Trường Đại học Khoa học tự nhiên - ĐHQG HCM

fit@hcmus

Khoa Công nghệ thông tin

BÁO CÁO ĐỒ ÁN

Cấu trúc dữ liệu và Giải thuật

Các thuật toán sắp xếp

Ngày 06 tháng 12 năm 2024

Lớp 23CTT3

Nhóm 01

23120233 - Nguyễn Lê Hữu Điền

23120255 - Lê Tấn Hiệp

23120262 - Tống Dương Thái Hoà

23120264 - Nguyễn Phúc Hoàng

Giảng viên hướng dẫn

Thầy Văn Chí Nam

Cô Phan Thị Phương Uyên

Mục lục

1 Giới thiệu

1.1 Chủ đề

Dự án này nghiên cứu 12 thuật toán sắp xếp bao gồm Selection Sort, Insertion Sort, Bubble Sort, Shaker Sort, Shell Sort, Binary Insertion Sort, Counting Sort, Radix Sort, Merge Sort, Quick Sort, Heap Sort và Flash Sort trên mảng với các tính chất dữ liệu và kích thước khác nhau. Đồng thời, dự án được triển khai bằng ngôn ngữ lập trình C++ cung cấp đánh giá về độ phức tạp với tính linh hoạt của các thuật toán.

1.2 Mục tiêu

Mục tiêu của dự án bao gồm:

- Giới thiệu các thuật toán sắp xếp.
- Phân tích và đánh giá độ phức tạp của các thuật toán sắp xếp.
- Triển khai và so sánh các thuật toán thông qua 5 lệnh được yêu cầu.
- Tạo trực quan hóa các thuật toán bằng biểu đồ minh họa.

1.3 Phương pháp nghiên cứu

Dưới đây là các bước chúng tôi thực hiện phương pháp nghiên cứu:

- Trình bày ý tưởng thuật toán.
- Nghiên cứu thuật toán.
- Phân tích độ phức tạp của thuật toán.
- Triển khai các thuật toán.
- Thu thập kết quả và tạo biểu đồ để đánh giá thuật toán

1.4 Thành tựu

Kết quả đạt được của dự án chúng tôi bao gồm:

- Hoàn thành việc triển khai 12 thuật toán sắp xếp.
- Viết thành công các chương trình sử dụng 5 lệnh được cung cấp.
- Thực hiện phân tích và đánh giá độ phức tạp của các thuật toán.
- Hoàn thành việc trực quan hóa các thuật toán sắp xếp.

1.5 Hỗ trợ của giảng viên

Chúng tôi xin gửi lời cảm ơn chân thành và sâu sắc đến các thầy cô đã đồng hành cùng chúng tôi trong suốt quá trình thực hiện dự án này. Chúng tôi đặc biệt cảm kích sự hỗ trợ quý báu từ:

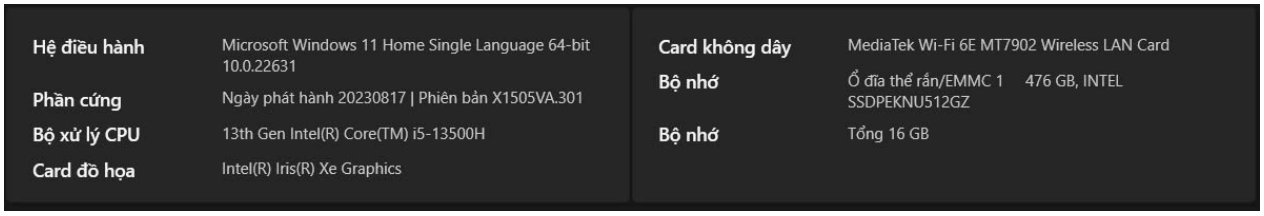
Thầy **Văn Chí Nam** (Giảng viên lý thuyết)

Cô **Phan Thị Phương Uyên** (Giảng viên thực hành)

Sự hướng dẫn tận tình và kiến thức sâu rộng của các thầy cô đã giúp chúng tôi vượt qua những khó khăn và thử thách trong suốt quá trình thực hiện đồ án. Đồ án này không chỉ là cơ hội để chúng tôi nâng cao kiến thức chuyên môn mà còn là dịp để học hỏi những bài học quý giá về sự kiên nhẫn, sáng tạo và trách nhiệm trong công việc. Xin cảm ơn các thầy cô vì đã truyền cảm hứng và tạo điều kiện để chúng tôi có thể phát triển cả về lý thuyết lẫn thực hành. Sự giúp đỡ của các thầy cô sẽ luôn là động lực to lớn giúp chúng tôi tiến xa hơn trong hành trình học tập và sự nghiệp sau này.

1.6 Cấu hình máy

Chúng tôi đã triển khai và đánh giá các thuật toán bằng Code Block trên một hệ thống máy tính với các thông số kỹ thuật sau:



2 Trình bày thuật toán

Để dễ dàng cho việc theo dõi, chúng tôi sẽ thống nhất một số quy ước, tên gọi sau:

1. **ARRAY**: Mảng đầu vào.
2. **n**: Số lượng phần tử trong mảng.
3. **MIN**: Phần tử nhỏ nhất của mảng.
4. **MAX**: Phần tử lớn nhất của mảng.
5. **Best-case**: Trường hợp tốt nhất.
6. **Average-case**: Trường hợp trung bình.
7. **Worst-case**: Trường hợp tệ nhất.
8. **Sắp xếp tăng (hoặc giảm)**: Tuỳ vào việc chọn tìm MIN (hoặc MAX) của thuật toán.

2.1 Selection Sort

Selection Sort là thuật toán sắp xếp dựa trên so sánh. Thuật toán này sắp xếp một mảng bằng cách chọn đi chọn lại MIN (hoặc MAX) từ phần chưa được sắp xếp và hoán đổi nó với phần tử chưa được sắp xếp đầu tiên. Quá trình này tiếp tục cho đến khi toàn bộ mảng được sắp xếp. [1]

2.1.1 Ý tưởng

Đầu tiên chúng ta tìm MIN (hoặc MAX) và hoán đổi nó với phần tử đầu tiên. Bằng cách này, chúng ta sẽ có được MIN (hoặc MAX) ở đúng vị trí của nó. Sau đó, chúng ta tìm MIN (hoặc MAX) trong số các phần tử còn lại và di chuyển nó đến vị trí chính xác bằng cách hoán đổi. Chúng ta tiếp tục làm như vậy cho đến khi di chuyển được tất cả các thành phần đến đúng vị trí.

2.1.2 Mô tả từng bước

Bước 1: Chia mảng ra làm 2 phần là *"đã sắp xếp"* (khởi tạo rỗng) và *"chưa sắp xếp"* (Toàn bộ ARRAY).

Bước 2: Tìm MIN (hoặc MAX) trong mảng *"chưa sắp xếp"*.

Bước 3: Thực hiện hoán đổi MIN (hoặc MAX) với phần tử đầu tiên trong phần *"chưa sắp xếp"*.

Bước 4: Thêm phần tử đầu tiên của mảng *"chưa sắp xếp"* vào cuối mảng *"đã sắp xếp"* và xoá phần tử đầu tiên đó khỏi mảng *"chưa sắp xếp"*.

Bước 5: Lặp lại bước 2 đến bước 4 cho đến khi mảng *"chưa sắp xếp"* rỗng.

Ví dụ:

Bước	ARRAY	Mô tả
1	9, 14, 28, 7, 12, 11, 32	Cấp phát mảng *"chưa sắp xếp"*
2	9, 14, 28, **7**, 12, 11, 32	Tìm MIN trong mảng *"chưa sắp xếp"*
3	**7**, 14, 28, **9**, 12, 11, 32	Hoán vị phần tử đầu tiên của mảng *"chưa sắp xếp"* và MIN
4	**7**, 14, 28, **9**, 12, 11, 32	Khi này phần tử 7 sẽ được xoá khỏi mảng **"chưa sắp xếp"**. Tiếp tục tìm MIN của mảng **"chưa sắp xếp"**
5	**7**, **9**, 28, **14**, 12, 11, 32	Hoán vị phần tử đầu tiên của mảng *"chưa sắp xếp"* và MIN
Tiếp tục lặp lại cho đến khi mảng *"chưa sắp xếp"* **rỗng**		
...	**7, 9, 11, 14, 28, 32**	Mảng sau khi đã sắp xếp tăng dần

2.1.3 Phân tích thuật toán

- Độ phức tạp về thời gian: Selection Sort là thuật toán có độ phức tạp về **thời gian** trong cả ba trường hợp Best-case, Average-case và Worst-case là như nhau là $O(n^2)$.
- Độ phức tạp về không gian: Selection Sort chỉ sử dụng thêm biến hằng số MIN (hoặc MAX) mà không phụ thuộc vào kích thước đầu vào nên độ phức tạp về **không gian** là $O(1)$
- Tính ổn định: Selection Sort là thuật toán không ổn định *(unstable)* do các phần tử trong mảng được hoán đổi liên tục nên không thể đảm bảo được vị trí tương đối ban đầu của các phần tử giống nhau.

2.1.4 Cải tiến và biến thể

Có hai loại biến thể đồng thời là cải tiến của Selection Sort:

Double Selection Sort: Thuật toán này tìm đồng thời cả phần tử nhỏ nhất và lớn nhất trong mỗi lần lặp và hoán đổi chúng với phần tử đầu tiên và phần tử cuối cùng của phần chưa sắp xếp. Điều này giúp giảm số lần lặp và số lần hoán đổi, làm cho thuật toán hiệu quả hơn so với selection sort cơ bản. [2]

Bingo Sort: Vì Selection Sort không có tính ổn định (unstable) nên để cải thiện được điều này, mỗi khi gặp phần tử giống với MIN, thay vì ta hoán vị hai phần tử thì ta sử dụng phương pháp chèn phần tử MIN vào vị trí đầu tiên của mảng "chưa sắp xếp"(gần giống với Insertion Sort). [2]

2.2 Insertion Sort

Insertion Sort là một thuật toán sắp xếp đơn giản dựa trên việc xây dựng mảng đã sắp xếp bằng cách chèn phần tử vào đúng vị trí một cách tuần tự.

2.2.1 Ý tưởng

Insertion Sort lấy ý tưởng tương tự như việc sắp xếp một bộ bài 52 lá. Bắt đầu với không lá bài nào trên tay trái và tất cả cá lá đều được úp dưới mặt bàn. Lấy một lá trên bàn và thực hiện so sánh với từng phần tử trên tay trái các lá từ phải sang trái. Sau đó, chèn vào đúng vị trí sao cho các lá bài bên tay trái luôn được sắp xếp. [3]

2.2.2 Mô tả từng bước

Bước 1: Bắt đầu từ phần tử thứ hai trong mảng (giả định phần tử đầu tiên đã được sắp xếp).

Bước 2: Lấy phần tử hiện tại làm ***key***.

Bước 3: So sánh ***key*** với các phần tử trước đó trong mảng *"đã sắp xếp"* từ phải sang trái.

- Nếu phần tử đang so sánh lớn hơn ***key*** thì dịch chuyển phần tử đó sang phải.
- Ngược lại, nếu nhỏ hơn thì dừng so sánh.

Bước 4: Chèn ***key*** vào vị trí đúng trong phần đã sắp xếp.

Bước 5: Lặp lại bước 2 đến bước 4 cho đến khi toàn bộ mảng được sắp xếp.

Ví dụ:

Bước	ARRAY	Mô tả
1	**12**, 4, 27, 8, 32, 16	Khởi tạo mảng, coi phần tử đầu tiên đã được sắp xếp
2	**12**, **4**, 27, 8, 32, 16	key là phần tử thứ 2, so sánh key với các phần tử trong mảng *"đã sắp xếp"*
3	**12**, **12**, 27, 8, 32, 16	Dịch chuyển các phần tử lớn hơn key sang phải
4	**4**, **12**, 27, 8, 32, 16	Chèn key vào vị trí đúng trong mảng *"đã sắp xếp"*
5	**4**, **12**, **27**, 8, 32, 16	key là phần tử thứ 3, so sánh key với các phần tử trong mảng *"đã sắp xếp"*
6	**4**, **12**, **27**, 8, 32, 16	Dịch chuyển các phần tử lớn hơn key sang phải
7	**4**, **12**, **27**, 8, 32, 16	Chèn key vào vị trí đúng trong mảng *"đã sắp xếp"*
8	**4**, **12**, **27**, **8**, 32, 16	key là phần tử thứ 4, so sánh key với các phần tử trong mảng *"đã sắp xếp"*
9	**4**, **12**, **12**, **27**, 32, 16	Dịch chuyển các phần tử lớn hơn key sang phải
10	**4**, **8**, **12**, **27**, 32, 16	Chèn key vào vị trí đúng trong mảng *"đã sắp xếp"*
Lặp lại bước 2 đến bước 4 cho đến khi left >= right.		
...	**4, 8, 12, 16, 27, 32**	Mảng sau khi được sắp xếp

2.2.3 Phân tích thuật toán

Độ phức tạp về thời gian:

- Best-case: Khi mảng đã được sắp xếp sẵn, độ phức tạp là $O(n)$
- Average-case: Gần như các phần tử ở đúng vị trí trong mảng *"đã sắp xếp"*, độ phức tạp là $O(n^2)$
- Worst-case: Mảng được sắp xếp ngược lại, độ phức tạp là $O(n^2)$

Độ phức tạp về không gian: Vì đây là thuật toán tại chỗ (không tạo thêm mảng phụ) nên độ phức tạp là $O(1)$.

Tính ổn định: Insertion Sort là một thuật toán sắp xếp ổn định (stable). Vì thuật toán này chèn vào vị trí thích hợp (vòng lặp lùi dừng khi gặp số bé hơn hoặc bằng).

2.2.4 Cải tiến và biến thể

Có hai loại biến thể đồng thời là cải tiến của Insertion Sort:

Binary Insertion Sort: Sử dụng tìm kiếm nhị phân để xác định vị trí chèn "key". Điều này giảm thời gian tìm vị trí từ O(n) xuống O(logn).

Shell Sort: Mở rộng ý tưởng Insertion Sort bằng cách sắp xếp các phần tử cách nhau một khoảng (gap), sau đó giảm gap dần về 1.

2.3 Bubble Sort

Bubble Sort là một thuật toán liên tục đi qua danh sách cần sắp xếp, so sánh các phần tử liền kề và hoán đổi các phần tử đó nếu sai thứ tự. Lặp lại quá trình cho đến khi toàn bộ mảng được sắp xếp. Như vậy, việc hoán đổi liên tục giống như việc các phần tử cần hoán vị "nổi lên"và bị đẩy về một phía.

2.3.1 Ý tưởng

Bắt đầu từ đầu danh sách chưa được sắp xếp và "nổi bọt" phần tử MAX (hoặc MIN) về cuối danh sách thông qua các lần hoán đổi liền kề. Sau khi đưa phần tử MAX (hoặc MIN) về cuối mảng thì lần sau không xét phẩn tử cuối mảng nữa. Quá trình duyệt và hoán đổi này được lặp lại cho đến khi toàn bộ danh sách được sắp xếp theo thứ tự tăng dần (hoặc giảm dần).

2.3.2 Mô tả từng bước

Bước 1: Khởi tạo giá trị i bằng 1 tương ứng cặp thứ i và phần tử thứ i.

Bước 2: So sánh cặp i (gồm phần tử thứ i và $i+1$).

Bước 3: Nếu hai phần tử trong cặp sai thứ tự thì thực hiện hoán đổi.

Bước 4: Lặp lại bước 2 với giá trị $i+1$ cho đến khi i bằng kích thước mảng *"chưa sắp xếp"* trừ 1.

Bước 5: Giảm kích thước mảng *"chưa sắp xếp"* đi một phần tử.

Bước 6: Lặp lại bước 2 đến bước 5 cho đến khi toàn bộ mảng được sắp xếp.

Ví dụ:

Bước	ARRAY	Mô tả
1	**12**, **4**, 27, 8, 32, 16	Xét cặp 1 (gồm phần tử 1 và 2).
2	**4**, **12**, 27, 8, 32, 16	Thấy cặp 1 sai thứ tự, thực hiện hoán đổi.
3	4, **12**, **27**, 8, 32, 16	Xét cặp 2 (gồm phần tử 2 và 3), thấy đúng thứ tự.
4	4, 12, **27**, **8**, 32, 16	Xét cặp 3 (gồm phần tử 3 và 4).
5	4, 12, **8**, **27**, 32, 16	Thấy cặp 3 sai thứ tự, thực hiện hoán đổi.
6	4, 12, 8, **27**, **32**, 16	Xét cặp 4 (gồm phần tử 4 và 5), thấy đúng thứ tự.
7	4, 12, 8, 27, **32**, **16**	Xét cặp 5 (gồm phần tử 5 và 6).
8	4, 12, 8, 27, **16**, **32**	Thấy cặp 5 sai thứ tự, thực hiện hoán đổi.
9	4, 12, 8, 27, 16, **32**	MAX đã được "nổi bọt"về cuối và phần không có MAX là mảng *"chưa sắp xếp"*
Lặp lại các bước trên mảng *"chưa sắp xếp"* cho đến khi toàn bộ mảng được sắp xếp		
...	**4, 8, 12, 16, 27, 32**	Mảng sau khi được sắp xếp

2.3.3 Phân tích thuật toán

- **Độ phức tạp về thời gian:** Do số lần hoán đổi giới hạn bởi một hằng số (số phần tử trong mảng trừ 1), thời gian xử lý trong cả ba trường hợp đều có độ phức tạp là $O(n^2)$.
- **Độ phức tạp về không gian:** Bubble Sort hoạt động "in-place", tức là nó không yêu cầu thêm bộ nhớ ngoài, chỉ sử dụng một lượng không gian cố định cho các biến tạm thời và chỉ hoán vị các phần tử trong mảng nên độ phức tạp không gian là $O(1)$.
- **Tính ổn định:** Bubble Sort là một thuật toán ổn định *(stable)*. Điều này có nghĩa là nếu hai phần tử trong mảng có giá trị bằng nhau, thuật toán sẽ giữ nguyên thứ tự ban đầu của chúng sau khi sắp xếp.

2.3.4 Cải tiến và biến thể

Có ba loại biến thể đồng thời là cải tiến của Bubble Sort:

Improved Bubble Sort: Chúng ta có thể theo dõi xem có hoán đổi nào được thực hiện trong mỗi lần lặp. Nếu không có hoán đổi nào, điều này có nghĩa là mảng đã được sắp xếp, và chúng ta có thể kết thúc quá trình sắp xếp sớm. Điều này giúp tránh các vòng lặp không cần thiết và giảm độ phức tạp thời gian cho các mảng đã được sắp xếp một phần.

Shaker Sort (còn gọi là Cocktail Sort): là một phiên bản cải tiến của Bubble Sort, hoạt động bằng cách di chuyển qua danh sách theo cả hai hướng: từ đầu đến cuối và từ cuối về đầu. Điều này giúp giảm thiểu số lần lặp qua mảng và có thể giúp sắp xếp nhanh hơn so với Bubble Sort.

Odd-Even Sort: Là một biến thể của Bubble Sort, hoạt động bằng cách so sánh và hoán đổi tất cả các cặp có chỉ số lẻ trước, sau đó so sánh và hoán đổi tất cả các cặp có chỉ số chẵn. Quá trình này được lặp đi lặp lại giữa các lượt lẻ và chẵn cho đến khi mảng được sắp xếp hoàn toàn. Phương pháp này có thể được song song hóa vì các lượt lẻ và chẵn độc lập với nhau, điều này có thể cải thiện hiệu suất trong một số tình huống [4].

2.4 Shaker Sort

Shaker Sort hay còn được gọi là Cocktail Sort hoặc là Bidirectional Bubble Sort. Đây là một biến thể của Bubble Sort, thuật toán này so sánh và hoán đổi đồng thời hai vị trí đầu và cuối của mảng *"chưa sắp xếp"*.

2.4.1 Ý tưởng

Xuất phát từ Bubble Sort, nhưng Shaker Sort sẽ duyệt mảng từ cả hai phía luân phiên trong mỗi lần lặp, với phần tử lớn nhất chưa được sắp xếp sẽ nổi lên trên và tương tự, phần tử nhỏ nhất chưa được sắp xếp sẽ chìm xuống dưới sau mỗi lần lặp.

2.4.2 Mô tả từng bước

Bước 1: Khởi tạo hai giá trị ***left*** và ***right*** lưu lần lượt chỉ số của phần tử đầu tiên và cuối cùng trong mảng *"chưa sắp xếp"*.

Bước 2: Đưa phần tử **MIN** (hoặc **MAX**) của mảng *"chưa sắp xếp"* về đầu mảng.

- So sánh cặp bắt đầu từ ***left***.
- Nếu hai phần tử trong cặp sai thứ tự thì thực hiện hoán đổi.
- Lặp lại bước 2 với các cặp tiếp theo từ trái qua cho đến khi duyệt hết mảng *"chưa sắp xếp"*.

Bước 3: Giảm ***right*** đi một đơn vị.

Bước 4: Đưa phần tử **MAX** (hoặc **MIN**) của mảng *"chưa sắp xếp"* về cuối mảng.

- So sánh cặp bắt đầu từ ***right***.
- Nếu hai phần tử trong cặp sai thứ tự thì thực hiện hoán đổi.
- Lặp lại bước 2 với các cặp tiếp theo từ phải qua cho đến khi duyệt hết mảng *"chưa sắp xếp"*.

Bước 5: Tăng ***left*** lên một đơn vị.

Bước 6: Lặp lại bước 2 đến bước 5 cho đến khi ***left*** bằng ***right***.

Ví dụ:

Bước	left	right	ARRAY	Mô tả
1	1	6	9, 4, 2, 1, 7, 5	Khởi tạo left là chỉ số phần tử đầu tiên và right là chỉ số phần tử cuối cùng
2.1	1	6	**9**, **4**, 2, 1, 7, 5	So sánh 9 và 4 thấy sai vị trí, thực hiện hoán đổi
2.2	1	6	4, **9**, **2**, 1, 7, 5	So sánh 9 và 2 thấy sai vị trí, thực hiện hoán đổi
2.3	1	6	4, 2, **9**, **1**, 7, 5	So sánh 9 và 1 thấy sai vị trí, thực hiện hoán đổi
2.4	1	6	4, 2, 1, **9**, **7**, 5	So sánh 9 và 7 thấy sai vị trí, thực hiện hoán đổi
2.5	1	6	4, 2, 1, 7, **9**, **5**	So sánh 9 và 5 thấy sai vị trí, thực hiện hoán đổi
3	1	5	4, 2, 1, 7, 5, **9**	MAX đã được đưa về cuối mảng, thực hiện giảm right đi một đơn vị
4.1	1	5	4, 2, 1, **7**, **5**, **9**	So sánh 5 và 7 thấy sai vị trí, thực hiện hoán đổi.
4.2	1	5	4, 2, **1**, **5**, 7, **9**	So sánh 5 và 1 đúng vị trí
4.3	1	5	4, **2**, **1**, 5 , 7, **9**	So sánh 1 và 2 thấy sai vị trí, thực hiện hoán đổi.
4.4	1	5	**4**, **1**, 2 ,5 , 7, **9**	So sánh 1 và 4 thấy sai vị trí, thực hiện hoán đổi.
5	2	5	**1**, 4, 2, 5, 7, **9**	MIN đã được đưa về đầu mảng, thực hiện tăng left lên một đơn vị.
Lặp lại các bước cho đến khi left bằng right				
...	3	3	**1, 2, 4, 5, 7, 9**	Mảng sau khi đã sắp xếp tăng dần.

2.4.3 Phân tích thuật toán

Độ phức tạp về thời gian:

- Best-case: Là $O(n^2)$ vì mảng so sánh $n(n-1)/2$ lần
- Average-case: Là $O(n^2)$ vì mảng so sánh $n(n-1)/2$ lần và gán trong khoảng từ 0 đến $n(n-1)/2$ lần vì phép so sánh là phép thống trị.
- Worst-case: Là $O(n^2)$ vì mảng so sánh $n(n-1)/2$ lần và gán $n(n-1)/2$ lần

Độ phức tạp về không gian: Thuật toán sử dụng các hằng số của bộ nhớ thêm và không dùng thêm mảng phụ nào nên độ phức tạp là $O(1)$

Tính ổn định: Shaker Sort là một thuật toán ổn định (stable). Vì với các phần tử thì thứ tự vẫn được giữ nguyên.

2.4.4 Cải tiến và biến thể

Đặt thêm một biến là flag để kiểm tra xem vòng lặp đó có thực hiện không. Nếu không có hoán vị diễn ra, tức là mảng đã được sắp xếp thì kết thúc vòng lặp, giảm đáng kể phép so sánh. Ngoài ra, Shaker Sort có thể kết hợp với các thuật toán hiệu quả hơn như Insertion Sort hoặc Quick Sort để xử lý các trường hợp mà Shaker Sort không hiệu quả.

2.5 Shell Sort

2.5.1 Ý tưởng

Shell Sort là một thuật toán cải tiến của Insertion Sort, được thiết kế để giảm thiểu thời gian thực thi do giảm thiểu số lần hoán đổi các phần tử khi làm việc với các mảng lớn. Thay vì so sánh và hoán đổi các phần tử liền kề, Shell Sort so sánh và hoán đổi các phần tử cách nhau một khoảng (gap). Bằng cách giảm dần khoảng cách này (gap) về 1, nó dần biến đổi thành Insertion Sort nhưng hiệu quả hơn, vì các phần tử đã gần như sắp xếp.

2.5.2 Mô tả từng bước

Bước 1: Chọn khoảng cách ***(gap)***. Bắt đầu với một giá trị ***gap*** lớn (thường bằng $n/2$ hoặc chọn các chuỗi tối ưu như chuỗi Knuth hay chuỗi Sedgewick).

Bước 2: Áp dụng Insertion Sort cho các phần tử cách nhau ***gap***.

Bước 3: Thực hiện bước 2 với các phần tử tiếp theo trong mảng tương ứng khoảng cách ***gap*** cho đến khi các phần tử trong mảng được sắp xếp theo ***gap***

Bước 4: Tiếp tục giảm ***gap*** (thường là chia đôi ***gap*** hoặc phần tử tiếp theo trong các chuỗi tối ưu) và quay lại bước 2 cho đến khi giá trị của ***gap*** bằng 1. Khi này, các phần tử gần như đã sắp xếp, thực hiện Insertion Sort ở lần cuối này sẽ hiệu quả.

Ví dụ:

Bước	gap	ARRAY	Mô tả
1	3	**64**, 34, 25, **12**, 22, 11, **90**	Chọn gap là 7 / 2 bằng 3
2	3	**12**, 34, 25, **64**, 22, 11, **90**	Sắp xếp các phần tử cách nhau 3 đơn vị
3.1	3	12, **34**, 25, 64, **22**, 11, 90	Chọn các phần tử tiếp theo trong mảng cách nhau gap
3.2	3	12, **22**, 25, 64, **34**, 11, 90	Sắp xếp các phần tử cách nhau 3 đơn vị
3.3	3	12, 22, **25**, 64, 34, **11**, 90	Chọn các phần tử tiếp theo trong mảng gap
3.4	3	12, 22, **11**, 64, 34, **25**, 90	Sắp xếp các phần tử cách nhau 3 đơn vị
4.1	1	**12, 22, 11, 64, 34, 25, 90**	Giảm gap đi một nửa là 3 / 2 bằng 1
4.2	1	**11, 12, 22, 25, 34, 64, 90**	Sắp xếp các phần tử cách nhau 1 đơn vị

2.5.3 Phân tích thuật toán

Độ phức tạp về thời gian: Phụ thuộc vào chuỗi khoảng cách được chọn.

- Best-case: Khi mảng gần như được sắp xếp, các phép so sánh và dịch chuyển phần tử giảm đáng kể. Với chuỗi khoảng cách tốt như chuỗi Knuth, chuỗi $(2^k - 1)$,.. thì độ phức tạp sẽ gần bằng $O(nlogn)$.
- Worst-case: Khi mảng có thứ tự sắp xếp ngược hoặc không có trật tự rõ ràng, với chuỗi không tối ưu như chuỗi chia đôi (n/2, n/4, n/8, ..., 1) thì độ phức tạp sẽ là $O(n^2)$. Còn với các chuỗi tối ưu như Knuth, Hibbard, Sedgewick thì sẽ là $O(n^{3/2})$) cho đến $O(nlog^2n)$.
- Average-case: Đối với chuỗi tối ưu như Sedgewick thì sẽ là $O(nlog^2n)$. Còn chuỗi không tối ưu thì sẽ là $O(n^2)$ [5]

Độ phức tạp về không gian: Shell Sort có sử dụng thêm mảng phụ để lưu khoảng cách nhưng yêu cầu ít bộ nhớ nên độ phức tạp là $O(1)$

Tính ổn định: Shell Sort là một thuật toán không ổn định (unstable). Vì với các gap lớn có thể làm thay đổi thứ tự của các phần tử bằng nhau

2.5.4 Cải tiến và biến thể

Có hai biến thể của Shell Sort là:

Shell-Merge Sort: Sử dụng Shell Sort để xử lý các khoảng cách lớn, sau đó sử dụng Merge Sort để sắp xếp cuối cùng.

Shell-Heap Sort: Tương tự như Shell-Merge Sort nhưng sử dụng Heap Sort cho việc sắp xếp cuối cùng.

2.6 Binary Insertion Sort

2.6.1 Ý tưởng

Binary Insertion Sort là một thuật toán sắp xếp tương tự như Insertion Sort, nhưng thay vì sử dụng tìm kiếm tuyến tính (Linear Search) để tìm vị trí chèn phần tử, chúng ta sử dụng tìm kiếm nhị phân (Binary Search).

Thuật toán được chia làm hai mảng con:

1. Mảng đã sắp xếp: Ban đầu chứa phần tử đầu tiên trong mảng.
2. Mảng chưa sắp xếp: Chứa tất cả các phần tử còn lại.

Mỗi phần tử trong mảng chưa sắp xếp sẽ được tìm kiếm vị trí phù hợp trong mảng đã sắp xếp bằng tìm kiếm nhị phân, sau đó chèn vào đúng vị trí [6]

2.6.2 Mô tả từng bước

Bước 1: Chọn phần tử đầu tiên là mảng con đã sắp xếp ban đầu.

Bước 2: Lặp qua các các phần tử từ vị trí thứ hai trở đi:

- Lưu giá trị hiện tại vào biến ***key***.
- Sử dụng Binary Search để tìm kiếm vị trí phù hợp trong mảng con đã sắp xếp.
- Dịch chuyển các phần tử lớn hơn ***key*** trong mảng con đã sắp xếp sang phải.
- Chèn ***key*** vào đúng vị trí đã tìm thấy.

Bước 3: Lặp lại bước 2 cho đến khi toàn bộ mảng được sắp xếp.

Ví dụ:

Bước	key	ARRAY	Mô tả
1	3	**5**, **3**, 7, 1	Khởi tạo mảng đã sắp xếp là 5
2.1	3	**5**, **3**, 7, 1	Chọn key bằng ***3***, tìm kiếm nhị phân ***key*** trong 5 xác định vị trí chèn là 1
2.2	3	**5**, **5**, 7, 1	Dịch chuyển các phần tử lớn hơn ***3*** trong mảng đã sắp xếp xang phải
2.3	3	**3**, **5**, 7, 1	Chèn ***3*** vào vị trí 1 đã tìm thấy
3.1	7	**3**, **5**, **7**, 1	Chọn key bằng ***7***, tìm kiếm nhị phân trong mảng đã sắp xếp, xác định vị trí chèn là 3
3.2	7	**3**, **5**, **7**, 1	Không có phần tử nào lớn hơn ***7***, thực hiện chèn ***7*** vào vị trí 3
4.1	1	**3**, **5**, **7**, **1**	Chọn key bằng ***1***, tìm kiếm nhị phân trong mảng đã sắp xếp, xác định vị trí chèn là 1
4.2	1	**3**, **3**, **5**, **7**	Dịch chuyển các phần tử lớn hơn ***1*** sang bên phải
4.3	1	**1**, **3**, **5**, **7**	Chèn ***1*** vào vị trí 1 đã tìm thấy

2.6.3 Phân tích thuật toán

Độ phức tạp về thời gian:

- Best-case: Mảng gần như đã sắp xếp, mỗi phần tử sẽ được tìm thấy ở cuối mảng và dịch chuyển số ít phần tử (hoặc không) nên độ phức tạp là $O(nlogn)$.
- Average-case: Mảng có đầu vào là các phần tử ngẫu nhiên, key sẽ được chèn vào giữa mảng và có sự dịch chuyển nên độ phức tạp là $O(n^2)$.
- Worst-case: Mảng được sắp xếp ngược, như vậy tất cả các phần tử đều phải dịch chuyển nên độ phức tạp là $O(n^2)$.

Độ phức tạp về không gian: Thuật toán hoạt động trên mảng gốc (in-place), không sử dụng thêm bộ nhớ ngoài nên độ phức tạp là O(1).
Tính ổn định: Vì các phần tử được chèn đúng vị trí mà không thay đổi thứ tự tương đối của các phần tử bằng nhau nên Binary Insertion Sort là thuật toán có tính ổn định (stable).

2.6.4 Cải tiến và biến thể

Kết hợp với các thuật toán như Quick Sort và Merge Sort hoặc áp dụng cấu trúc trên Danh sách liên kết sẽ giúp cho thuật toán Binary Insertion Sort cải thiện được đáng kể thời gian dịch chuyển các phần tử.

2.7 Counting Sort

Counting Sort là một thuật toán sắp xếp hoạt động hiệu quả đối với dữ liệu gồm các giá trị là số nguyên hoặc có thể ánh xạ thành số nguyên và phạm vi giá trị nhỏ hơn nhiều so với số lượng phần tử cần sắp xếp

2.7.1 Ý tưởng

Counting Sort là thuật toán sắp xếp không dựa trên so sánh. Ý tưởng cơ bản đằng sau là đếm tần số xuất hiện của từng phần tử riêng biệt trong mảng đầu vào và thông qua việc tính tổng cộng dồn, thuật toán xác định được vị trí chính xác của các phần tử.

2.7.2 Mô tả từng bước

Bước 1: Tìm MAX trong ARRAY.

Bước 2: Tạo mảng đếm (CountArray) với kích thước là MAX + 1 và khởi tạo tất cả giá trị là 0.

Bước 3: Duyệt qua ARRAY và tăng giá trị tại chỉ số tương ứng trong CountArray theo tần suất của từng phần tử.

Bước 4: Cộng dồn CountArray bằng cách giữ giá trị đầu tiên, các giá trị tiếp theo bằng giá trị phần tử trước đó cộng chính nó.

Bước 5: Duyệt ngược ARRAY, tạo một mảng để lưu trữ các phần tử đã sắp xếp OutputArray.

- Mỗi phần tử có vị trí tương ứng với chỉ số phần tử đó trong CountArray trừ đi 1.

- Đưa phần tử đó vào đúng vị trí trong OutputArray được CountArray lưu trữ.
- Mỗi phần tử sau khi đặt vào vị trí thì giảm giá trị trong CountArray đi một đơn vị.

Bước 6: Trả về mảng OutputArray. [7]

Ví dụ:

Bước 1: Tìm MAX trong ARRAY:

ARRAY	2	**5**	3	0	2	3
Index	0	1	2	3	4	5

Bước 2: Tạo mảng đếm với kích thước MAX + 1 bằng 5 + 1 và khởi tạo tất cả giá trị bằng 0.

CountArray	0	0	0	0	0	0
Index	0	1	2	3	4	5

Bước 3: Duyệt qua ARRAY và đếm tần suất xuất hiện của các giá trị trong ARRAY, lưu vào CountArray.

CountArray	1	0	2	2	0	1
Index	0	1	2	3	4	5

Bước 4: Cộng dồn CountArray.

CountArray	1	1	3	5	5	6
Index	0	1	2	3	4	5

Bước 5.1: Duyệt ngược ARRAY, tạo mảng OutputArray. Đưa từng phần tử trong ARRAY vào đúng vị trí được lưu trữ trong CountArray của OutputArray:

ARRAY	2	5	3	0	2	**3**
Index	0	1	2	3	4	5

Kiểm tra giá trị của 3 tương ứng với chỉ số trong mảng CountArray

CountArray	1	1	3	**5**	5	6
Index	0	1	2	**3**	4	5

Thấy giá trị của chỉ số 3 là 5 tức trong mảng sắp xếp thì số 3 ở vị trí 4 (5 trừ 1).

OutputArrray	0	0	0	0	**3**	0
Index	0	1	2	3	**4**	5

Chỉ số 3 đã được điền vào vị trí, thực hiện trừ đi một đơn vị

CountArray	1	1	3	**4**	5	6
Index	0	1	2	**3**	4	5

Bước 5.2: Kiểm tra giá trị của 2 tương ứng với chỉ số trong mảng CountArray

CountArray	1	1	**3**	4	5	6
Index	0	1	**2**	3	4	5

Thấy giá trị của chỉ số 2 là 3 tức trong mảng sắp xếp thì số 2 ở vị trí 2 (3 trừ 1).

OutputArrray	0	0	**2**	**3**	0	0
Index	0	1	**2**	**3**	4	5

Lặp lại các bước cho đến khi mảng ARRAY đã được duyệt hết. Mảng sau khi sắp xếp:

OutputArrray	**0**	**2**	**2**	**3**	**3**	**5**
Index	**0**	**1**	**2**	**3**	**4**	**5**

2.7.3 Phân tích thuật toán

- **Độ phức tạp về thời gian:** Tìm phần tử lớn nhất, đếm tần suất và sắp xếp các phần tử có độ phức tạp đều là O(n), còn cộng dồn mảng CountArray sẽ là O(MAX). Vậy độ phức tạp của thuật toán này là $O(n + MAX)$ với cả ba trường hợp Best-case, Average-case, Worst-case.
- **Độ phức tạp về không gian:** Thuật toán này yêu cầu hai mảng bổ sung (CountArray và OutputArray) nên độ phức tạp là O(n + MAX).
- **Tính ổn định:** Counting Sort là thuật toán ổn định (stable), vì ta điền trực tiếp vị trí tương đối của từng phần từ từ cuối về của ARRAY vào OutputArray.

2.7.4 Cải tiến và biến thể

- **Hỗ trợ giá trị âm:** Nếu mảng đầu vào chứa giá trị âm, cần ánh xạ các giá trị này thành số nguyên dương (bằng cách thêm giá trị bù, ví dụ offset = abs(min)
- **Kết hợp với Radix Sort:** Sắp xếp các chữ số của số nguyên theo từng chữ số (đơn vị, chục, trăm...). Radix Sort sẽ mở rộng khả năng của Counting Sort cho dãy số lớn hơn.
- **Lọc dữ liệu lớn:** Thay vì lưu trữ toàn bộ CountArray trong bộ nhớ, có thể tối ưu bằng cách chia nhỏ dãy giá trị đầu vào thành các nhóm và xử lý từng nhóm riêng biệt.

- **Tối ưu không gian:** Nếu chỉ cần biết thứ tự, không cần mảng kết quả, có thể tối ưu bộ nhớ bằng cách sử dụng chính mảng đầu vào để lưu kết quả.
- **Dùng với số thực:** Thoả điều kiện tổng số chữ số lớn nhất của phần nguyên và phần thập phân không quá 9 khi nhân mỗi số với 10 mũ (số chữ số phần thập phân lớn nhất), rồi ánh xạ để có thể ra được Counting Sort ban đầu.

2.8 Radix Sort

2.8.1 Ý tưởng

Radix Sort là một thuật toán sắp xếp không dựa trên phép so sánh, hoạt động bằng cách xử lý từng chữ số (hoặc ký tự) của các phần tử dựa trên cơ số xác định. Thuật toán đảm bảo tính ổn định, tức là thứ tự ban đầu của các phần tử có cùng giá trị sẽ được giữ nguyên sau khi sắp xếp. [8]

2.8.2 Mô tả từng bước

Bước 1: Xác định số lượng chữ số lớn nhất trong mảng (dựa trên số có nhiều chữ số nhất).

Bước 2: Bắt đầu từ chữ số ít quan trọng nhất (chữ số ngoài cùng bên phải), sắp xếp các số dựa trên chữ số đó bằng cách sử dụng một thuật toán sắp xếp ổn định (chẳng hạn như Counting Sort hoặc Bucket Sort).

Bước 3: Lặp lại bước 2 cho từng vị trí chữ số tiếp theo, di chuyển dần về phía chữ số quan trọng nhất (chữ số ngoài cùng bên trái).

Bước 4: Sau khi duyệt qua tất cả các chữ số, mảng sẽ được sắp xếp hoàn chỉnh. [9]

Ví dụ:

Bước	ARRAY	Mô tả
1	12, 8, 7, 43, 170, 90, **802**	Số lượng chữ số lớn nhất trong mảng là 3 (MAX là 802).
2	17**0**, 9**0**, 1**2**, 80**2**, 4**3**, **7**, **8**	Sắp xếp mảng theo hàng đơn vị
3.1	80**2**, **0**7, **0**8, 1**2**, 17**0**, 9**0**	Sắp xếp mảng theo hàng chục
3.2	**0**07, **0**08, **0**12, **0**90, **1**70, **8**02	Sắp xếp mảng theo hàng trăm
4	**7, 8, 12, 90, 170, 802**	Sau khi duyệt qua tất cả các chữ số, mảng đã sắp xếp

2.8.3 Phân tích thuật toán

Độ phức tạp về thời gian: Cả Average-case, Best-case, Worst-case đều là $O(n)$ bởi vì số lượng chữ số sẽ không đáng kể so với số phần tử.

Độ phức tạp về không gian: Radix Sort sẽ cần bộ nhớ thêm để lưu các chữ số, độ phức tạp về không gian sẽ là O(n + b) với b là cơ số của phần tử.

Tính ổn định: Radix Sort là một thuật toán sắp xếp ổn định (stable). Bởi vì tính ổn định làm nên thuật toán Radix Sort.

2.8.4 Cải tiến và biến thể

- Một biến thể cũng như là cải tiến của Radix Sort là **Most significant digit (MSD)**. Thay vì sắp xếp từ phải qua trái thì thuật toán này sắp xếp từ trái sang bên phải và với mỗi bước sẽ chia ra các "bucket"khác nhau. Độ phức tạp về thời gian ta có Best-case, Average-case và Worst-case đều là $O(n)$. Độ phức tạp không gian là: O(n + mb) với b là cơ số của phần tử.
- Radix sort có thể cải tiến giá trị âm bằng cách chia mảng thành 2, một mảng chứa giá trị âm và một mảng chứa giá trị không âm, thực hiện radix sort trên mảng không âm. Mảng chứa giá trị âm có thể dùng abs để radix sort tương tự rồi ánh xạ về mảng cũ. Sau đó ghép mảng âm và mảng không âm lại.
- Radix sort cũng có thể cải tiến với số thực với điều kiện: tổng số chữ số lớn nhất của phần nguyên và phần thập phân không qua số chữ số lớn nhất của kiểu dữ liệu, ví dụ long long có lớn nhất 19 chữ số. Nhân mỗi số với 10 mũ (số chữ số phần thập phân lớn nhất), rồi ánh xạ để có thể về được Radix Sort ban đầu.

2.9 Merge Sort

Merge Sort rất phù hợp cho các tình huống yêu cầu sắp xếp ổn định và hiệu quả với bộ dữ liệu lớn, đặc biệt khi dữ liệu không thể được xử lý hoàn toàn trong bộ nhớ, hoặc khi dữ liệu cần được chia thành các phần để xử lý song song.

2.9.1 Ý tưởng

Merge Sort là một thuật toán sắp xếp sử dụng phương pháp chia để trị (divide and conquer). Ý tưởng chính của thuật này là chia mảng đầu vào thành các mảng con nhỏ hơn, sắp xếp các mảng con này, và sau đó gộp chúng lại để tạo thành mảng đã sắp xếp. Quá trình này được lặp đi lặp lại cho đến khi toàn bộ mảng được sắp xếp.

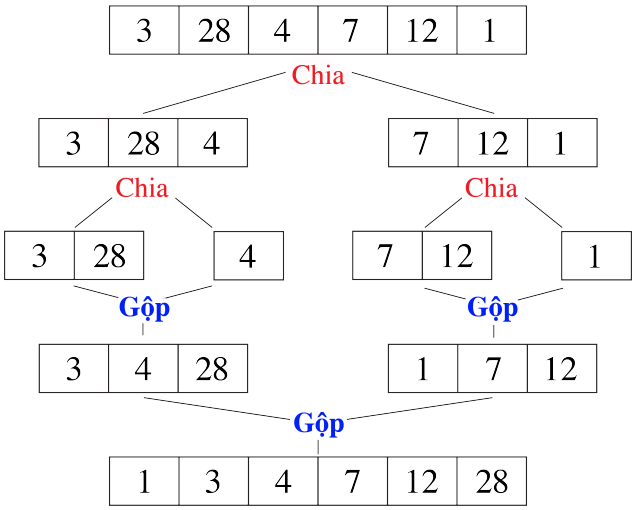
2.9.2 Mô tả từng bước

Bước 1: Chia (Divide) - Đệ quy chia mảng thành hai phần cho đến khi các phần chỉ còn một phần tử hoặc không có phần tử nào. Mảng đơn phần tử là mảng đã sắp xếp.

Bước 2: Trị (Conquer) - Khi mảng đã được chia thành các mảng con, mỗi mảng con được sắp xếp (thực hiện lại bước chia cho đến khi mảng con có một phần tử).

Bước 3: Gộp (Merge) - Sau khi các mảng con đã được sắp xếp, ta bắt đầu gộp các mảng này lại. Quá trình gộp diễn ra bằng cách so sánh từng phần tử của các mảng con và đưa chúng vào mảng kết quả theo thứ tự tăng dần. [10]

Ví dụ:



2.9.3 Phân tích thuật toán

- **Độ phức tạp về thời gian:** Trong tất cả các trường hợp thì Merge Sort đều chia mảng ra để so sánh rồi gộp, vì thế độ phức tạp là như nhau $O(n \log n)$.
- **Độ phức tạp về không gian:** Merge Sort có sử dụng thêm mảng phụ để lưu trữ các mảng con đã chia, vì thế độ phức tạp là $O(n)$.
- **Tính ổn định:** Hàm gộp phần tử của Merge Sort sẽ truy lần từng phần tử của hai mảng con và sắp xếp chúng. Vì thế, các phần từ giống nhau sẽ bảo toàn tính tương đối thứ tự nên Merge Sort có tính ổn định (stable).

2.9.4 Cải tiến và biến thể

- Thuật toán đang trình bày ở trên được gọi là Top-down Merge Sort tức là chia từ mảng lớn về thành các mảng bé. Tuy nhiên, thuật toán này yêu cầu bộ nhớ bổ sung cho các mảng phụ. Ta có thể cải tiến bằng cách sử dụng Bottom-up Merge Sort tức là chia mảng ban đầu thành các phần tử đơn lẻ, sau đó gộp dần bằng cách sử dụng vòng lặp cho đến khi toàn bộ mảng được sắp xếp, vì thế ta sẽ sử dụng vòng lặp thay cho đệ quy và không cần mảng phụ. [11]
- Thay vì sử dụng mảng, Merge Sort cũng có thể được thực hiện trên các danh sách liên kết. Việc gộp các danh sách liên kết có thể tiết kiệm bộ nhớ vì không cần phải sao chép các phần tử vào mảng phụ.
- Kết hợp với Insertion Sort có thể là việc tối ưu quy trình gộp khi mảng con gần như đã sắp xếp.

2.10 Quick Sort

2.10.1 Ý tưởng

Quick Sort là một thuật toán sắp xếp theo phương pháp chia để trị (Divide-and-Conquer). Ý tưởng chính của thuật toán là chọn một phần tử làm pivot, sau đó phân chia mảng thành hai phần dựa trên giá trị của pivot:

– Các phần tử nhỏ hơn pivot nằm bên trái.

– Các phần tử lớn hơn hoặc bằng pivot nằm bên phải.

Quá trình này được lặp lại đệ quy cho đến khi mỗi phân đoạn chỉ còn một phần tử hoặc rỗng.

2.10.2 Mô tả từng bước

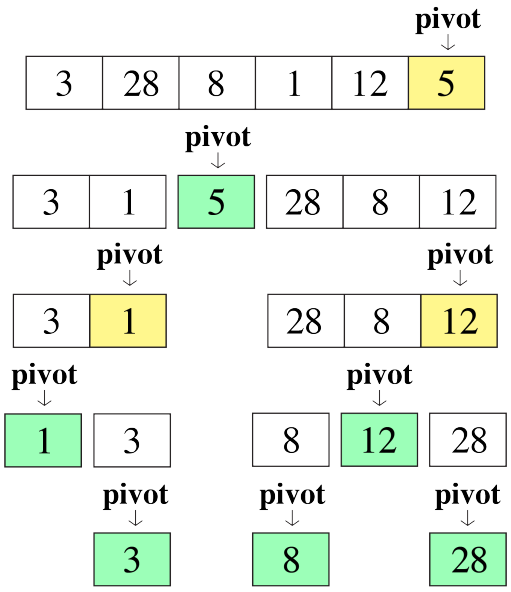
Bước 1: Chọn pivot. Có nhiều cách để chọn pivot, dưới đây là 2 cách cơ bản:

- Chọn phần tử đầu tiên hoặc cuối cùng của mảng đang xét để làm pivot.
- Đối với mỗi mảng đang xét, ta chọn ba phần tử ở đầu mảng, giữa mảng và cuối mảng. Tìm trung vị của ba phần tử này, lấy phần tử đó làm pivot.

Bước 2: Phân hoạch mảng sao cho các phần tử lớn hơn hoặc bằng pivot nằm bên phải pivot và nhỏ hơn pivot thì nằm bên trái pivot. Sau đó đưa pivot về đúng vị trí của nó trong mảng đã sắp xếp *(Ta có thể tham khảo 3 thuật toán phân hoạch nổi tiếng là phân hoạch Naive, Lomuto, Hoare).*

Bước 3: Gọi đệ quy Quick Sort cho mảng bên trái và mảng bên phải của Pivot cho đến khi mảng con chỉ còn một phần tử hoặc rỗng.

Ví dụ: Cho mảng ARRAY = {3, 28, 8, 1, 12, 5}, sắp xếp tăng dần với cách chọn pivot là phần tử cuối cùng



Mảng sau khi sắp xếp: ARRAY = {1, 3, 5, 8, 12, 28} (phần màu xanh).

2.10.3 Phân tích thuật toán

Độ phức tạp về thời gian: Phụ thuộc vào các chọn pivot của từng mảng con

– Best-case: Xảy ra khi pivot chia mảng thành hai nửa cân bằng, Độ phức tạp sẽ là $O(n \log n)$

– Average-case: Xảy ra khi pivot chia mảng thành hai phần tương đối đều nhau, độ phức tạp là $O(n \log n)$

- Worst-case: Xảy ra khi pivot luôn là MIN hoặc MAX (ví dụ như mảng đã sắp xếp), khi này tất cả các phần tử sẽ bị dồn sang một bên dẫn đến mảng bị phân không đồng đều, độ phức tạp sẽ là $O(n^2)$

Độ phức tạp về không gian:

- Average-case: Do sử dụng đệ quy nên sẽ tốn bộ nhớ cho ngăn xếp, độ phức tạp sẽ là $O(n)$.
- Worst-case: Xảy ra tương tự như Worst-case về thời gian, nếu như phân hoạch không đều thì đệ quy sẽ đưa toàn bộ mảng vào ngăn xếp dẫn đến độ phức tạp $O(n)$.

Tính ổn định: Quick Sort **không ổn định** (unstable), vì khi sắp xếp các phần tử có giá trị bằng nhau có thể bị thay đổi thứ tự tương đối do hoán đổi vị trí ở hàm phân hoạch.

2.10.4 Cải tiến và biến thể

- Quick Sort là một thuật toán có hiệu quả cao trong nhiều trường hợp, vậy nên để cải tiến thuật toán này ta có thể xây dựng các thuật toán chọn pivot sao cho phân hoạch đều ở các mảng phụ.
- Khi mảng có kích thước lớn, ta sử dụng Quick Sort cho đến các mảng con có kích thước nhỏ (thường là nhỏ hơn 10), ta kết hợp thuật toán Insertion Sort để sắp xếp các mảng con. Thuật toán này còn được gọi là Hybrid Algorithms [12].
- Độ phức tạp về không gian là một vấn đề của Quick Sort, vì thế ta có thể tối ưu bằng cách cài đặt đệ quy đuôi cho thuật toán, khi này ngăn xếp sẽ được tối ưu hoá.

2.11 Heap Sort

Heap Sort là một thuật toán sắp xếp dựa trên so sánh, hoạt động dựa trên cấu trúc Binary Heap. Đây có thể được coi là một cải tiến của Selection Sort, trong đó:

- Phần tử lớn nhất (hoặc nhỏ nhất) được tìm thấy và hoán đổi với phần tử cuối cùng (hoặc đầu tiên).
- Quá trình này lặp lại cho đến khi mảng được sắp xếp.

2.11.1 Ý tưởng

1. Đầu tiên, xây dựng ARRAY về thành một MAX-HEAP (hoặc MIN-HEAP).
   MAX-HEAP có tính chất sau: Cho mảng có chỉ số bắt đầu từ 0, Tất cả các phần tử thuộc nửa mảng đầu tiên lớn hơn hoặc bằng các phần tử ở vị trí thứ $2i + 1$ và $2i + 2$ (với i là phần tử đang xét).
2. Việc xây dựng ARRAY về thành một MAX-HEAP (hoặc MIN-HEAP) sẽ đưa MAX (hoặc MIN) về đầu mảng. Như vậy, việc ta cần làm là hoán đổi hai phần tử ở vị trí đầu mảng và cuối mảng cho nhau.
3. Giảm kích thước Heap và sử dụng Heapify để xây dựng lại MAX-HEAP (hoặc MIN-HEAP).
4. Lặp lại việc hoán đổi và xây dựng Heap cho đến khi Heap chỉ còn 1 phần tử. Như vậy mảng sẽ được sắp xếp. [13]

2.11.2 Mô tả từng bước

Bước 1: Khởi tạo i bằng $(n/2-1)$ và duyệt từ i về đầu mảng. Xây dựng MAX-HEAP (Heapifiy) tại phần tử (Node) i:

- Lưu tạm giá trị i vào biến ***largest***.
- Nếu Node i nhỏ hơn Node con trái i $(2i+1)$ thì gắn largest bằng Node con trái i.
- Nếu largest nhỏ hơn Node con phải i $(2i+2)$ thì gắn largest bằng Node con phải i.
- Nếu largest khác i chứng tỏ Node i ban đầu không phải Node lớn nhất thì hoán đổi giá trị của chỉ số i cho giá trị của chỉ số largest. Sau đó thực hiện gọi đệ quy hàm xây dựng MAX-HEAP tại vị trí largest.

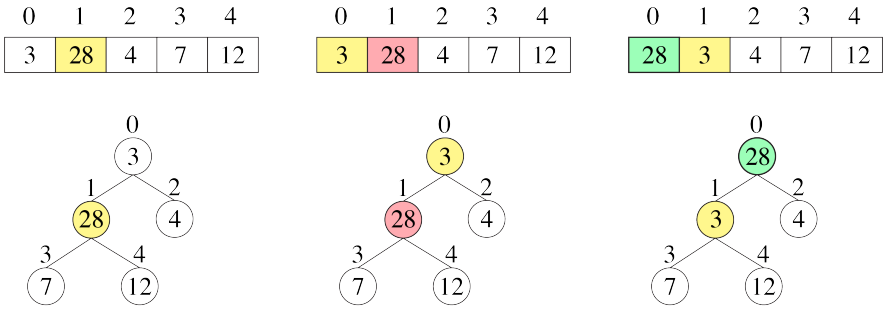
Bước 2: Hoán đổi Node gốc (MAX) với phần tử cuối cùng trong Heap (Đối với mảng thì có thể hiểu đơn giản là hoán đổi phần tử đầu tiên và phần tử cuối cùng sau khi xây dựng được mảng MAX-HEAP).

Bước 3: Xoá Node cuối cùng ra khỏi Heap và gọi lại Heapify để duy trì Heap mỗi khi thay đổi.

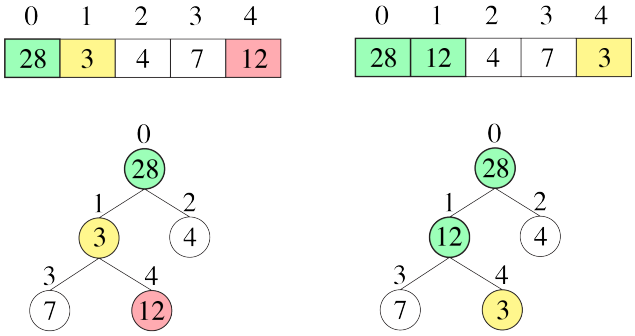
Bước 4: Lặp lại bước 2 và 3 cho đến khi Heap chỉ còn một phần tử. Khi này mảng sẽ được sắp xếp.

Ví dụ: Cho mảng ARRAY = {3, 28, 4, 7, 12}. Hình bên dưới minh hoạ các bước bằng cây và bằng mảng (Màu **vàng** là Node đang xét, màu **đỏ** là Node đang ở sai vị trí, màu **xanh** chỉ Node đã ở đúng vị trí):

Bước 1: Xây dựng MAX-HEAP từ phần tử i bằng 1 (n/2 - 1 bằng 5/2 - 1). Ta thấy 28 đang ở đúng vị trị. Tiếp tục với vị trí i - 1 bằng 0, ta thấy 28 và 3 ở sai vị trí, thực hiện hoán đổi.

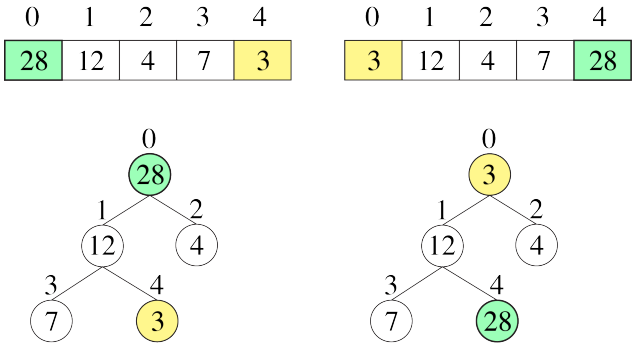


Xét Node mới hoán đổi thấy 12 ở sai vị trí, hoán đổi 12 với 3.

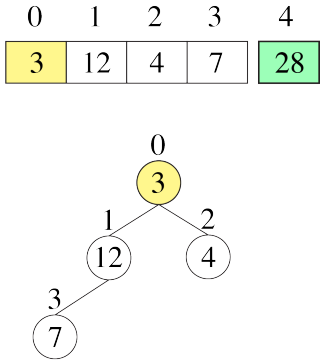


Như vậy đã xây dựng thành công MAX-HEAP.

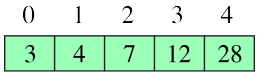
Bước 2: Thực hiện hoán đổi Node gốc và Node cuối cùng.



Bước 3: Xoá Node cuối cùng và gọi Heapify để xây dựng lại Heap mới.



Bước 4: Lặp lại các bước cho đến khi Heap còn 1 Node. Khi này mảng sẽ được sắp xếp.



2.11.3 Phân tích thuật toán

- **Độ phức tạp về thời gian:** Xây dựng MAX-HEAP (hoặc MIN-HEAP) sẽ có độ phức tạp là $O(n)$ và quá trình Heapify sẽ tốn $O(\log n)$ vậy nên Heap Sort sẽ có độ phức tạp với cả ba trường hợp Best-case, Average-case, Worst-case về thời gian là $O(n \log n)$.
- **Độ phức tạp về không gian:** Nếu như cài đặt Heap Sort bằng đệ quy, thuật toán này sẽ có độ phức tạp về không gian là $O(\log n)$.
- **Tính ổn định:** Heap Sort là một thuật toán không ổn định (unstable). Vì khi heapify và lấy phần tử ra khỏi cây có thể làm thay đổi thứ tự ban đầu của các phần tử bằng nhau.

2.11.4 Cải tiến và biến thể

Heap Sort nếu như cài đặt bằng đệ quy thì thuật toán này sẽ có độ phức tạp về không gian là $O(\log n)$, tuy nhiên ta có thể giảm xuống O(1) nếu như dùng Heapify dạng lặp (iterative).

2.12 Flash Sort

Flash Sort là thuật toán sắp xếp có hiệu suất cao, được thiết kế để xử lý mảng lớn và có phân phối đều. Thuật toán kết hợp việc phân vùng dữ liệu và sắp xếp cục bộ nhằm đạt tốc độ sắp xếp nhanh gần như tuyến tính trong trường hợp lý tưởng. [14]

2.12.1 Ý tưởng

1. **Phân vùng (Class Mapping)**: Thuật toán chia mảng thành k lớp (buckets) dựa trên giá trị của các phần tử. Mỗi lớp sẽ chứa các phần tử trong một phạm vi giá trị cụ thể.
2. **Hoán đổi (Swapping)**: Sử dụng chỉ số lớp của mỗi phần tử để di chuyển chúng đến vị trí đúng (Flash Placement). Các phần tử sẽ được hoán đổi cho đến khi chúng nằm đúng vị trí trong mảng.
3. **Sắp xếp cục bộ (Local Sorting)**: Sau khi phân bố các phần tử vào các lớp, thuật toán sẽ áp dụng một thuật toán sắp xếp đơn giản (như Insertion Sort) cho các lớp để hoàn thành quá trình sắp xếp.

2.12.2 Mô tả từng bước

Bước 1: Tìm giá trị nhỏ nhất và lớn nhất: Tìm giá trị nhỏ nhất và lớn nhất trong mảng để xác định phạm vi giá trị của các lớp.

Bước 2: Tính toán số lớp (buckets): Tính số lớp cần thiết dựa trên kích thước mảng và phạm vi giá trị giữa phần tử nhỏ nhất và lớn nhất. Số lớp thường được tính bằng công thức:

$$k = \lfloor \alpha \cdot n \rfloor$$

trong đó α là một hệ số và n là số phần tử trong mảng và k lớn hơn bằng 2 (nếu k nhỏ hơn 2 thì cho k bằng 2).

Bước 3: Phân bổ các phần tử vào các lớp: Mỗi phần tử trong mảng sẽ được gán vào một lớp dựa trên giá trị của nó. Chỉ số lớp sẽ được tính dựa trên giá trị phần tử và phạm vi của các lớp.

Bước 4: Chuyển đổi mảng lớp thành mảng vị trí: Mỗi lớp sẽ có một vị trí cuối cùng được tính toán dựa trên số phần tử trong mỗi lớp. Việc chuyển đổi này giúp xác định nơi mỗi phần tử sẽ được đặt.

Bước 5: Đặt các phần tử vào lớp đúng của chúng: Hoán đổi các phần tử trong mảng bằng cách sử dụng chỉ số lớp của chúng, cho đến khi tất cả phần tử được đặt đúng lớp.

Bước 6: Sử dụng Insertion Sort: Sau khi hoán đổi, mỗi lớp sẽ được sắp xếp lại bằng một thuật toán sắp xếp đơn giản như Insertion Sort.

Ví dụ: Cho ARRAY = {45, 70, 50, 32, 90, 65, 85, 15, 75, 60}, n = 10 phần tử. Chọn $\alpha = 0.3 \rightarrow k = 0.3 \cdot 10 = 3$

Bước 1: Xác định cực trị

$$\min = 15, \quad \max = 90.$$

Bước 2: Phân chia lớp

Công thức xác định lớp:

$$\text{class}(x) = \left\lfloor k \cdot \frac{x - \min}{\max - \min + 1} \right\rfloor$$

Kết quả:

$$\text{class}(45) = \left\lfloor 3 \cdot \frac{45 - 15}{90 - 15 + 1} \right\rfloor = 1, \quad \text{class}(70) = 2, \quad \text{class}(50) = 1,$$
$$\text{class}(32) = 0, \quad \text{class}(90) = 2, \quad \text{class}(65) = 2,$$
$$\text{class}(85) = 2, \quad \text{class}(15) = 0, \quad \text{class}(75) = 2, \quad \text{class}(60) = 1.$$

Bước 3: Phân phối dữ liệu vào các lớp

$$\text{Lớp 0: } \{32, 15\}, \quad \text{Lớp 1: } \{45, 50, 60\}, \quad \text{Lớp 2: } \{70, 90, 65, 85, 75\}.$$

Bước 4: Sắp xếp trong từng lớp

- Lớp 0: Sắp xếp $\{32, 15\} \to \{15, 32\}$,
- Lớp 1: Sắp xếp $\{45, 50, 60\} \to \{45, 50, 60\}$,
- Lớp 2: Sắp xếp $\{70, 90, 65, 85, 75\} \to \{65, 70, 75, 85, 90\}$.

Bước 5: Kết hợp các lớp Kết quả sau khi gộp:

$$\text{ARRAY} = \{15, 32, 45, 50, 60, 65, 70, 75, 85, 90\}.$$

2.12.3 Phân tích thuật toán

Độ phức tạp về thời gian:

- Best-case: Đối với mảng có phân bố đều, độ phức tạp có thể lên đến O(n).
- Average-case: Với số lớp chọn hợp lý, thì độ phức tạp sẽ là $O(n)$
- Worst-case: Xảy ra Khi phần tử bị phân phối không đều hoặc các lớp không cân bằng. Độ phức tạp sẽ là $O(n^2)$

Độ phức tạp về không gian: Flash Sort yêu cầu một mảng phụ để lưu số lượng phần tử mỗi lớp, nên độ phức tạp về không gian là $O(n)$

Tính ổn định: Đây là thuật toán không ổn định (unstable) vì khi thêm phần tử vào buckets thì các phần tử không giữ được thứ tự ban đầu và tại bước hoán đổi, thứ tự tương đối không được bảo toàn.

2.12.4 Cải tiến và biến thể

Một số cải tiến và biến thể của thuật toán Flash Sort bao gồm:

- **Tối ưu hóa số lớp m:** Thông thường, $m = 0.2n$ (20% số lượng phần tử).
- **Tối ưu hóa hoán đổi chỗ:** Sử dụng chiến lược dịch chuyển (tránh hoán đổi lặp đi lặp lại).
- **Kết hợp với các thuật toán khác:** Áp dụng Quick Sort hoặc Merge Sort khi số lượng phần tử trong lớp vượt một ngưỡng nhất định.

3 Thực nghiệm và nhận xét

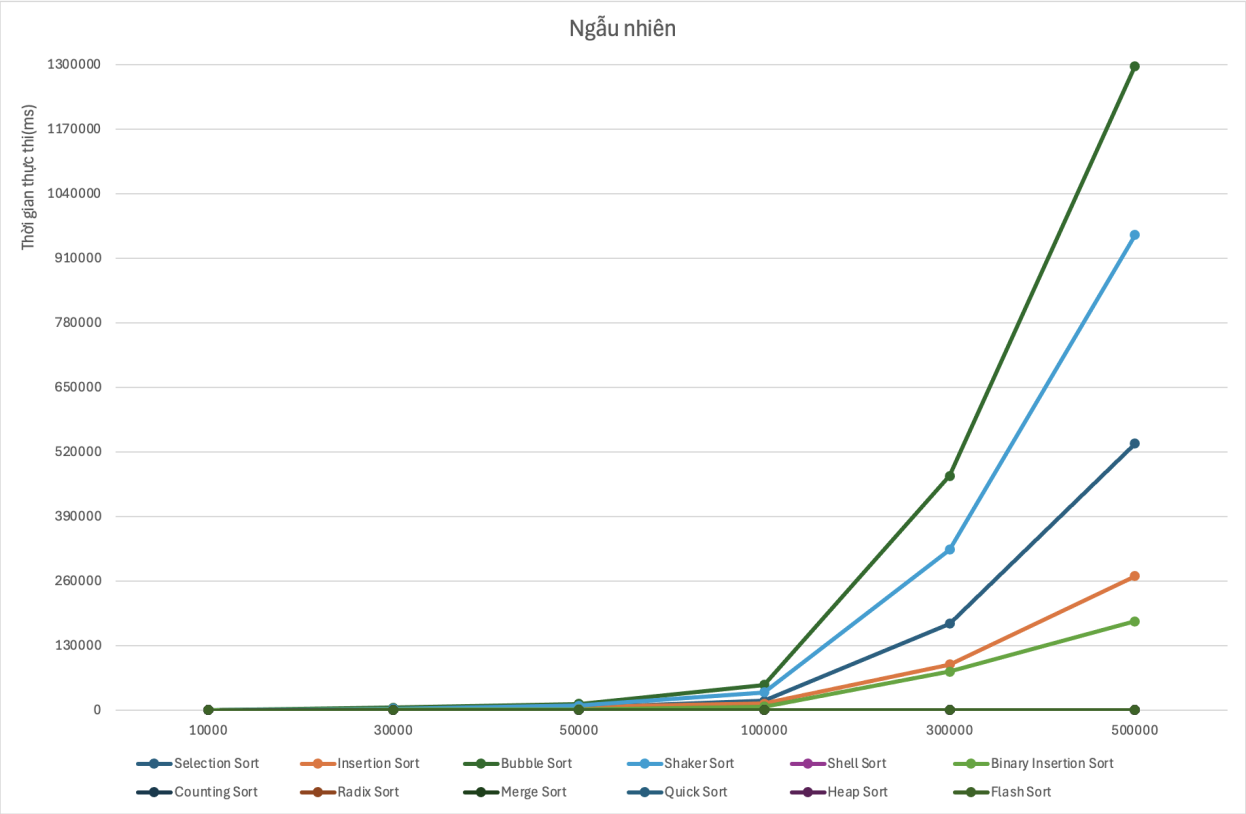
Quy ước các từ viết tắt: Số phép so sánh là CP (phép); Thời gian thực thi là R.T (ms)

3.1 Dữ liệu ngẫu nhiên (Random data)

3.1.1 Bảng thực nghiệm

Kiểu sắp xếp dữ liệu: **Randomize**						
Kích thước dữ liệu	**10000**		**30000**		**50000**	
Thống kê kết quả	**CP (phép)**	**R.T (ms)**	**CP (phép)**	**R.T (ms)**	**CP (phép)**	**R.T (ms)**
Selection Sort	100009999	164,802	900029999	1844,8	2500049999	4626,49
Insertion Sort	25079861	90,7693	225288048	880,8	625131701	6756,58
Bubble Sort	100003617	511,128	899938897	4758,32	2499944765	12810,2
Shaker Sort	75264350	347,072	678719748	3003,31	1874224728	9313,13
Shell Sort	261408	6,2785	931239	8,9835	1823101	15,3853
Binary Insertion Sort	25370629	60,0225	226044201	764,983	628541322	1720,49
Counting Sort	50002	0	150002	0,9842	232771	1,0159
Radix Sort	140056	1,0103	510070	1,994	850070	3,9995
Merge Sort	583516	3,9852	1937649	12,4691	3382923	22,727
Quick Sort	351281	1,9958	1121302	4,2005	1969889	7,0141
Heap Sort	637706	5,3444	2150420	8,2225	3772830	15,3182
Flash Sort	91354	1,0087	283126	1,9989	448231	3,0157
Kích thước dữ liệu	**100000**		**300000**		**500000**	
Thống kê kết quả	**CP (phép)**	**R.T (ms)**	**CP (phép)**	**R.T (ms)**	**CP (phép)**	**R.T (ms)**
Selection Sort	10000099999	18740,8	90000299999	173979	250000499999	536490
Insertion Sort	2507172265	12841,4	22519411990	91458,9	62502690341	269691
Bubble Sort	10000061617	50901,5	90000521601	471714	250000998065	1296380
Shaker Sort	7472775744	36000	67655880120	322959	187267032378	956775
Shell Sort	4119128	36,772	14853648	125,921	29122438	267,817
Binary Insertion Sort	2507353260	6744,49	22532736225	77142,8	62497402407	179491
Counting Sort	432771	2,0002	1232771	6,3651	2032771	8,5799
Radix Sort	1700070	7,433	5100070	24,4109	8500070	41,438
Merge Sort	7166103	43,5238	23381952	154,25	40383370	312,088
Quick Sort	4237958	16,6312	15126536	51,2483	28552335	91,4255
Heap Sort	8043773	32,5868	26489326	115,502	45969444	186,145
Flash Sort	862027	4,2764	2775445	13,438	4909923	28,6984

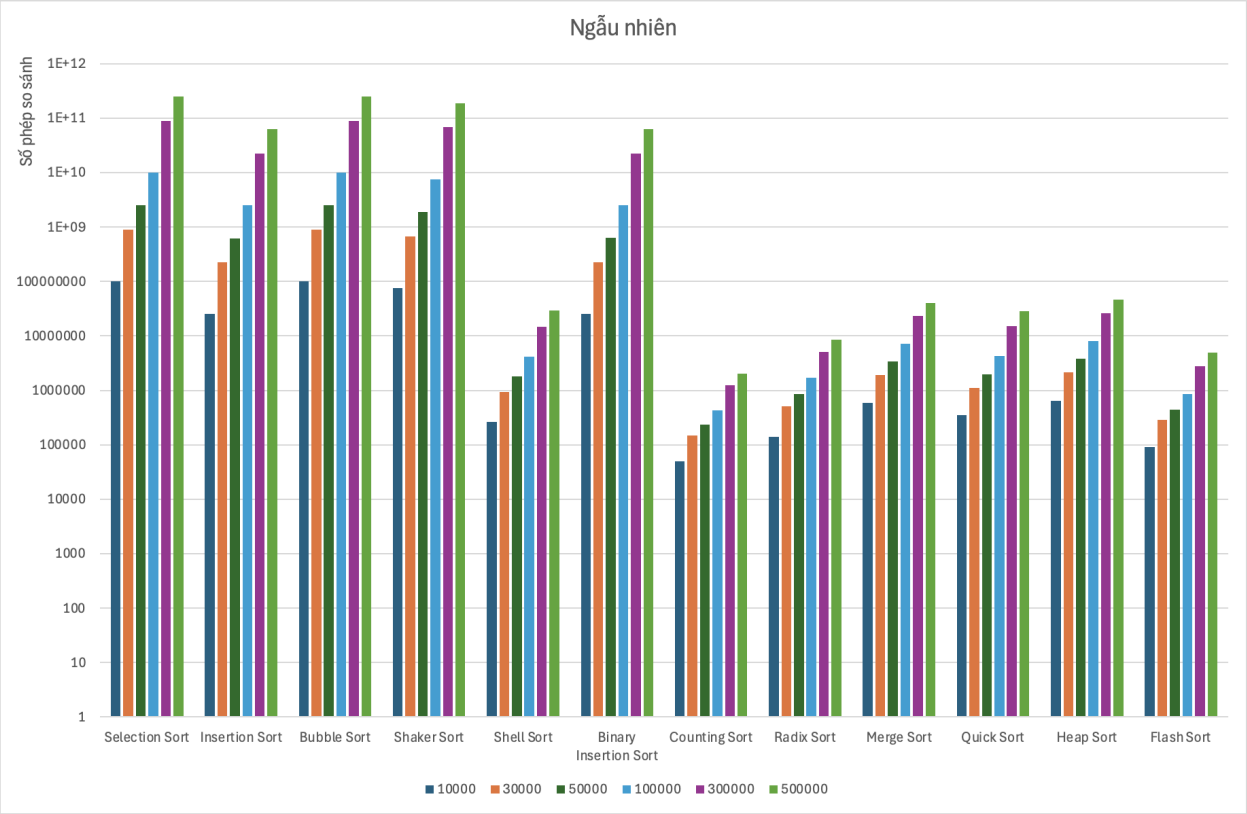
3.1.2 Biểu đồ đường thể hiện thời gian thực thi



Nhận xét:

- Thời gian chạy lớn nhất của Bubble Sort là 1296380 ms khi số phần tử là 500000, tiếp đó là Shaker Sort với thời gian 956775 ms khi số phần tử là 500000. Thời gian chạy thấp nhất là của Counting Sort với 0 ms khi số phần tử là 10000.
- Các thuật toán có độ phức tạp thời gian $O(n^2)$ hiển thị rõ ràng các đường riêng biệt trên biểu đồ (như Bubble Sort, Shaker Sort, Selection Sort, Insertion Sort, Binary Insertion Sort theo thứ tự từ cao xuống thấp). Sự khác biệt giữa các thuật toán này bắt đầu rõ rệt kể từ khi số phần tử dữ liệu đạt 50000 trở lên.
- Các thuật toán có độ phức tạp thời gian $O(n \log n)$ hay $O(n)$ như Shell Sort, Counting Sort, Radix Sort, Merge Sort, Quick Sort, Heap Sort, Flash Sort đều gần như gắn với trục hoành (do thời gian chạy của chúng rất nhỏ so với thời gian chạy lớn nhất). Ngay cả với dữ liệu có số phần tử lớn nhất, các thuật toán này vẫn giữ mức thời gian thấp và không tạo ra sự khác biệt lớn trên biểu đồ.

3.1.3 Biểu đồ cột thể hiện số phép so sánh



Nhận xét:

- Số phép so sánh lớn nhất của Bubble Sort là 250000998065 khi số phần tử là 500000, tiếp đó là Selection Sort với tổng số phép so sánh là 250000499999 khi số phần tử là 500000. Số phép so sánh ít nhất là của Counting Sort với 50002 phép so sánh khi số phần tử là 10000.
- Biểu đồ sử dụng bước nhảy theo cấp số nhân 10, do đó, tỉ lệ các cột được thể hiện theo log cơ số 10. Điều này có nghĩa là khi một cột vượt qua mốc độ chia, giá trị của nó gấp 10 lần mốc độ chia trước đó.
- Có sự khác biệt rõ rệt giữa các thuật toán có độ phức tạp thời gian $O(n^2)$ và các thuật toán có độ phức tạp $O(n \log n)$ hay $O(n)$. Các thuật toán $O(n^2)$ như Insertion Sort, Binary Insertion Sort, Shaker Sort, Selection Sort, và Bubble Sort đã vượt qua mốc 1×10^{11} với dữ liệu có số phần tử là 500000. Ngay cả với dữ liệu nhỏ nhất (số phần tử 10000), giá trị của các thuật toán $O(n^2)$ vẫn vượt qua mốc 1×10^7 và thậm chí chạm mốc 1×10^8 (như Selection Sort và Bubble Sort).
- Trong khi đó, các thuật toán có độ phức tạp thời gian $O(n \log n)$ hay $O(n)$ như Shell Sort, Counting Sort, Radix Sort, Merge Sort, Quick Sort, Heap Sort, và Flash Sort, ngay cả với dữ liệu lớn nhất (500000 phần tử), chỉ đạt giá trị trong khoảng 1×10^7 đến 0.75×10^8 (thấp hơn cả giá trị nhỏ nhất của các thuật toán $O(n^2)$). Các thuật toán $O(n \log n)$ có chiều cao gần như đồng đều trên biểu đồ, trong khi

các thuật toán $O(n)$ thấp hơn rõ rệt và khoảng cách lớn nhất giữa các thuật toán $O(n)$ là một nửa mốc độ chia.

– Với dữ liệu có số phần tử từ 30000 trở đi, các thuật toán $O(n^2)$ thực hiện số phép so sánh ít nhất gấp 10 lần số phép so sánh của các thuật toán $O(n \log n)$ hay $O(n)$.

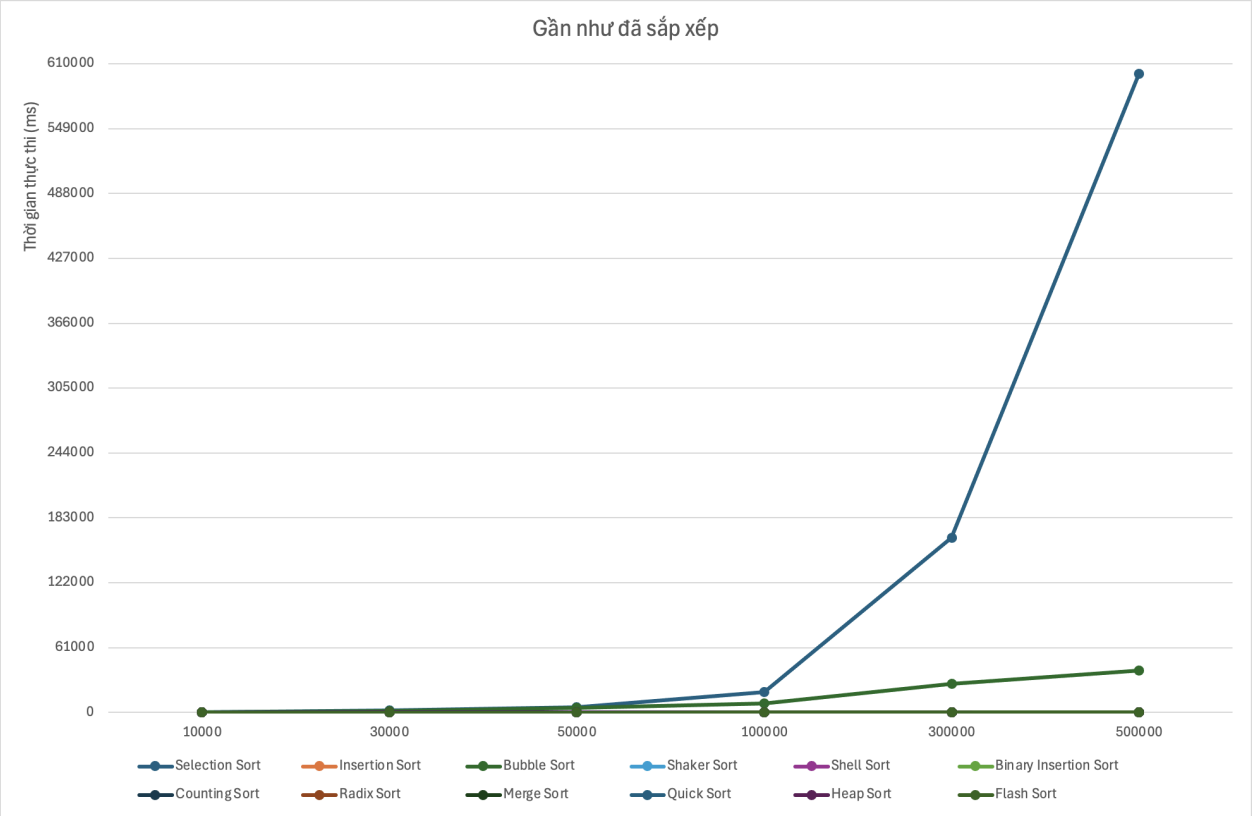
Nhận xét 2 biểu đồ: Có mối liên hệ giữa số phép so sánh và thời gian chạy giữa các thuật toán. Thời gian chạy sẽ tỉ lệ thuận với số phép so sánh của các thuật toán. Ví dụ như các thuật toán có độ phức tạp thời gian n^2 sẽ cao hẳn so với các thuật toán có độ phức tạp thời gian $n \log n$ hay n thì sẽ xuất hiện rõ ràng trong biểu đồ đường về thời gian thực thi. Tương tự với các thuật toán gắn với trục hoành ở biểu đồ đường của thời gian thực thi (là các thuật toán có độ phức tạp thời gian $n \log$ hay n) thì sẽ là những cột thấp so với những cột cao nhất của biểu đồ cột của số phép so sánh.

3.2 Dữ liệu gần như sắp xếp (Nearly sorted data)

3.2.1 Bảng thực nghiệm

Kiểu sắp xếp dữ liệu: **Nearly sorted**						
Kích thước dữ liệu	**10000**		**30000**		**50000**	
Thống kê kết quả	**CP (phép)**	**R.T (ms)**	**CP (phép)**	**R.T (ms)**	**CP (phép)**	**R.T (ms)**
Selection Sort	100009999	167,216	900029999	1778,29	2500049999	4932,34
Insertion Sort	66437	0,9976	266141	1,9998	324233	3,9976
Bubble Sort	94939485	259,492	797301232	1519,87	1585702880	4118,55
Shaker Sort	279818	1,0002	959760	2,295	1599760	3,0093
Shell Sort	137360	0	473689	3,0033	797392	5,1737
Binary Insertion Sort	426801	1,0002	1463815	2,983	2554271	5,508
Counting Sort	50003	0	150003	0	250003	1,0005
Radix Sort	140056	1,0128	510070	1,998	850070	2,9982
Merge Sort	509875	3,0038	1660231	9,4214	2831617	15,89
Quick Sort	4029036	2,9983	23426705	18,3802	84174668	61,9324
Heap Sort	670017	1,9916	2236608	6,345	3924829	11,4873
Flash Sort	114473	1,003	343475	1,0098	572472	0,9902
Kích thước dữ liệu	**100000**		**300000**		**500000**	
Thống kê kết quả	**CP (phép)**	**R.T (ms)**	**CP (phép)**	**R.T (ms)**	**CP (phép)**	**R.T (ms)**
Selection Sort	10000099999	19418,7	90000299999	164251	250000499999	600839
Insertion Sort	478403	2,001	763381	3,9825	1182909	6,5834
Bubble Sort	4104878040	8734,94	10313520480	27061,4	26951657040	39423,1
Shaker Sort	2799818	4,8957	9599760	22,8965	15999760	29,1638
Shell Sort	1597392	10,2651	5179457	24,2457	8592572	55,2035
Binary Insertion Sort	4916313	7,623	16006849	28,7114	27723062	52,3693
Counting Sort	500003	1,0242	1500003	5,9855	2500003	9,1481
Radix Sort	1700070	8,0162	6000084	31,4236	10000084	52,6741
Merge Sort	5854449	32,3432	18717262	148,112	32119705	187,822
Quick Sort	59910396	47,0206	33545238	36,1932	29332987	41,7886
Heap Sort	8365013	26,6972	27413250	82,9548	47405049	138,518
Flash Sort	1144974	3,3729	3434974	11,214	5724974	18,0917

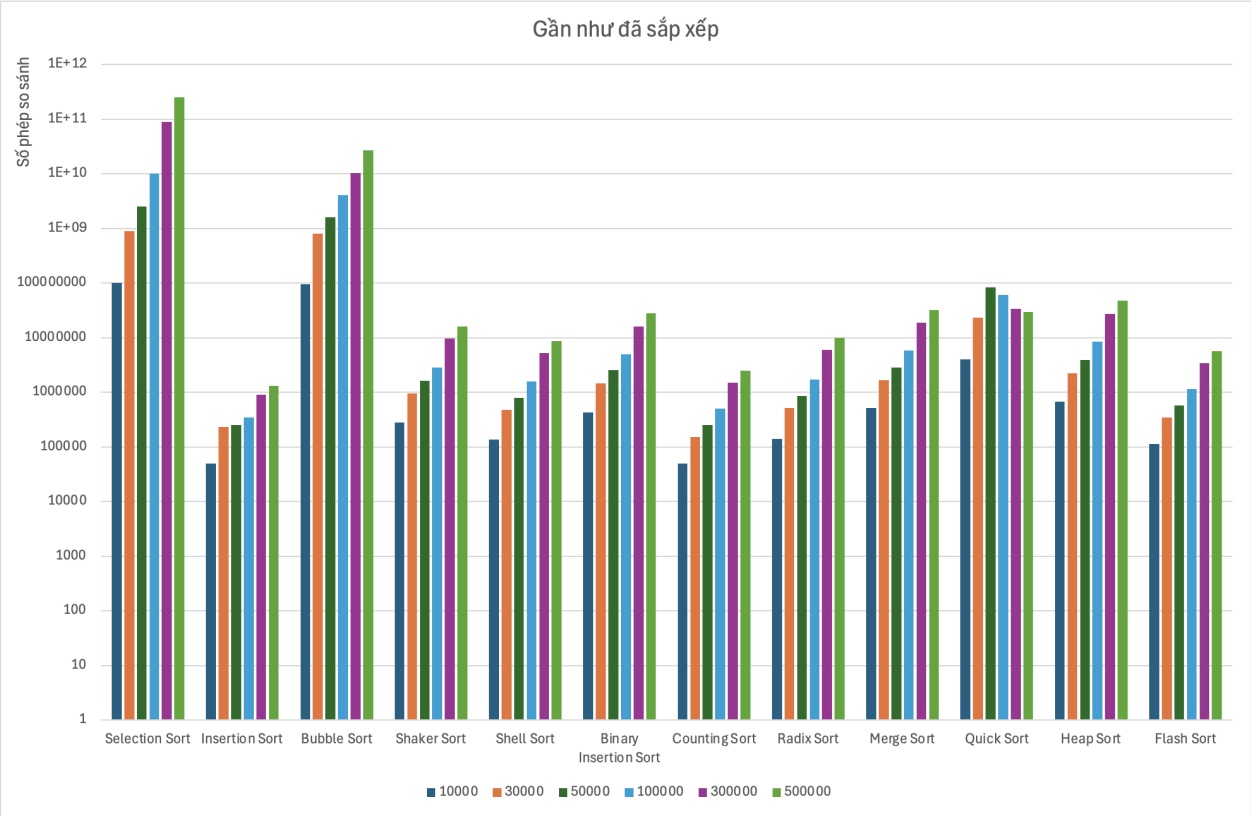
3.2.2 Biểu đồ đường thể hiện thời gian thực thi



Nhận xét:

- Thời gian chạy lớn nhất của Selection Sort là 600839 ms khi số phần tử là 500000, tiếp đó cũng là Selection Sort với 164251 ms khi số phần tử là 300000. Thời gian chạy thấp nhất được ghi nhận ở các thuật toán Shell Sort và Counting Sort với 0 ms khi số phần tử là 10000, cũng như Counting Sort khi số phần tử là 30000, và Insertion Sort khi số phần tử là 10000, 30000, 50000, và 100000.
- Thuật toán Selection Sort và Bubble Sort (đều có độ phức tạp thời gian $O(n^2)$) hiển thị rõ ràng các đường riêng biệt trên biểu đồ, đặc biệt kể từ dữ liệu với số phần tử từ 50000 trở đi.
- Các thuật toán còn lại, bao gồm cả những thuật toán có độ phức tạp thời gian $O(n^2)$ (như Insertion Sort, Shaker Sort, Binary Insertion Sort trong trường hợp Average-case) và các thuật toán có độ phức tạp $O(n \log n)$ hay $O(n)$, đã được tối ưu trong trường hợp Best-case. Do đó, thời gian chạy trong trường hợp mảng gần như đã được sắp xếp nhỏ hơn đáng kể so với thời gian chạy lớn nhất, và tất cả đều gần như gắn với trục hoành, ngay cả với dữ liệu có số phần tử lớn nhất.

3.2.3 Biểu đồ cột thể hiện số phép so sánh



Nhận xét:

- Số phép so sánh lớn nhất của Selection Sort với tổng số là 250000499999 khi số phần tử là 500000, tiếp đó là Selection Sort với tổng số là 90000299999 khi số phần tử là 300000. Số phép so sánh ít nhất của Insertion Sort với tổng số là 49709 khi số phần tử là 10000.
- Đây là biểu đồ thể hiện bước nhảy theo cấp số nhân 10 nên tỉ lệ các cột được thể hiện theo log cơ số 10, nghĩa là khi cột này vượt qua cột mốc độ chia của giá trị thì sẽ gấp 10 lần cột một độ chia của giá trị trước đó.
- Có sự khác biệt giữa các thuật toán có độ phức tạp thời gian tại trường hợp Average-case hoặc gần đạt tới trường hợp Best-case $O(n^2)$ với $O(n \log n)$ hay $O(n)$. Các thuật toán có độ phức tạp $O(n^2)$ vượt qua mốc 1×10^{10} như Bubble Sort hoặc thậm chí vượt quá mốc 1×10^{11} như Selection Sort với dữ liệu có số phần tử 500000. Thậm chí với dữ liệu có số phần tử nhỏ nhất như 10000, giá trị của Selection Sort đã vượt qua 1×10^8 và Bubble Sort đã tiệm cận giá trị này.
- Còn với các thuật toán có độ phức tạp thời gian trong trường hợp Average-case hoặc gần đạt tới trường hợp Best-case $O(n \log n)$ hay $O(n)$ như Shell Sort, Counting Sort, Radix Sort, Merge Sort, Quick Sort, Heap Sort, Flash Sort, Insertion Sort, Shaker Sort, Binary Insertion Sort, giá trị lớn nhất có thể đạt

được với dữ liệu có số phần tử là 500000 chỉ khoảng 0.5×10^8 (còn thấp hơn cả giá trị của các thuật toán $O(n^2)$ với số phần tử nhỏ nhất). Độ cao lớn nhất của các thuật toán này là của Quick Sort khi số phần tử là 50000, xảy ra trong trường hợp đa số đều chọn pivot là giá trị cực trị. Trong khi các thuật toán $O(n \log n)$ (Merge Sort, Quick Sort, Heap Sort, Binary Insertion Sort) có chiều cao gần như đều nhau, các thuật toán có độ phức tạp thời gian $O(n)$ trong trường hợp Average-case hoặc gần đạt tới Best-case đều thấp hơn hẳn. Còn với dữ liệu có số phần tử từ 30000 trở đi của các thuật toán $O(n^2)$, số phép so sánh đều ít nhất gấp 10 lần số phép so sánh của các thuật toán $O(n \log n)$ hay $O(n)$.

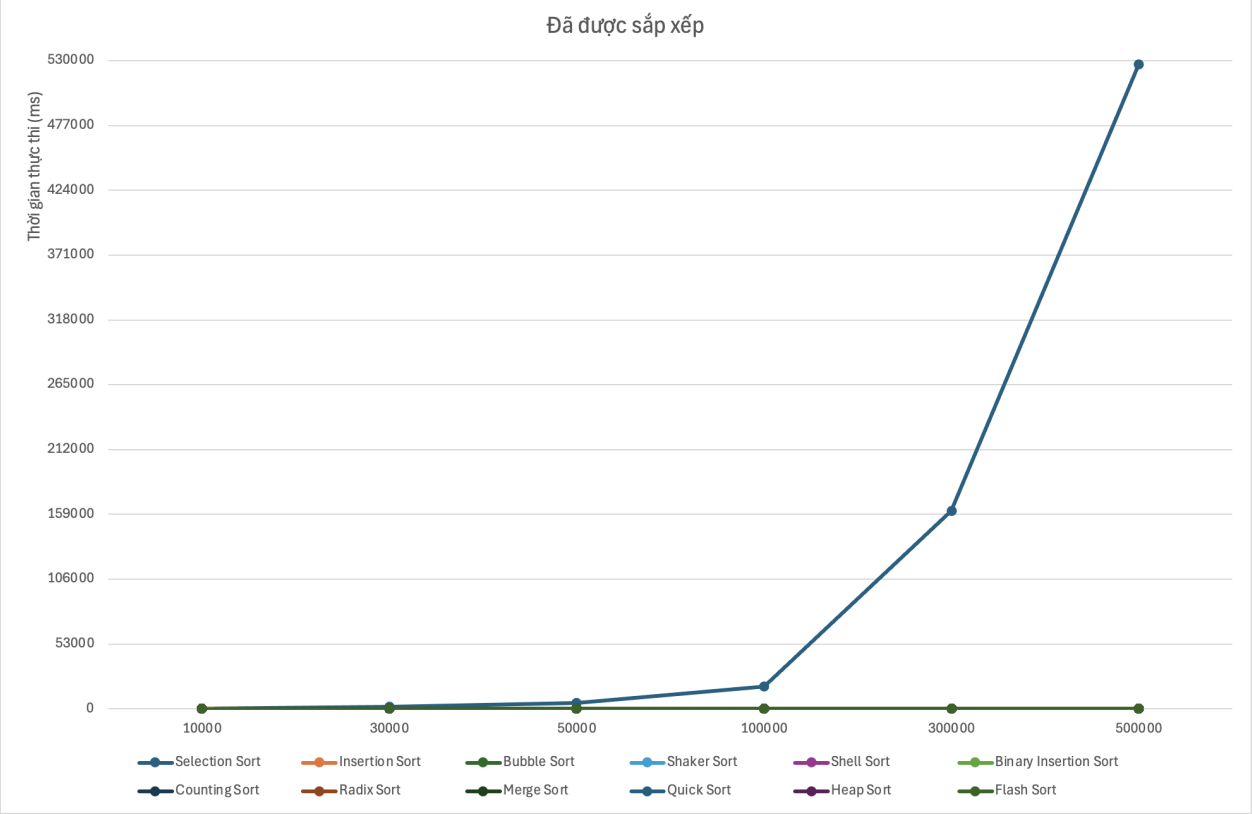
Nhận xét 2 biểu đồ: Có mối liên hệ giữa số phép so sánh và thời gian chạy giữa các thuật toán. Thời gian chạy sẽ tỉ lệ thuận với số phép so sánh của các thuật toán. Ví dụ như các thuật toán có độ phức tạp thời gian trong trường hợp average case hay gần đạt Best-case là n^2 cao hơn hẳn các thuật toán còn lại (kể từ dữ liệu có số phần tử là 50000 thì gấp 100 lần các thuật toán còn lại) trong biểu đồ cột của số phép so sánh thì sẽ xuất hiện rõ ràng trong biểu đồ đường của thời gian chạy kể từ dữ liệu có số phần tử là 50000 (Selection Sort và Bubble Sort), tương tự với các thuật toán gắn với trục hoành ở biểu đồ đường của thời gian thực thi (là các thuật toán có độ phức tạp thời gian $n \log n$ hay n) thì cũng là một cột thấp so với 2 cột được hiển thị rõ tương ứng với đường trong biểu đồ đường của biểu đồ cột của số phép so sánh.

3.3 Dữ liệu đã sắp xếp (Sorted data)

3.3.1 Bảng thực nghiệm

Kiểu sắp xếp dữ liệu: **Sorted**						
Kích thước dữ liệu	**10000**		**30000**		**50000**	
Thống kê kết quả	**CP (phép)**	**R.T (ms)**	**CP (phép)**	**R.T (ms)**	**CP (phép)**	**R.T (ms)**
Selection Sort	100009999	219,325	900029999	1706,11	2500049999	4963,35
Insertion Sort	19999	0	59999	1,0044	99999	1,0993
Bubble Sort	20001	0	60001	0	100001	1,0145
Shaker Sort	39998	0	119998	0	199998	0,9934
Shell Sort	120005	0	390007	2,109	700006	3,1614
Binary Insertion Sort	380851	1,0001	1281699	2,5067	2253395	3,0034
Counting Sort	50003	1,0337	150003	1,1694	250003	1,0001
Radix Sort	140056	0	510070	2,0167	850070	3,4211
Merge Sort	475242	3,3256	1559914	9,3186	2722826	15,8312
Quick Sort	260878	1,0148	870028	1,9998	1550030	3,0005
Heap Sort	670329	1,0226	2236648	6,2589	3925351	12,792
Flash Sort	114499	0	343499	2,0017	572499	2,016
Kích thước dữ liệu	**100000**		**300000**		**500000**	
Thống kê kết quả	**CP (phép)**	**R.T (ms)**	**CP (phép)**	**R.T (ms)**	**CP (phép)**	**R.T (ms)**
Selection Sort	10000099999	18367,7	90000299999	161609	250000499999	526991
Insertion Sort	199999	0	599999	1,9992	999999	5,0126
Bubble Sort	200001	1,0204	600001	1,0001	1000001	2,2107
Shaker Sort	399998	0,5028	1199998	2,5176	1999998	2,0023
Shell Sort	1500006	7,7776	5100008	19,501	8500007	34,1844
Binary Insertion Sort	4806787	11,5945	15827139	33,0848	27427139	51,8008
Counting Sort	500003	2,5722	1500003	5,4446	2500003	9,2419
Radix Sort	1700070	7,4234	6000084	28,3772	10000084	47,9408
Merge Sort	5745658	32,9868	18645946	133,4	32017850	177,032
Quick Sort	3300032	5,9974	10727176	18,7898	18500036	33,3705
Heap Sort	8365080	25,2112	27413230	84,433	47404886	142,901
Flash Sort	1144999	4,0001	3434999	11,4055	5724999	19,0332

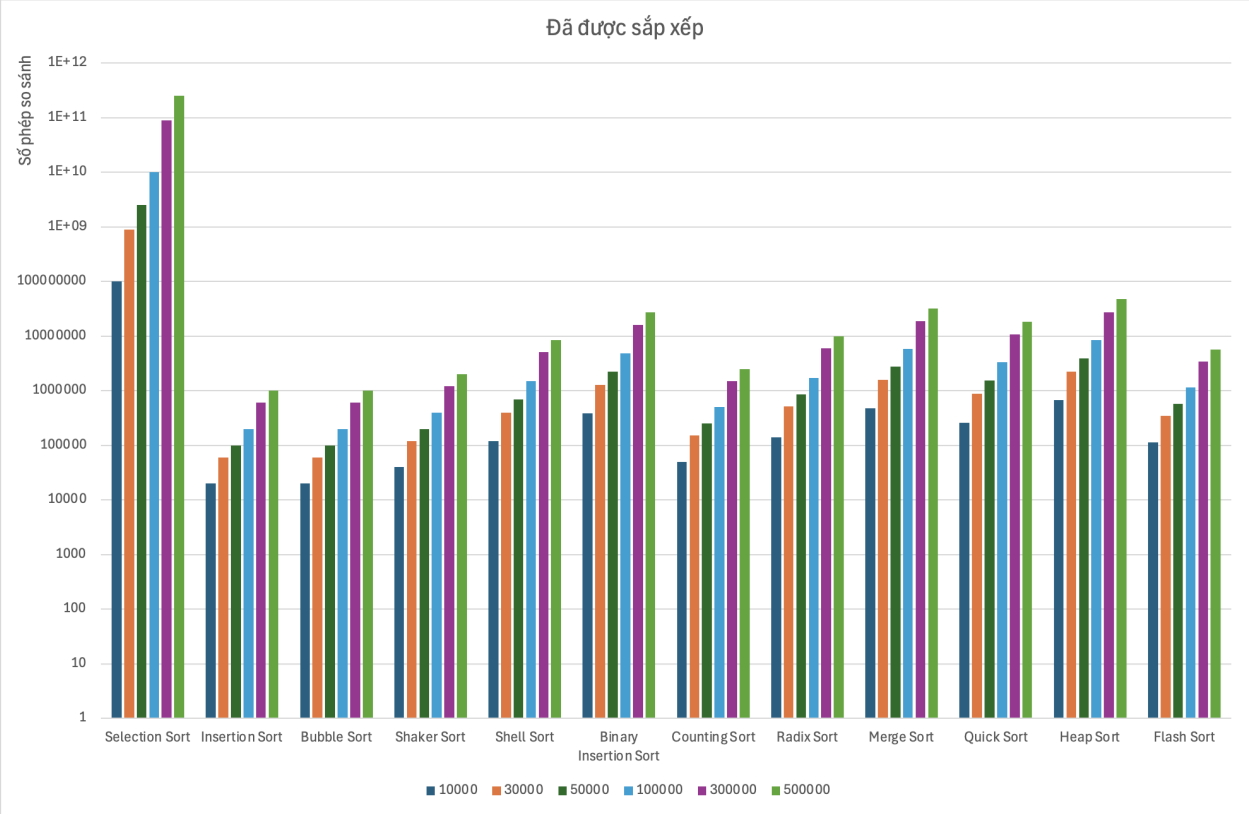
3.3.2 Biểu đồ đường thể hiện thời gian thực thi



Nhận xét:

- Thời gian chạy lớn nhất của Selection Sort với tổng số là 526991 ms khi số phần tử là 500000, tiếp đó là Selection Sort với thời gian là 161609 ms khi số phần tử là 300000. Thời gian chạy thấp nhất của Insertion Sort, Bubble Sort, Shaker Sort, Shell Sort, Radix Sort, Flash Sort với thời gian chạy là 0 ms khi số phần tử là 10000; Insertion Sort, Bubble Sort, Shaker Sort, Counting Sort khi số phần tử là 30000; và Insertion Sort, Shaker Sort, Counting Sort khi số phần tử là 50000, 100000, 300000; cũng như Insertion Sort khi số phần tử là 500000.
- Với Best-case, thuật toán Selection Sort có độ phức tạp thời gian $O(n^2)$ hiển thị rõ ràng các đường riêng biệt trên biểu đồ, rõ ràng nhất là kể từ dữ liệu với số phần tử từ 50000 trở đi.
- Các thuật toán còn lại, dù có độ phức tạp thời gian trung bình $O(n^2)$ như Bubble Sort (đã được tối ưu khi mảng đã được sắp xếp), Insertion Sort, Shaker Sort, Binary Insertion Sort trong Average-case hay các thuật toán có độ phức tạp thời gian $O(n \log n)$ hoặc $O(n)$, đều được tối ưu trong trường hợp Best-case. Vì vậy, thời gian chạy trong trường hợp mảng gần như đã được sắp xếp nhỏ hơn rất nhiều so với thời gian chạy lớn nhất, nên biểu đồ của chúng gần như gắn với trục hoành, ngay cả với dữ liệu có số phần tử lớn nhất.

3.3.3 Biểu đồ cột thể hiện số phép so sánh



Nhận xét:

- Số phép so sánh lớn nhất của Selection Sort với tổng số là 250000499999 khi số phần tử là 500000, tiếp đó là Selection Sort với tổng số là 90000299999 khi số phần tử là 300000. Số phép so sánh ít nhất của Insertion Sort với tổng số là 19999 khi số phần tử là 10000.
- Đây là biểu đồ thể hiện bước nhảy theo cấp số nhân 10, nên tỉ lệ các cột được thể hiện theo log cơ số 10. Nghĩa là khi cột này vượt qua cột mốc độ chia của giá trị, thì sẽ gấp 10 lần cột một độ chia của giá trị trước đó.
- Có sự khác biệt giữa các thuật toán có độ phức tạp thời gian tại trường hợp Best-case $O(n^2)$ với $O(n \log n)$ hay $O(n)$. Thuật toán có độ phức tạp $O(n^2)$ như Selection Sort vượt qua mốc $1E + 11$ với dữ liệu có số phần tử 500000. Thậm chí với dữ liệu có số phần tử nhỏ nhất như 10000, thì giá trị của Selection Sort đã vượt hơn mốc $1E + 08$ một ít.
- Các thuật toán có độ phức tạp thời gian trung bình $O(n^2)$ như Bubble Sort, Insertion Sort, Shaker Sort, Binary Insertion Sort được tối ưu trong Best-case thành $O(n \log n)$ hay $O(n)$. Cùng với đó, các thuật toán có độ phức tạp trung bình và Best-case $O(n \log n)$ hoặc $O(n)$ như Shell Sort, Counting Sort, Radix Sort, Merge Sort, Quick Sort, Heap Sort, Flash Sort có giá trị lớn nhất đạt được với dữ liệu có

số phần tử 500000 chỉ khoảng $0.5E+08$, còn thấp hơn giá trị với dữ liệu nhỏ nhất của các thuật toán $O(n^2)$.

– Độ cao lớn nhất trong nhóm này thuộc về Heap Sort khi số phần tử là 500000. Các thuật toán có độ phức tạp $O(n \log n)$ (Shell Sort, Binary Insertion Sort, Merge Sort, Quick Sort, Heap Sort) có chiều cao gần như đều nhau. Trong khi đó, với dữ liệu từ 30000 trở đi, Selection Sort với độ phức tạp $O(n^2)$ có số phép so sánh ít nhất gấp 10 lần số phép so sánh của các thuật toán có độ phức tạp $O(n \log n)$ hoặc $O(n)$ trong Best-case.

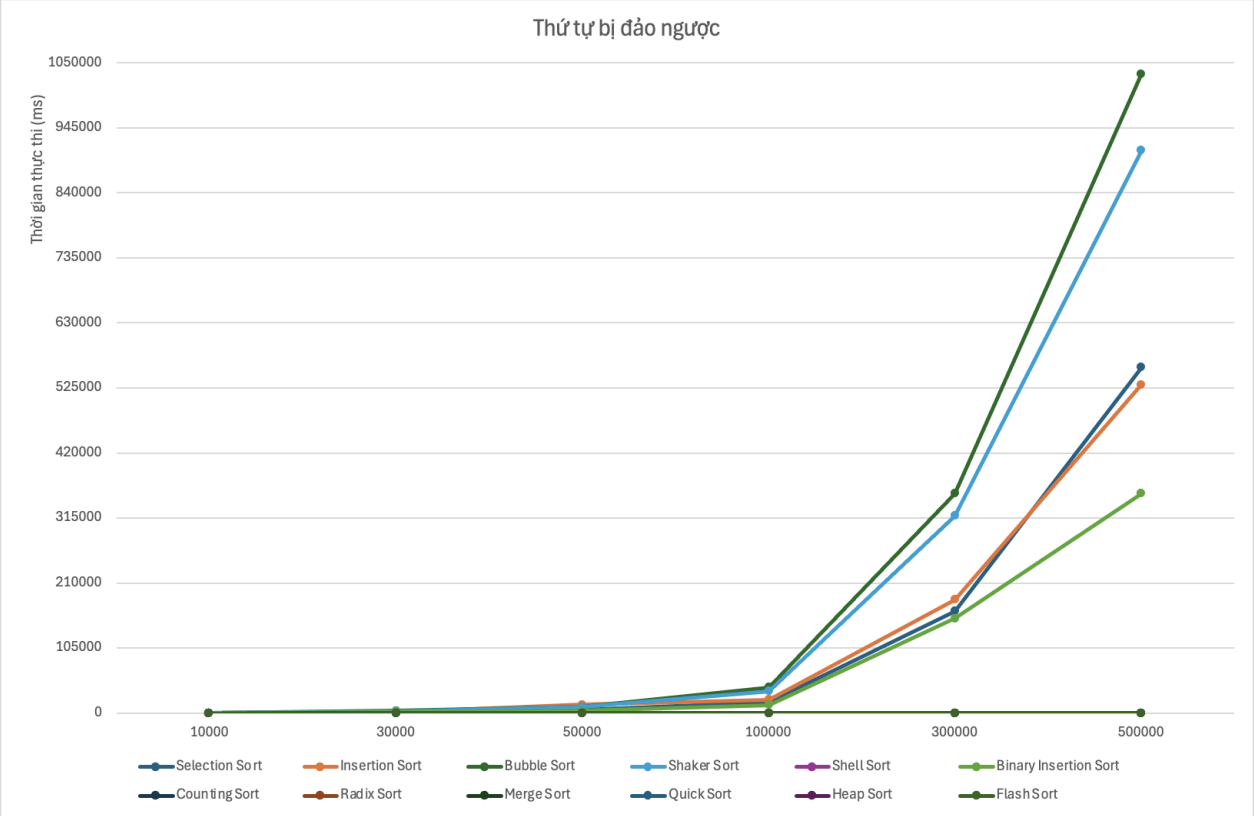
Nhận xét 2 biểu đồ: Có mối liên hệ giữa số phép so sánh và thời gian chạy giữa các thuật toán. Thời gian chạy sẽ tỉ lệ thuận với số phép so sánh của các thuật toán. Ví dụ như các thuật toán có độ phức tạp thời gian trong trường hợp Best-case là n^2 là Selection Sort cao hơn hẳn các thuật toán còn lại (kể từ dữ liệu có số phần tử là 50000 thì gấp 100 lần các thuật toán còn lại) trong biểu đồ cột của số phép so sánh thì sẽ xuất hiện rõ ràng trong biểu đồ đường của thời gian chạy kể từ dữ liệu có số phần tử là 50000 (Selection Sort), tương tự với các thuật toán gắn với trục hoành ở biểu đồ đường của thời gian thực thi (là các thuật toán có độ phức tạp thời gian $n \log n$ hay n) thì cũng là những cột thấp so với cột được hiển thị rõ tương ứng với đường trong biểu đồ đường của biểu đồ cột của số phép so sánh.

3.4 Dữ liệu sắp xếp ngược (Reverse sorted data)

3.4.1 Bảng thực nghiệm

Kiểu sắp xếp dữ liệu: **Reverse sorted**						
Kích thước dữ liệu	**10000**		**30000**		**50000**	
Thống kê kết quả	**CP (phép)**	**R.T (ms)**	**CP (phép)**	**R.T (ms)**	**CP (phép)**	**R.T (ms)**
Selection Sort	100009999	175.158	900029999	1736.04	2500049999	5031.97
Insertion Sort	50014999	180.449	450044999	1812.04	1250074999	13362.9
Bubble Sort	100019998	429.06	900059998	3812	2500099998	10102.1
Shaker Sort	100010001	374.058	900030001	3194.64	2500050001	10471.1
Shell Sort	172578	1.0026	567016	4.0083	1047305	5.1376
Binary Insertion Sort	50405806	226.984	451356651	1288.34	1252378344	3566.32
Counting Sort	50003	0	150003	0	250003	0
Radix Sort	140056	0.9966	510070	2.0033	850070	3.4516
Merge Sort	476441	3.0168	1573465	10.2751	2733945	16.4986
Quick Sort	311235	1.2239	1056698	2.1534	1836828	3.2957
Heap Sort	606771	2.0008	2063324	6.213	3612724	13.5495
Flash Sort	89253	0	267753	1.0023	446253	1.1997
Kích thước dữ liệu	**100000**		**300000**		**500000**	
Thống kê kết quả	**CP (phép)**	**R.T (ms)**	**CP (phép)**	**R.T (ms)**	**CP (phép)**	**R.T (ms)**
Selection Sort	10000099999	17782.2	90000299999	164790	250000499999	557788
Insertion Sort	5000149999	20956.7	45000449999	183139	125000749999	530140
Bubble Sort	10000199998	40948.9	90000599998	354256	250000999998	1032440
Shaker Sort	10000100001	35483.3	90000300001	318432	250000500001	908751
Shell Sort	2244585	11.086	7300919	49.5835	12428778	84.1319
Binary Insertion Sort	5005056733	13070.8	45016577079	152605	125028677079	354126
Counting Sort	500003	2.0267	1500003	5.0166	2500003	8.1436
Radix Sort	1700070	7.6009	6000084	29.0252	10000084	46.5231
Merge Sort	5767897	35.8477	18708313	119.242	32336409	211.839
Quick Sort	3923584	6.3524	13038709	21.3157	22726506	37.8353
Heap Sort	7718943	24.2535	25569379	77.9357	44483348	141.845
Flash Sort	892503	2.9981	2677503	11.2352	4462503	16.351

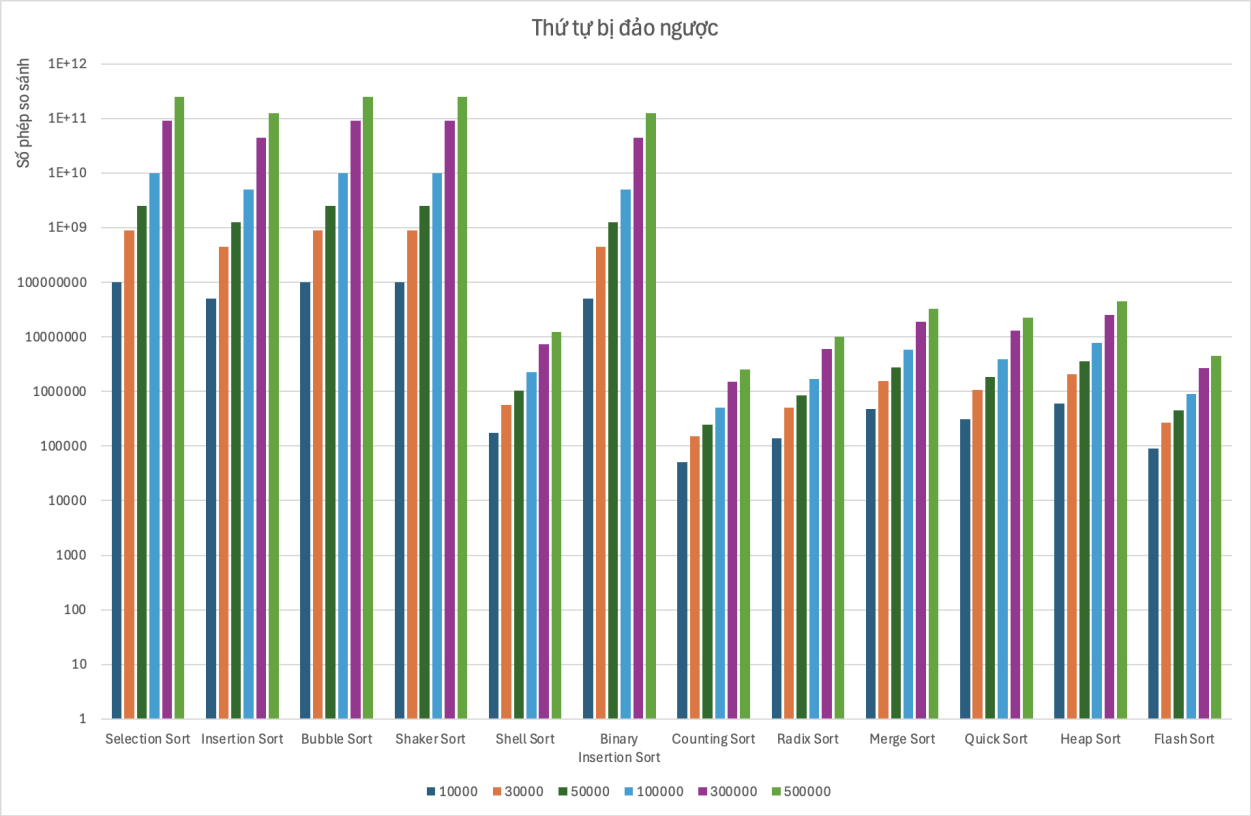
3.4.2 Biểu đồ đường thể hiện thời gian thực thi



Nhận xét:

- Thời gian chạy lớn nhất của Bubble Sort đạt 1032440 ms khi số phần tử là 500000, tiếp đó là Shaker Sort với thời gian 908751 ms khi số phần tử là 500000. Thời gian chạy thấp nhất thuộc về Counting Sort, Flash Sort với thời gian 0 ms khi số phần tử là 10000, và Counting Sort khi số phần tử là 30000.
- Các thuật toán có độ phức tạp thời gian trong Worst-case $O(n^2)$ hiển thị rõ ràng các đường riêng biệt trên biểu đồ. Ví dụ: Bubble Sort, Shaker Sort, Selection Sort, Insertion Sort, và Binary Insertion Sort theo thứ tự từ cao đến thấp. Sự khác biệt lớn giữa các thuật toán này trở nên rõ rệt kể từ khi dữ liệu có số phần tử từ 50000 trở đi.
- Trong khi đó, các thuật toán có độ phức tạp thời gian Worst-case $O(n \log n)$ hoặc $O(n)$ như Shell Sort, Counting Sort, Radix Sort, Merge Sort, Quick Sort, Heap Sort, và Flash Sort hầu như gắn liền với trục hoành trên biểu đồ. Ngay cả với dữ liệu có số phần tử lớn nhất, thời gian chạy của các thuật toán này vẫn nhỏ đến mức không đáng kể so với thời gian chạy lớn nhất của Bubble Sort.

3.4.3 Biểu đồ cột thể hiện số phép so sánh



Nhận xét:

- Số phép so sánh lớn nhất của Bubble Sort là 250000999998 khi số phần tử là 500000, tiếp đó là Shaker Sort với tổng số phép so sánh 250000500001 khi số phần tử là 500000. Số phép so sánh nhỏ nhất của Counting Sort là 50003 khi số phần tử là 10000.
- Đây là biểu đồ thể hiện bước nhảy theo cấp số nhân 10, với tỉ lệ các cột được thể hiện theo log cơ số 10. Khi cột này vượt qua mốc độ chia của giá trị, giá trị của nó sẽ gấp 10 lần cột một độ chia của giá trị trước đó.
- Có sự khác biệt rõ rệt giữa các thuật toán có độ phức tạp thời gian Worst-case $O(n^2)$ và $O(n \log n)$ hoặc $O(n)$. Các thuật toán có độ phức tạp $O(n^2)$ như Bubble Sort, Shaker Sort, Selection Sort, Binary Insertion Sort, và Insertion Sort đều vượt qua mốc $1E + 11$ khi số phần tử đạt 500000. Thậm chí, với dữ liệu có số phần tử nhỏ nhất (10000), các thuật toán $O(n^2)$ cũng có giá trị vượt quá $0.5E + 08$ một chút.
- Trong khi đó, các thuật toán có độ phức tạp thời gian Worst-case là $O(n \log n)$ hay $O(n)$ như Shell Sort, Counting Sort, Radix Sort, Merge Sort, Quick Sort, Heap Sort, Flash Sort, dù có dữ liệu 500000 phần tử, giá trị lớn nhất của chúng chỉ đạt khoảng $0.5E + 08$, thấp hơn so với các thuật toán $O(n^2)$.

Độ cao lớn nhất của các thuật toán này thuộc về Heap Sort khi số phần tử là 500000. Các thuật toán $O(n \log n)$ như Merge Sort, Quick Sort, Heap Sort, Binary Insertion Sort có độ cao gần như đồng đều, trong khi các thuật toán $O(n)$ như Counting Sort, Radix Sort, Flash Sort đều thấp hơn các thuật toán $O(n \log n)$.

– Với dữ liệu có số phần tử từ 30000 trở đi, các thuật toán có độ phức tạp thời gian Worst-case $O(n^2)$ sẽ có ít nhất số phép so sánh gấp 10 lần số phép so sánh của các thuật toán có độ phức tạp thời gian Worst-case $O(n \log n)$ hay $O(n)$.

Nhận xét 2 biểu đồ: Ở Worst-case thì mối liên hệ giữa số phép so sánh và thời gian chạy giữa các thuật toán đã khác đi nhiều (không còn tỉ lệ thuận như những dữ liệu trên). Càng nhiều số phép so sánh không đồng nghĩa thời gian chạy càng cao, thuật toán này có số phép so sánh nhiều hơn nhưng vẫn chạy nhanh hơn thuật toán có số phép so sánh ít hơn, thể hiện ở biểu đồ có 2 đường thẳng cắt nhau. Ví dụ như số phép so sánh Binary Insertion Sort lớn hơn Insertion Sort nhưng mà thời gian chạy của Binary Insertion Sort nhỏ hơn Insertion Sort. Hay là số phép so sánh của Selection Sort gần gấp đôi của số phép so sánh Insertion Sort, nhưng mà thời gian chạy của Selection Sort chỉ gấp 1.05 lần thời gian chạy của Insertion Sort. Nguyên nhân là với trường hợp Worst-case (mảng được sắp xếp ngược), thì có sự khác biệt trong phép gán, ban đầu phép so sánh thống trị, nhưng với mỗi phép sắp xếp 1 phần tử, có thuật toán chỉ có 2 phép gán (như Selection Sort), nhưng có thuật toán lại gán tới số phép bằng số phần tử còn lại chưa được sắp xếp (như Insertion Sort).

3.5 Kết luận

Nhận xét tổng thể:

– Thuật toán Selection Sort chạy chậm nhất trong hầu hết các trường hợp Average-case và Best-case.

– Thuật toán Bubble Sort và Shaker Sort chạy chậm nhất trong trường hợp Worst-case

– Thuật toán Counting Sort chạy nhanh nhất trong hầu hết các trường hợp Average-case và Worst-case, chỉ thua các thuật toán đã được tối ưu bằng cách kiểm tra mảng đã được sắp xếp trong Best-case như Bubble Sort, Shaker Sort. Tuy nhiên, không có nghĩa là luôn sử dụng Counting Sort trong mọi trường hợp cần sắp xếp dữ liệu. Cần phải căn cứ vào đầu vào kiểu dữ liệu (nếu là số âm hay số thực thì phải cải tiến lại thuật toán) hay số phần tử quá nhỏ so với khoảng cách giữa giá trị lớn nhất và giá trị nhỏ nhất trong mảng thì sẽ hoạt động tệ trong trường hợp đó, nếu khoảng cách từ số nhỏ nhất và số lớn nhất quá lớn thì không thể cài Counting Sort (lớn hơn số phần tử tối đa có thể tạo bằng mảng), và có chi phí khi đổi chỗ hay không (Selection Sort tốt nhất với số phép đổi chỗ ít nhất).

– Cũng có thể sử dụng các thuật toán có độ phức tạp thời gian trung bình $O(n \log n)$ như Merge Sort, Heap Sort, Quick Sort khi cần sử dụng so sánh.

Thuật toán sắp xếp ổn định (các số bằng nhau vẫn giữ vị trí tương đối với nhau sau sắp xếp, ví dụ hai số bằng nhau nhưng số a ban đầu ở trước số b thì sau sắp xếp số a vẫn ở trước số b): ***Counting Sort, Radix***

Sort, Bubble Sort, Shaker Sort, Insertion Sort, Binary Insertion Sort, Merge Sort. **Thuật toán sắp xếp không ổn định** (ngược lại với định nghĩa trên): ***Selection Sort, Shell Sort, Quick Sort, Heap Sort, Flash Sort.***

4 Tổ chức dự án và các ghi chú

4.1 Tổ chức dự án

Trong dự án này, chúng tôi đã sắp xếp các tệp mã của mình bằng cách sử dụng quy chuẩn thống nhất chung.

Các tập tin trong thư mục SOURCE/

- **Algorithms/header/*.hpp**: Những tệp tiêu đề ở đây sẽ khai báo 12 loại sắp xếp.
- **Algorithms/source/*.cpp**: Những tệp nguồn ở đây sẽ định nghĩa cho 12 loại sắp xếp.
- **Others/header/**: Những tệp tiêu đề để khai báo tên các hàm hỗ trợ nằm trong đây:

Các hàm trong **Others/header/dataGenerator.hpp**	
Tên hàm	**Công dụng**
`void generateRandomData(std::vector<int> &arr, int n)`	Hàm tạo những dữ liệu ngẫu nhiên dựa theo số lượng.
`void generateNearlySortedData(std::vector<int> &arr, int n)`	Hàm tạo những dữ liệu gần như sắp xếp dựa theo số lượng.
`void generateSortedData(std::vector<int> &arr, int n)`	Hàm tạo những dữ liệu đã sắp xếp dựa theo số lượng.
`void generateData(std::vector<int> &arr, int n, int dataType)`	Hàm tạo những dữ liệu dựa theo số lượng và kiểu dãy dữ liệu mong muốn.
`std::string getDataOrderName(int data_order_id)`	Hàm lấy tên của kiểu dãy dữ liệu dựa theo mã định danh của kiểu dãy dữ liệu đó.
`int getDataOrderID(char agr_flag[])`	Hàm lấy mã định danh của kiểu dãy dữ liệu dựa vào cờ command mà người dùng nhập.

Các định nghĩa trong **header/helpFunctions.hpp**	
Định nghĩa	**Công dụng**
`template <class T> void swap(T &a, T &b)`	Hàm hoán đổi giá trị 2 biến a và b, với mọi kiểu dữ liệu được định nghĩa T.
`int random(int l, int r)`	Hàm sinh 1 số ngẫu nhiên trong khoảng l đến r.
`bool readData(std::vector<int> &arr, std::string fileName)`	Hàm đọc dãy dữ liệu từ tệp tài nguyên, trả về true nếu đọc thành công và false nếu đọc thất bại.
`bool writeData(std::vector<int> &arr, std::string fileName)`	Hàm ghi dãy dữ liệu đã được sắp xếp hoặc dạng dãy dữ liệu theo yêu cầu ra các tệp tài nguyên, trả về true nếu đọc thành công và false nếu đọc thất bại.
`bool isNumber(char *str)`	Hàm kiểm tra chuỗi có phải là số nguyên hay không.
`int getOutputParameterID(char *param_flag)`	Hàm lấy mã của 3 loại tham số đầu ra: -time, -comp, -both dựa vào cờ lệnh, nếu không phải sẽ trả về -1.
`bool isOutputParameterID(char *param_flag)`	Hàm kiểm tra xem cờ lệnh có là các tham số đầu ra hay không.

Các định nghĩa trong **header/commandLine.hpp**	
Định nghĩa	**Công dụng**
`void executeWithCommandLine(SortExperiment &experiment, int &argv, char **&argc)`	Hàm xử lý các dòng lệnh nhập vào để chạy chương trình theo 5 loại thí nghiệm
`void commandLine..(SortExperiment &experiment, int &argv, char **&argc);`	5 hàm xử lý tùy vào từng loại dòng lệnh được yêu cầu.

Các định nghĩa trong **header/sortExperiment.hpp**	
Định nghĩa	**Công dụng**
`#define NUMBER_DATA_ORDER 4`	Định nghĩa tên cho số lượng của các loại dãy dữ liệu: ngẫu nhiên, gần như sắp xếp, đã sắp xếp, ngược.
`#define NUMBER_SORT_ALGORITHM 12`	Định nghĩa số lượng của các thuật toán sắp xếp đã được yêu cầu.
`#define .._SORT ..`	Định nghĩa cho 12 mã định danh cho 12 loại thuật toán sắp xếp.
`class ResultOfSorting`	Khai báo 1 lớp dữ liệu dành cho việc lưu 2 loại kết quả được yêu cầu là thời gian chạy (running_time) và số lần so sánh (count_comparison) của thuật toán sắp xếp.
`class SortExperiment`	Khai báo 1 lớp cho việc thí nghiệm của chương trình.
`std::vector< std::vector<int> > arr`	Khai báo mảng 2 chiều để lưu trữ 4 kiểu dãy dữ liệu để có thể sử dụng cho trường hợp cần cả 4 kiểu.
`std::string file_name`	Khai báo biến lưu tên của tệp tài nguồn dùng để đọc dãy dữ liệu vào chương trình thí nghiệm.
`int output_parameter`	Khai báo biến lưu mã định danh của tham số đầu ra.
`int input_size`	Khai báo biến để lưu kích thước của dãy dữ liệu
`int data_order_id`	Khai báo biến để lưu mã định danh của kiểu dãy dữ liệu.
`int algorithm_id[2]`	Khai báo mảng 2 phần tử dành cho việc thí nghiệm với chế độ so sánh 2 loại thuật toán sắp xếp, nếu ở chế độ thuật toán thì sẽ dùng phần tử đầu để nhận dạng mã định danh của các loại thuật toán.
`bool is_algorithm_mode`	Khai báo biến để lưu chế độ mà chương trình cần làm việc, cụ thể true nếu là chế độ thuật toán và false là chế độ so sánh.
`bool is_input_from_file`	Khai báo biến để lưu trạng thái có đọc dãy dữ liệu từ tệp tài nguyên hay không.
`bool is_running_all`	Khai báo biến để lưu trạng thái có chạy hết 4 loại dãy dữ liệu hay không.
`std::vector<ResultOfSorting> results`	Khai báo mảng Kết quả thuật toán để lưu kết quả của việc thí nghiệm, cụ thể mảng có 4 phần tử dành cho việc thí nghiệm trên 4 loại dãy dữ liệu, nếu chỉ thí nghiệm 1 loại dãy dữ liệu thì sẽ dùng phần tử đầu tiên của mảng để lưu trữ kết quả.

`SortExperiment()`	Khai báo phương thức để khởi tạo các giá trị mặc định cho các biến trong lớp này.
`~SortExperiment()`	Khai báo hàm hủy để tự động xóa sạch các loại dữ liệu trong lớp khi thoát chương trình.
`void runCompareMode()`	Khai báo hàm dùng để chạy chế độ so sánh.
`void runAlgorithmMode()`	Khai báo hàm dùng để chạy chế động thuật toán.
`void printResult()`	Khai hàm dùng để in kết quả của thí nghiệm ra màn hình máy tính.
`void sort(std::vector<int>..)`	Khai báo 2 hàm dành cho việc sắp xếp dãy dữ liệu dựa vào mã định danh thuật toán và dùng để kiểm tra thời gian hoặc số lần so sánh của loại thuật toán đó.
`std::string getAlgorithmName(int algorithm_id)`	Hàm dùng để lấy tên của loại thuật toán sắp xếp dựa vào mã định danh của loại thuật toán đó.
`int getAlgorithmID(std::string sort_name)`	Hàm dùng để lấy mã định danh của loại thuật toán sắp xếp dựa vào cờ lệnh được nhập vào chương trình đã được định nghĩa theo yêu cầu.

- **Others/source/.*cpp**: Những tệp nguồn để định nghĩa các hàm hỗ trợ được khai báo trong các tệp tiêu đề nằm trong đây, đặc biệt có tệp nguồn **main.cpp** là tệp đặc biệt dùng để liên kết và xử lý chương trình.

4.2 Các loại thư viện

- **iostream**: Thư viện này giúp chúng ta thực hiện các thao tác nhập và xuất cơ bản trong chương trình C++ của mình. Chúng ta có thể sử dụng nó để tạo giao diện thân thiện, nhận dòng lệnh từ người dùng và hiển thị kết quả trên màn hình.
- **vector**: Thư viện này giúp chúng ta sử dụng được dạng mảng chứa của lớp vector và một số hàm hỗ trợ.
- **fstream**: Với thư viện này, chúng ta có thể đọc dữ liệu từ các tệp và ghi dữ liệu vào các tệp trong chương trình C++ của mình. Nó giúp chúng ta lưu trữ thông tin cần thiết cho nghiên cứu hoặc theo dõi kết quả.
- **cstring**: Thư viện này có các hàm giúp thao tác chuỗi trong C++. Vì các đối số dòng lệnh được biểu diễn dưới dạng chuỗi char*, chúng ta có thể sử dụng thư viện này để xử lý và sửa đổi chúng.
- **ctime**: Thư viện này dùng để hỗ trợ trong việc tạo ra các giá trị ngẫu nhiên.
- **chrono**: Thư viện này cung cấp các công cụ để đo thời gian trong chương trình C++ của chúng tôi. Chúng tôi sử dụng nó để đo hiệu suất của các thuật toán và theo dõi thời gian hoàn thành.

4.3 Hướng các biên dịch và chạy chương trình

Để compile file dùng để chạy command, đồ án sử dụng g++ với cú pháp sau: Mô tả từng bước:

Bước 1: Di chuyển vào thư mục tổng (chứa 01.exe, SOURCE/,...)

Bước 2: Nhập lệnh sau vào terminal hoặc command prompt:

g++ SOURCE/Algorithms/source/*.cpp SOURCE/Others/source/*.cpp -o 01.exe

Trong cú pháp trên:

- SOURCE/Algorithms/source/*.cpp: Compile các file cpp trong thư mục Algorithms - Các file hàm thuật toán sắp xếp.
- SOURCE/Others/source/*.cpp: Compile các file cpp dùng cho việc hỗ trợ thí nghiệm trong thư mục Others.
- 01.exe: Kết quả quá trình compile các file trên sẽ được viết vào file **01.exe**.

Ở đây người dùng có thể sử dụng tên file khác để đặt cho file exe dùng để chạy chương trình.

Bước 3: Chạy chương trình bằng lệnh **./01.exe** cùng 5 loại lệnh đã được yêu cầu.

Ngoài ra chúng tôi còn tích hợp thêm chế độ thí nghiệm cho tất cả loại dãy dữ liệu với 12 loại thuật toán cùng các số lượng dữ liệu là 10, 30, 50, 100, 300, 500 sẽ được ghi vào các file csv và in ra màn hình để xem với lệnh: **./01.exe -experiment**

Lưu ý: Chúng tôi đã cài đặt cờ sau để tránh số lượng dữ liệu thực tế của yêu cầu Lab 3 quá lớn để có thể chờ, vậy nên chúng tôi đã cài đặt cờ **#define EXPERIMENT** để giảm 1000 lần số lượng dữ liệu thực thế của yêu cầu Lab 3. Cờ này nằm ở tệp main.cpp, nếu muốn số lượng dữ liệu như yêu cầu lab 3 thì có thể tắt nó.

Tài liệu

[1] GeeksforGeeks. *Selection Sort Algorithm*. https://www.geeksforgeeks.org/selection-sort-algorithm-2/. Truy cập: 21-11-2024.

[2] Wikipedia. *Improvement Selection Sort Algorithm*. https://en.wikipedia.org/wiki/Selection_sort. Truy cập: 21-11-2024.

[3] Văn Chí Nam. *Sorting Algorithms*. Bài giảng học kỳ 1 năm học 2024-2025. Trang 17 đến 21. 2024.

[4] Wikipedia. *Odd-even Sort Algorithm*. https://en.wikipedia.org/wiki/Odd-even_sort. Truy cập: 23-11-2024.

[5] Wikipedia. *Shell sort Sort Algorithm*. https://en.wikipedia.org/wiki/Odd-even_sort. Phần Gap sequences. Truy cập: 25-11-2024.

[6] GeeksforGeeks. *Binary Insertion Sort*. https://www.geeksforgeeks.org/binary-insertion-sort/. Truy cập: 27-11-2024.

[7] GeeksforGeeks. *Counting Sort*. https://www.geeksforgeeks.org/counting-sort/. Phần How does Counting Sort Algorithm work?. Truy cập: 27-11-2024.

[8] Wikipedia. *Radix Sort*. https://en.wikipedia.org/wiki/Radix_sort. Truy cập: 17-11-2024.

[9] GeeksforGeeks. *Radix Sort*. https://www.geeksforgeeks.org/radix-sort/. Phần How does Radix Sort Algorithm work?. Truy cập: 17-11-2024.

[10] GeeksforGeeks. *Merge Sort*. https://www.geeksforgeeks.org/merge-sort/. Phần How does Merge Sort Algorithm work?. Truy cập: 17-11-2024.

[11] GeeksforGeeks. *Iterative Merge Sort*. https://www.geeksforgeeks.org/iterative-merge-sort/. Truy cập: 18-11-2024.

[12] GeeksforGeeks. *Hybrid Sort*. https://www.geeksforgeeks.org/hybrid-sorting-algorithms/. Truy cập: 18-11-2024.

[13] Văn Chí Nam. *Sorting Algorithms*. Bài giảng học kỳ 1 năm học 2024-2025. Trang 44 đến 48. 2024.

[14] CodeLearn. *Flash Sort*. https://en.wikipedia.org/wiki/Flashsort. Truy cập: 20-11-2024.

[15] GeeksforGeeks. *Heap Sort*. https://www.geeksforgeeks.org/heap-sort. Truy cập: 21-11-2024.

[16] thanguyen165. *HCMUS-sorting-algorithms*. https://github.com/thanguyen165/HCMUS-sorting-algorithms/tree/main. Tham khảo một số mã nguồn cài đặt cho việc đếm thời gian và số phép so sánh. Truy cập 21-11-2024.

[17] vnpq. *Sorting Project*. https://github.com/vnpq/HCMUS-DSA/tree/main/Lab3. Tham khảo định dạng báo cáo bằng latex. Truy cập 20-11-2024.