu tiên vì dễ cho việc lập trình.
- Tuy nhiên, đối với dữ liệu gần như đã sắp xếp hoặc sắp xếp sẵn, Insertion Sort và Shaker Sort có thể vượt trội hơn không chỉ so với Selection Sort và Bubble Sort mà còn so với các thuật toán khác.
- Ôn định: Theo quan điểm của nhóm, Insertion Sort, Bubble Sort và Shaker Sort là ổn định.
- Không ổn định: Theo quan điểm của nhóm, Selection Sort là không ổn định.

• $O(n)$:

- Các thuật toán có độ phức tạp thời gian $O(n)$ bao gồm Flash Sort, Counting Sort và Radix Sort. Tất cả đều tiêu tốn không gian để tối ưu hóa thời gian chạy thuật toán.
- Ôn định: Theo quan điểm của nhóm, Counting Sort và Radix Sort, Flash Sort đều ổn định.

• $O(n \log n)$:

- Các thuật toán có độ phức tạp thời gian này bao gồm Heap Sort, Merge Sort, Quick Sort và Shell Sort.
- Heap Sort và Shell Sort không yêu cầu không gian bộ nhớ bổ sung. Tuy nhiên, Heap Sort thực tế được ưu tiên hơn vì nó giữ được độ phức tạp $O(n \log n)$ trong mọi trường hợp.
- Merge Sort và Quick Sort cần yêu cầu không gian bộ nhớ bổ sung, nhưng Merge Sort thường yêu cầu nhiều bộ nhớ hơn so với Quick Sort. Nếu Quick Sort không chọn pivot xấu, nó luôn chạy nhanh hơn Merge Sort. Do đó, mặc dù Merge Sort luôn có độ phức tạp $O(n \log n)$ trong mọi trường hợp và Quick Sort có thể có độ phức tạp $O(n^2)$ với trường hợp xấu nhất, nhưng trong thực tế Quick Sort vẫn được sử dụng nhiều hơn Merge Sort.
- Ôn định: Theo quan điểm của nhóm, Quick Sort là ổn định nhất. Merge Sort và Shell Sort tương đối ổn định.
- Không ổn định: Theo quan điểm của nhóm, Heap Sort là không ổn định.

5. CÁC LUU Ý VỚI BÀI NỘP

5.1. Tổ chức dự án

Dự án của chúng em được tổ chức thành bốn tệp mã nguồn chính như sau:

- **main.cpp**: Tệp này chứa mã nguồn thực thi chính của chương trình. Trong tệp main.cpp, nhóm thực hiện các thao tác chính để chạy các thuật toán và thực hiện các yêu cầu dựa trên tham số dòng lệnh.
- **function.h**: Tệp này là nơi khai báo tất cả các hàm và thư viện được sử dụng trong chương trình. Các định nghĩa về kiểu dữ liệu, đối số và kiểu trả về của các hàm được mô tả tại đây.
- **DataGenerator.cpp**: Tệp này chứa các hàm do giảng viên cung cấp sẵn. Những hàm này hỗ trợ việc sinh dữ liệu đầu vào cho chương trình, như tạo dữ liệu ngẫu nhiên, chuỗi dữ liệu mẫu hoặc dữ liệu thử nghiệm, giúp nhóm kiểm tra và thử nghiệm các thuật toán.
- **function.cpp**: Tệp này chứa các định nghĩa và triển khai các thuật toán mà nhóm đã khai báo trong function.h. Các hàm tại đây thực hiện các thao tác cốt lõi của chương trình, như sắp xếp, so sánh hoặc các phép toán khác tùy vào yêu cầu của người dùng.

Nhóm tổ chức mã nguồn như vậy là để đảm bảo tính rõ ràng, dễ dàng hơn trong việc quản lý dự án và thuận tiện cho việc bảo trì mã nguồn trong tương lai.

5.2. Lưu ý trong mã nguồn

Thư viện “Chrono” được sử dụng để tính thời gian chạy thuật toán.

Thư viện “Vector” được sử dụng để hỗ trợ viết các hàm sắp xếp.

6. TÀI LIỆU THAM KHẢO

Ý tưởng và mã giả của Insertion sort:

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, “Sorting and Order Statistic” in *Introduction to Algorithms*, Cambridge, MA: The MIT Press, 2009, pp. 16–23.

Độ phức tạp của Insertion:

[2] GeeksforGeeks. (2025, Jan. 2). *Insertion Sort – Data Structures and Algorithms Tutorials - GeeksforGeeks*. [Trực tuyến]. Có sẵn:
<https://www.geeksforgeeks.org/insertion-sort-algorithm/>.

Nội dung tham khảo ý tưởng của Heap Sort

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, “Sorting and Order Statistic” in *Introduction to Algorithms*, Cambridge, MA: The MIT Press, 2009, pp. 151–162.

Độ phức tạp của Heap Sort:

[4] GeeksforGeeks. (2025, Jan. 2) "Heap Sort – Data Structures and Algorithms Tutorials" *GeeksforGeeks*. [Trực tuyến]. Có sẵn:
<https://www.geeksforgeeks.org/heap-sort/#complexity-analysis-of-heap-sort>.

[5] T. H. Nhi and D. A. Đức, *Nhập môn cấu trúc dữ liệu và thuật toán*. TP. Hồ Chí Minh: Xưởng in trường Đại học Khoa học Tự nhiên, 2003.

Mã giả của Flash Sort

[6] CodeLearn. haiduco147. (2021, Mar. 18). *Flash Sort - Thuật Toán Sắp xếp thành thách*. [Trực tuyến]. Có sẵn:
<https://codelearn.io/sharing/flash-sort-thuat-toan-sap-xep-than-thach>.

Ý tưởng, mã giả và độ phức tạp của Shell Sort

[7] GeeksforGeeks. (2024, Jun. 11). *Shell Sort – Data Structures and Algorithms Tutorials - GeeksforGeeks*. [Trực tuyến]. Có sẵn:
<https://www.geeksforgeeks.org/shell-sort/>.

[8] Github.HaiDuc0147.(2020). *Sorting Algorithm*. [Online]. Có sẵn:
<https://github.com/HaiDuc0147/SortingAlgorithm/blob/main/reportSort/Sort.cpp>

Độ phức tạp thời gian của Counting Sort

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, “Sorting and Order Statistic” in *Introduction to Algorithms*, Cambridge, MA: The MIT Press, 2009, pp. 194–197.

Mã giả và định nghĩa của Counting Sort

[10] K. Bajpai and A. Kots, “Implementing and Analyzing an Efficient Version of Counting Sort (E-Counting Sort),” International Journal of Computer Applications vol. 98, no. 9, Jul. 2014, ISSN 0975-8887.

Code Counting Sort và độ phức tạp không gian

[11] GeeksforGeeks. (2024, Aug. 6). Counting Sort – Data Structures and Algorithms Tutorials. [Trực tuyến]. Có sẵn: [Counting Sort - Hướng dẫn về Cấu trúc dữ liệu và Thuật toán - GeeksforGeeks](#).

Các bước của Counting Sort

[12] Programiz. *Counting Sort Algorithm*. [Trực tuyến]. Có sẵn: <https://www.programiz.com/dsa/counting-sort>. [Accessed: Jan. 2, 2025]

Định nghĩa và độ phức tạp của Counting Sort

[13] V. Mansotra and K. Sourabh, "Implementing Bubble Sort Using a New Approach," *Proceedings of the 5th National Conference; INDIACom-2011 Computing For Nation Development*, Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi, India, Mar. 10–11, 2011, ISBN 978-93-80544-00-7, ISSN 0973-7529.

Code và các bước của Shaker Sort

[14] Quốc Khánh Trần. (2022, Jun. 14). *Cấu trúc dữ liệu và giải thuật: Giải thuật sắp xếp rung lắc (Shaker Sort)*, DAYNHAUHOC. [Trực tuyến]. Có sẵn: [Cấu trúc dữ liệu và giải thuật: Giải thuật sắp xếp rung lắc \(Shaker sort\) - share / writes - Day Nhau Hoc](#)

Mã giả của Shaker Sort

[15] A. H. Elkahlout and A. Y. A. Maghari, "A comparative study of sorting algorithms: Comb, Cocktail and Counting Sorting," *International Research Journal of Engineering and Technology (IRJET)*, vol. 4, no. 1, pp. 1–7, Jan. 2017. [Online]. Available: www.irjet.net

Định nghĩa Merge Sort, Radix Sort, code Merge Sort, ví dụ Merge Sort, Radix Sort

[16] T. H. Nhi and D. A. Đức, *Nhập môn cấu trúc dữ liệu và thuật toán*. TP. Hồ Chí Minh: Xưởng in trường Đại học Khoa học Tự nhiên, 2003.

Code Radix Sort

[17] GeeksforGeeks. (2024, Nov. 25). *Radix Sort – Data Structures and Algorithms Tutorials - GeeksforGeeks*. [Trực tuyến]. Có sẵn: <https://www.geeksforgeeks.org/radix-sort/>.

Mã giả Radix Sort

[18] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, “Sorting and Order Statistic” in *Introduction to Algorithms*, Cambridge, MA: The MIT Press, 2009, pp. 211–214.

Mã giả Quick Sort

[19] D. T. V. Dharmajee Rao and B. Ramesh, "Experimental Based Selection of Best Sorting Algorithm," *International Journal of Modern Engineering Research*, vol. 2, no. 4, pp. 2912–2917, July-Aug 2012.

Định nghĩa của Bubble Sort, Selection Sort và Quick Sort

[20] ResearchGate. A. Author(s).(1996, Mar). “A Comparative Study of Different Types of Comparison Based Sorting Algorithms”. *Data Structure*. [Trực tuyến]. Có sẵn: <https://www.researchgate.net/publication/306258293>

Mã giả Bubble Sort, Selection Sort

[21] P. Ganapathi and R. Chowdhury, “Parallel Divide-and-Conquer Algorithms for Bubble Sort, Selection Sort and Insertion Sort.” in *The Computer Journal*, vol. 64, no. 12, pp. Jun-Nov 2021.

Ý tưởng của Flash Sort

[22] Vladimir Marek , “A Comparative Study of Different Types of Comparison-Based Sorting Algorithms in Data Structure,” ResearchGate. [Trực tuyến]. Có sẵn: <https://www.neubert.net/FSOIntro.html>.

[23] CodeLearn. haiduc0147.(2021, Mar. 18). *Flash Sort - Thuật Toán Sắp Xếp Thành Thanh*. [Trực tuyến]. Có sẵn:

<https://codelearn.io/sharing/flash-sort-thuat-toan-sap-xep-than-thanh>.

Độ phức tạp của Flash Sort

[24] C. Nguyễn.Chung Nguyễn.(2017). *Kính thưa các thuật toán sort*. [Trực tuyến]. Có sẵn: https://chungnguyen.xyz/posts/kinh-thua-cac-thuat-toan-sort#google_vignette.

7. LỜI KẾT

Sau khi hoàn thành dự án, nhóm em rút ra được nhiều bài học quý giá về cả kiến thức chuyên môn lẫn kỹ năng thực hành. Việc nghiên cứu sâu về các thuật toán sắp xếp không chỉ giúp nhóm hiểu rõ hơn về nguyên lý hoạt động của chúng, mà còn nâng cao khả năng phân tích và xử lý dữ liệu thông qua việc tạo các biểu đồ minh họa.

Mặc dù đã nỗ lực tối đa trong quá trình thực hiện, nhưng nhóm em nhận thức được rằng dự án vẫn còn những điểm cần được cải thiện và hoàn thiện thêm. Nhóm mong muốn nhận được những góp ý từ thầy để có thể tiếp tục phát triển và nâng cao chất lượng cho các dự án tương lai.

Nhóm xin gửi lời cảm ơn sâu sắc đến thầy đã tận tình hướng dẫn, chia sẻ kiến thức quý báu và dành thời gian đánh giá dự án. Những kinh nghiệm và kiến thức thu được từ dự án này sẽ là nền tảng quan trọng cho sự phát triển chuyên môn của mỗi thành viên trong nhóm.