ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

Báo cáo về các giải thuật sắp xếp

Thực hành cấu trúc dữ liệu và giải thuật

Sinh viên thực hiện

Nguyễn Văn Hưng - 20120009

Giảng viên hướng dẫn

Nguyễn Thanh Phương
Bùi Huy Thông

Tháng 11 năm 2021

Mục Lục

1. Giới thiệu tổng quan

Bài báo cáo bao gồm 11 thuật toán sắp xếp với các loại dữ liệu khác nhau. Cùng thêm đó là các bảng và biểu đồ thể hiện thời gian chạy và số bước so sánh trong một thuật toán sắp xếp.

Cấu hình máy chạy để thực thi cũng như đo các kết quả tính toán:

- Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz
- RAM: 8GB

Dữ liệu được sinh có giới hạn sau:

- $1 \leq n \leq 5 \cdot 10^5$
- $0 \leq a_i < n$

Để dịch trường trình sang file thực thi, ta cần di chuyển đến thi mục chứa file main.cpp và dịch với lệch sau: g++ main.cpp Sort.cpp SortComp.cpp -std=c++14 -o main

2. Các thuật toán sắp xếp

2.1. Selection Sort

2.1.1. Ý tưởng

Ý tưởng của thuật toán sắp xếp chọn khá là cơ bản. Ban đầu mảng sẽ chia làm hai phần: Phần bên trái sẽ chứa các phần tử đã sắp xếp và phần bên phải chứa các phần tử chưa được sắp.

Khi đó thuật toán sẽ thực hiện là chọn ra phần tử nhỏ nhất của phần bên phải và thêm vào cuối của phần bên trái. Thao tác sẽ lặp lại cho đến khi phần bên phải rỗng.

2.1.2. Các bước giải thuật

Thử chạy từng bước với mảng $a = \{5, 2, 4, 1, 3\}$. Trong đó phần được sắp bên trái để dễ nhìn ta sẽ tô màu xanh và phần tử nhỏ nhất trong mỗi lần tìm kiếm ta sẽ để màu đỏ.

Các bước	Mảng a	Chú thích
1	{5, 2, 4, 1, 3}	Phần tử nhỏ nhất là 1. Đổi chỗ với 5 - điều này tương tự như việc thêm 1 vào sau phần bên trái được sắp (lúc này đang rỗng).
2	{1, 2, 4, 5, 3}	Phần tử nhỏ nhất là 2 sau đó sẽ được đưa vào sau phần được sắp bên trái
3	{1, 2, 4, 5, 3}	Tương tự như bước 1
4	{1, 2, 3, 5, 4}	
5	{1, 2, 3, 4, 5}	
6	{1, 2, 3, 4, 5}	Kết thúc thuật toán

2.1.3. Đánh giá độ phức tạp

Thời gian:

Với n là kích thước của mảng ta có thể thấy để tìm kiếm phần tử nhỏ nhất trong mỗi lần lặp:

- Lần 1: Tốn $n-1$ bước
- Lần 2: Tốn $n-2$ bước
- ...
- Lần n - 1: Tốn 1 bước.

Và các thao tác trên không phụ thuộc vào bộ dữ liệu do đó tổng số bước là:

$$(n-1)+(n-2)+...+1=\frac{(n-1)\cdot n}{2}$$

Vậy ta có bảng đánh giá sau:

Worst Case	Average Case	Best Case
$O(n^2)$	$O(n^2)$	$O(n^2)$

Không gian: $O(1)$.

2.2. Insertion Sort

2.2.1. Ý tưởng

Ý tưởng sắp xếp chèn bước khởi đầu khá là giống Selection Sort là chia mảng ban đầu thành hai phần: Phần bên trái sẽ chứa các phần tử đã sắp xếp và phần bên phải chứa các phần tử chưa được sắp.

Khi đó thuật toán sẽ chọn phần tử đầu tiên trong phần chưa được sắp và chèn vào phần bên phải sao cho đúng vị trí.

2.2.2. Các bước giải thuật

Xét mảng $a=\{5,2,4,1,3\}$, các phần tử thuộc phần bên trái được sắp và tô màu xanh, phần tử được chọn đầu tiên của phần bên phải mỗi lần duyệt sẽ tô màu đỏ.

Các bước	Mảng a	Chú thích
1	{5, 2, 4, 1, 3}	Phần tử đầu tiên ta sẽ đưa vào tập bên trái và bắt đầu tính từ vị trí kế tiếp
2	{5, 2, 4, 1, 3} → {2, 5, 4, 1, 3}	Đưa phần tử 2 về đúng vị trí của phần đã sắp bên trái.
3	{2, 5, 4, 1, 3} → {2, 4, 5, 1, 3}	Tương tự như bước 2, ta đưa 4 về đúng vị trí của nó.
4	{2, 4, 5, 1, 3} → {1, 2, 4, 5, 3}	
5	{1, 2, 4, 5, 3} → {1, 2, 3, 4, 5}	Tới đây thuật toán kết thúc.

2.2.3. Đánh giá độ phức tạp

Thời gian:

Nhận thấy rằng ta sẽ cần duyệt qua $n-1$ lần và mỗi lần như vậy ta cần đưa phần tử đầu tiên của phần bên phải vào vị trí thích hợp của phần bên trái đã được sắp. Tuy nhiên việc tìm ra vị trí thích hợp như thế thì trong trường hợp tệ nhất có thể phân tích như sau:

- Lần 1: Tốn 1 bước để tìm ra vị trí thích hợp

- Lần 2: Tốn 2 bước
- ...
- Lần n - 1: Tốn $n-1$ bước

Do đó trường hợp xấu nhất thì độ phức tạp có thể lên tới $O(n^2)$. Trường hợp tốt nhất dễ thấy đó chính là vị trí đầu tiên là vị trí thích hợp và độ phức tạp cho trường hợp này là $O(n)$. Độ phức tạp trung bình sẽ là $O(n^2)$.

Worst Case	Average Case	Best Case
$O(n^2)$	$O(n^2)$	$O(n)$

Không gian: $O(1)$

2.3. Bubble Sort

2.3.1. Ý tưởng

Sắp xếp nổi bọt là một thuật toán có ý tưởng đơn giản. Mục tiêu là trong mỗi lần duyệt, ta sẽ đưa phần tử lớn nhất về cuối mảng.

Để làm được điều đó, trong mỗi lần duyệt ta sẽ kiểm tra xem phần tử ở trước mà lớn hơn phần tử ngay sau đó hay không, thì ta sẽ đổi chỗ hai phần tử này. Khi đó kết thúc mỗi lần duyệt thì sẽ có thêm một phần tử lớn nhất trong lần duyệt đó được đưa về cuối mảng.

2.3.2. Các bước giải thuật

Cách hoạt động của Bubble Sort khá là đơn giản, trong ví dụ dưới đây ta sẽ quan sát mỗi lần so sánh phần tử lớn nhất sẽ "nổi lên" như thế nào.

Các bước	Mảng a
1	{5, 2, 4, 1, 3} → {2, 5, 4, 1, 3}
2	{2, 5, 4, 1, 3} → {2, 4, 5, 1, 3}
3	{2, 4, 5, 1, 3} → {2, 4, 1, 5, 3}
4	{2, 4, 1, 5, 3} → {2, 4, 1, 3, 5}
5	{2, 4, 1, 3, **5**}
6	{2, 4, 1, 3, **5**} → {2, 1, 4, 3, **5**}
7	{2, 1, 4, 3, **5**} → {2, 1, 3, 4, **5**}
8	{2, 1, 3, **4**, **5**} → {1, 2, 3, **4**, **5**}
9	{1, 2, 3, **4**, **5**}
10	{1, 2, **3**, **4**, **5**}
11	{**1**, **2**, **3**, **4**, **5**}

2.3.3. Cải tiến

Nhận thấy rằng trong quá trình duyệt để "làm nổi" phần tử lớn lên trên, nếu như không có phép đổi chỗ nào tức là lúc này mảng được sắp xếp. Do đó ta có thể không xét tiếp nữa.

2.3.4. Đánh giá độ phức tạp

Thời gian:

Cùng với cải tiến đã nêu ở trên ta có thể thấy rằng trường hợp tốt nhất là mảng đã sắp sẵn hoặc gần như sắp sẵn thì độ phức tạp của Bubble Sort sẽ $O(n)$. Tất nhiên trường hợp trung bình và trường hợp xấu nhất sẽ là $O(n^2$ với mỗi lần duyệt ta đều có phải thực hiện thao tác đổi chỗ.

Tổng quan thì với cải tiến trên thì Bubble Sort chạy rất nhanh với mảng đã sắp hoặc gần như được sắp.

Worst Case	Average Case	Best Case
$O(n^2)$	$O(n^2)$	$O(n)$

Không gian: $O(1)$

2.4. Shell Sort

2.4.1. Ý tưởng

Shell Sort là một giải thuật dựa trên sắp xếp chèn (Insertion Sort), tuy nhiên tính hiệu quả nó đạt được là cao hơn.

Với Insertion Sort, trong trường hợp tệ nhất thì ta có thể tốn rất nhiều thời gian để đưa phần tử vào đúng vị trí do chỉ được hoán đổi hai phần từ kề nhau, thì trong giải thuật này cho phép hoán vị các phần tử cách xa nhau.

Cụ thể hơn, ta sẽ sắp xếp lại các mảng con với khoảng cách là **gap** và dần dần thu hẹp **gap** về 1.

Ví dụ mảng con $a_0, a_h, a_{2h}, ...$ được xem như là mảng con với khoảng cách h.

Trong thuật thoán Shell Sort có rất nhiều cách để chọn **gap** ví dụ như:

- $\{n/2, n/4, ..., 1\}$ (gap "nguyên bản")
- $\{1, 3, 7, ..., 2^k - 1\}$ (do Hibbard đề xuất)
- $\{1, 4, 13, ..., (3^k - 1)/2\}$ (do Knuth đề xuất)
- ...

2.4.2. Các bước giải thuật

Cùng chạy thử trên mảng $a = \{5, 2, 4, 1, 3\}$ với $gap = \{n/2, n/4, ..., 1\}$

Để dễ quan sát hơn, ta sẽ để các phần tử cùng thuộc một *gap* sẽ có cùng màu.

Các bước	Gap	Mảng a	Chú thích
0	2	{5, 2, 4, 1, 3}	Mảng a ban đầu có hai mảng con với khoảng cách $gap = 2$ là {5, 4, 3} và {2, 1}
1	2	{**4**, 2, 5, 1, 3}	Trên mảng con màu xanh dương, chèn phần tử 4 về vị trí thích hợp.
2	2	{**3**, 2, 4, 1, 5}	Trên mảng con màu xanh dương, chèn phần tử 3 về vị trí thích hợp.
3	2	{3, **1**, 4, 2, 5}	Trên mảng con màu xanh lá, chèn phần tử 1 về vị trí thích hợp
4	1	{3, **1**, 4, 2, 5} → {**1**, 3, 4, 2, 5}	Lúc này $gap = 1$ khá giống Insertion Sort, 1 chưa đúng vị trí ta chèn vào vị trí đầu tiên thích hợp
5	1	{1,**2**, 3, 4, 5}	Phần tử 2 được đưa về đúng vị trí và có thể thấy đến đây mảng đã được sắp.

2.4.3. Đánh giá độ phức tạp

Thời gian

Trường hợp tốt nhất và trung bình là $O(n \log n)$ với việc cho phép đổi chỗ 2 phần tử cách xa nhau do đó số lượng phép so sánh khá là ít đối với đa số bộ dữ liệu.

Trường hợp tệ nhất là $O(n^2$ khi mà phải thực hiện nhiều nhất có thể số bước hoán vị.

Worst Case	Average Case	Best Case
$O(n \log n)$	$O(n \log n)$	$O(n)$

Không gian: $O(1)$

2.5. Shaker Sort

2.5.1. Ý tưởng

Đây là một giải thuật cải tiến từ Bubble Sort. Thay vì mỗi lần duyệt ta chỉ đưa phần tử lớn nhất về cuối thì ta thực hiện một thao tác nữa đó là đưa thêm phần tử nhỏ nhất lên đầu.

2.5.2. Các bước giải thuật

Cách hoạt động khá giống với Bubble Sort, tuy nhiên là ngoài phần các phần tử lớn "nổi lên" thì còn có các phần tử nhỏ "chìm xuống", ta sẽ quan sát thử 2 đầu của mảng a trong test minh họa dưới đây.

Các bước	Mảng a
1	{5, 2, 4, 1, 3} → {2, 5, 4, 1, 3}
2	{2, 5, 4, 1, 3} → {2, 4, 5, 1, 3}
3	{2, 4, 5, 1, 3} → {2, 4, 1, 5, 3}
4	{2, 4, 1, 5, 3} → {2, 4, 1, 3, 5}
5	{2, 4, 1, 3, **5**}
6	{2, 4, 1, 3, **5**} → {2, 1, 4, 3, **5**}
7	{2, 1, 4, 3, **5**} → {1, 2, 4, 3, **5**}
8	{**1**, 2, 4, 3, **5**}
9	{**1**, 2, 4, 3, **5**} → {**1**, 2, 3, 4, **5**}
10	{**1**, **2**, **3**, **4**, **5**}

2.5.3. Cải tiến

CŨng như Bubble Sort, ta có thể dừng thuật toán khi trong một lần thực hiện mà không có phép hoán vị nào xảy ra.

2.5.4. Đánh giá độ phức tạp

Thời gian:

Trường hợp xấu nhất và trung bình giải thuật này có độ phức tạp là $O(n^2)$ và tốt nhất là $O(n)$.

Worst Case	Average Case	Best Case
$O(n^2)$	$O(n^2)$	$O(n)$

Không gian: $O(1)$

2.6. Heap Sort

2.6.1. Max-heap

Max-heap là một cây nhị phân trong đó giá trị của nút cha sẽ lớn hơn giá trị của 2 nút con.

Ta có thể biểu diễn dưới dạng mảng:

- Với mảng đánh số từ 0, thì nếu nút cha được lưu tại chỉ mục i thì 2 nút con tương ứng là $2*i+1$ và $2*i+2$.
- Với mảng đánh số từ 1, thì nếu nút cha được lưu tại chỉ mục i thì 2 nút con tương ứng là $2*i$ và $2*i+1$.

2.6.2. Ý tưởng

Đây là một thuật toán có ý tưởng khá hay khi đưa cấu trúc dữ liệu Heap dưới dạng mảng một chiều và thao tác sắp xếp lại không cần tốn thêm bộ nhớ.

Thuật toán sắp xếp này bao gồm 2 giai đoạn chính:

Bước 1. Xây dựng Max-heap: Bằng cách sử dụng các thao tác hiệu chỉnh (heapify) để từ mảng bình thường thành mảng heap.

Bước 2. Sắp xếp lại mảng. Giả sử mảng hiện tại có kích thước là n:

- Hoán đổi phần tử cuối đầu tiên với phần tử cuối cùng. Vì đây là Max-heap do đó phần tử cuối cùng này sẽ mang giá trị lớn nhất.
- Tiếp tục duy trì Max-heap trên $n-1$ phần tử còn lại với thao tác hiệu chỉnh (heapify).
- Thực hiện lại thao tác đầu tiên với kích thước mảng lúc này là $n-1$.

Thao tác hiệu chỉnh có thể tóm tắt như sau: Nếu như ta cần hiệu chỉnh lại ở chỉ mục i, ta xem xét xem liệu tại nút i có lớn hơn 2 nút con tương ứng hay không, nếu không thỏa ta sẽ đổi chỗ sao cho thỏa mãn và hiệu chỉnh nút con tương ứng.

2.6.3. Các bước giải thuật

Ta sẽ chạy từng bước thuật toán với $a = \{5, 2, 4, 1, 3\}$ và mảng đánh số từ 0. Ta sẽ tô màu xanh tương ứng với nút cha và màu vàng cho nút con. Các phần từ in đậm được coi là đã cố định và không nằm trong mảng heap nữa.

Các bước	Giai đoạn	Mảng a	Chú thích
1	Xây dựng Heap	{5, 2, 4, 1, 3} → {5, 3, 4, 1, 2}	Phần tử 2 có chỉ mục là 1 và hai nút con tương ứng là 1 và 3 có chỉ mục tương ứng là 3 và 4. Ta thấy rằng $2 < 3$ do đó ta sẽ đổi chỗ lại để thỏa mãn sau đó đi xuống nút con tương ứng vừa được đổi để tiếp tục hiệu chỉnh.
2	Xây dựng Heap	{5, 3, 4, 1, 2}	Phần tử tại chỉ mục 4 không có nút con.
3	Xây dựng heap	{5, 3, 4, 1, 2}	Nhận thấy rằng nút cha với giá trị 5 đã thỏa mãn lớn hơn hai nút con tương ứng rồi nên sẽ không có sự thay đổi tại đây. Lúc này, ta cũng đã hoàn thành công đoạn xây dựng heap.
4	Sắp xếp mảng	{2, 3, 4, 1, **5**}	Đổi chỗ phần tử lớn nhất về sau.
5	Sắp xếp mảng	{2, 3, 4, 1, **5**} → {4, 3, 2, 1, **5**}	Ở đây ta sẽ hiệu chỉnh lại heap từ chỉ mục 0. Ta thấy rằng nút cha mang giá trị 2 nhỏ hơn 4 do đó ta sẽ đổi chỗ lại và tiếp tục hiệu chỉnh tại nút con tương ứng trong bước tiếp theo.
6	Sắp xếp mảng	{4, 3, 2, 1, **5**}	Tại nút có chỉ mục 2 này không tồn tại nút con nên là không thực hiện.
7	Sắp xếp mảng	{1, 3, 2, **4**, **5**}	Đổi chỗ phần tử lớn nhất về sau.
8	Sắp xếp mảng	{1, 3, 2, **4**, **5**} → {3, 1, 2, **4**, **5**}	Tiếp tục hiệu chỉnh heap theo quy tắc trên.
9	Sắp xếp mảng	{1, 2, **3**, **4**, **5**}	Đổi chỗ phần tử lớn nhất về sau.
10	Sắp xếp mảng	{1, 2, **3**, **4**, **5**} → {2, 1, **3**, **4**, **5**}	Tiếp tục hiệu chỉnh heap theo quy tắc trên.
11	Sắp xếp mảng	{1, **2**, **3**, **4**, **5**}	Đổi chỗ phần tử lớn nhất về sau.
12	Sắp xếp mảng	{**1**, **2**, **3**, **4**, **5**}	Kết thúc thuật toán.

2.6.4. Đánh giá độ phức tạp

Thời gian:

Ở bước xây dựng heap, nhìn tổng quan ta có thể thấy rằng ta sẽ hiệu chỉnh lại các nút con trong

đoạn 0 đến $\lfloor \frac{n}{2} \rfloor$ (do các phần tử ngoài đoạn này thì không có nút con). Mà mỗi thao tác hiệu chỉnh là $O(\log n)$ do đó độ phức tạp ở bước xây dựng heap là $O(n \log n)$. Tuy nhiên nếu đánh giá kĩ hơn thì thao tác xây dựng heap này chỉ tốn là $O(n)$.

Còn ở bước sắp xếp lại, ta thấy rằng cần n lần hiệu chỉnh lại heap sau mỗi lần đổi chỗ, mà mỗi lần đổi chỗ chi phí có thể sẽ là độ cao của cây nhị phân này. Độ cao của cây có thể lên tới $\log n$.

Do đó trường hợp xấu nhất và trung bình thì Heap Sort sẽ có độ phức tạp là $O(n \log n)$ và trường hợp tốt nhất sẽ là $O(n)$.

Worst Case	Average Case	Best Case
$O(n \log n)$	$O(n \log n)$	$O(n)$

Không gian: $O(1)$ hoặc $O(\log n)$ tùy thuộc vào cách cài đặt hàm heapify có sử dụng đệ quy hay không.

2.7. Merge Sort

2.7.1. Ý tưởng

Thuật toán săp xếp trộn này áp dụng tư tưởng chính của chia để trị. Hàm sắp xếp sử dụng với mảng a nhiều hơn một phần tử như sau:

- Chia dãy ra làm 2 phần, bên trái tạm gọi là a_{left} bên phải tạm gọi là a_{right}
- Gọi hàm sắp xếp lại a_{left} và a_{right}
- Trộn 2 mảng con a_{left} và a_{right} lại với nhau

2.7.2. Các bước giải thuật

Xét mảng $a = \{5, 2, 4, 1, 3\}$

Các bước	Mảng a	Chú thích
1	{5, 2, 4} {1, 3}	Tách mảng ban đầu ra 2 phần.
2	{5, 2} {4} {1, 3}	Tiếp tục chia nhỏ mảng con {5, 2, 4} thành hai phần nhỏ hơn.
3	{5} {2} {4} {1, 3}	Chia nhỏ {5, 2} ra làm hai.
4	{2, 5} {4} {1, 3}	Trộn {5} và {2} lại với nhau.
5	{2, 4, 5} {1, 3}	Trộn {2, 5} và {4} lại với nhau.
6	{2, 4, 5} {1} {3}	Chia nhỏ {1, 3} ra làm hai.
7	{2, 4, 5} {1, 3}	Trộn {1} và {3} lại với nhau.
8	{1, 2, 3, 4, 5}	Trộn {2, 4, 5} và {1, 3} lại với nhau và kết thúc thuật toán.

2.7.3. Đánh giá độ phức tạp

Thời gian:

Ta thấy rằng mỗi lần ta chia nhỏ mảng ra làm hai phần, do đó số tầng đệ quy tối đa sẽ là $\log n$. Tuy nhiên tổng số thao tác cho việc trộn 2 mảng con ở mỗi tầng là n. Do đó độ phức tạp của thuật toán trong mọi trường hợp sẽ là $O(n \log n)$

Worst Case	Average Case	Best Case
$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

Thời gian: $O(n)$ trong đó $O(n)$ cho việc sử dụng thêm mảng phụ và $O(\log n)$ cho số tầng đệ quy

2.8. Quick Sort

2.8.1. Ý tưởng

Đây cũng là một giải thuật áp dụng chia để trị để giải quyết bài toán. Vơi mảng a có nhiều hơn 1 phần tử:

- Chọn ra một phần tử chốt (pivot)
- Tiến hành phân hoạch mảng a sao cho các phần tử nhỏ hơn pivot sẽ nằm bên trái (tạm gọi là a_{left}) và các phần tử lớn hơn pivot nằm ở bên phải (tạm gọi là a_{right}).
- Tiến hành sắp xếp trên 2 mảng con a_{left} và a_{right}.

2.8.2. Các bước thuật toán

Chạy thử trên $a = \{5, 2, 4, 1, 3\}$, chọn chốt *pivot* trong một đoạn con là phần tử bên phải của đoạn con đó.

Các bước	Mảng a	Chú thích
1	$\{\underline{5,\ 2,\ 4,\ 1,\ \mathbf{3}}\} \rightarrow \{\underline{2,\ 1,\ \mathbf{3},\ 5,\ 4}\}$	Chọn chốt $pivot = 3$ và phân hoạch lại sao cho bên trái sẽ là các phần tử nhỏ hơn hoặc bằng 3 và bên phải là các phần tử lớn hơn 3.
2	$\{\underline{2,\ \mathbf{1}},\ 3,\ 5,\ 4\} \rightarrow \{\underline{\mathbf{1},\ 2},\ 3,\ 5,\ 4\}$	Xét mảng con $\{2, 1\}$, chọn $pivot = 1$ và phân hoạch lại mảng con này.
3	$\{1,\ 2,\ 3,\ \underline{5,\ \mathbf{4}}\} \rightarrow \{1,\ 2,\ 3,\ \underline{\mathbf{4},\ 5}\}$	Xét mảng con $\{5, 4\}$, chọn $pivot = 4$ và phân hoạch lại mảng con này. Lúc này mảng đã được sắp xếp.

2.8.3. Đánh giá độ phức tạp

Thời gian:

Độ phức tạp trong trường hợp tốt nhất và trung bình của thuật toán này là $O(n \log n)$ khi chốt *pivot* được chọn đủ đẹp để sau khi phân hoạch ta được 2 nửa có kích thước không qua chênh lệch nhau.

Trường hợp xấu nhất có độ phức tạp là $O(n^2)$ khi chốt được chọn gần như là lớn nhất hoặc nhỏ nhất.

Không gian: $O(n)$ với n là độ sâu trong trường hợp tệ nhất khi gọi đệ quy.

2.8.4. Cải tiến

Trong Quick Sort còn có một phiên bản cải tiến và có trong Java đó là Dual Pivot Quick Sort. Tuy nhiên cải tiến này chỉ giảm khả năng rơi vào trường hợp chọn chốt xấu nhất và trường hợp xấu nhất vẫn là $O(n^2)$.

2.9. Counting Sort

2.9.1. Ý tưởng

Sắp xếp đếm phân phối sử dụng một mảng để đếm số lần xuất hiện của các phần tử trong mảng (có thể gọi là mảng đếm phân phối). Sau đó duyệt qua mảng đếm phân phối này sẽ giúp ta đưa ra được mảng sắp xếp.

2.9.2. Các bước thuật toán

Giả sử ta có mảng $a = \{5, 1, 5, 2, 3, 5, 1, 1\}$. Ta sử dụng một mảng *cnt* làm mảng đếm phân phối với $cnt[x]$ có ý nghĩa là số lượng phần tử mang giá trị x có trong mảng a. Có thể dễ dàng tính được mảng *cnt* tương ứng là $cnt[0..5] = \{0, 3, 1, 0, 1, 3\}$.

Dựa vào mảng trên, bằng cách for tuần tự qua mảng *cnt* thì có thể đưa ra mảng a đã được sắp xếp.

2.9.3. Đánh giá độ phức tạp

Thời gian:

Đặt $m = max_a - min_a + 1$ trong đó max_a và min_a lần lượt là giá trị lớn nhất vả nhỏ nhất của mảng a, ta thấy rằng độ phức để for qua mảng a để tính mảng *cnt* sẽ là $O(n)$ và duyệt lại mảng *cnt* để sort sẽ có độ phức tạp là $O(m)$

Vậy độ phức tạp là $O(n + m)$.

Worst Case	Average Case	Best Case
$O(n + m)$	$O(n + m)$	$O(n + m)$

Không gian: Tùy vào cách cài đặt có thể tốn độ phức tạp là $O(m)$ hoặc $O(n + m)$.

2.10. Radix Sort

2.10.1. Ý tưởng

Đây là một thuật toán khá thú vị khi không sử dụng các phép so sánh là thao tác cơ sở như các thuật toán Sort khác. Thay vào đó là, Radix Sort phân loại các phần tử theo cơ số nào đó.

Cụ thể hơn, ta sẽ sử dụng cơ số 10. Giả sử các phần tử này có tối đa m chữ số. Ta tiến hành xét các phần tử từ hàng đơn vị đi lên - cách duyệt này gọi là least significant digit (LSD). Khi xét đến chữ số thứ k nào đó, ta sẽ sử dụng Counting Sort để sắp xếp lại mảng tại vị trí này.

2.10.2. Các bước thuật toán

Ta thực hiện trên mảng $a = \{2145, 2131, 2358, 542, 3151\}$

Các bước	Chữ số hàng	Mảng a	Chú thích
1	Đơn vị	{2145, 2131, 2358, 542, 3151} → {2131, 3151, 542, 2145, 2358}	Chữ số hàng đơn vị tương ứng là {5, 1, 8, 2, 1}, ta thấy rằng 2131 và 3151 có cùng chữ số hàng đơn vị, tuy nhiên 3151 xuất hiện sau nên sau khi sắp xếp lại cũng phải ở phía sau.
2	Chục	{2131, 3151, 542, 2145, 2358} → {2131, 542, 2145, 3151, 2358}	Chữ số hàng chục tương ứng là {3, 5, 4, 4, 5}.
3	Trăm	{2131, 542, 2145, 3151, 2358} → {2131, 2145, 3151, 2358, 542}	Chữ số hàng trăm tương ứng là {1, 5, 1, 1, 3}.
4	Nghìn	{2131, 2145, 3151, 2358, 542} → {542, 2131, 2145, 2358, 3151}	Chữ số hàng nghìn tương ứng là {2, 2, 3, 2, 0}.

2.10.3. Đánh giá độ phức tạp

Thời gian:

Với m là số lượng chữ số trong số lớn nhất, thì Radix sort có độ phức tạp là $O(nm)$ trong mọi trường hợp.

Worst Case	Average Case	Best Case
$O(nm)$	$O(nm)$	$O(nm)$

Không gian: $O(n)$ cho việc tốn thêm mảng phụ

2.10.4. Các phiên bản khác

Ngoài cách duyệt từ chữ số hàng đơn vị lên ta có thể duyệt từ hàng cao nhất về và được gọi là most significant digit (MSD).

Bên cạnh đó ta cũng có thể sử dụng hệ cơ số 2 bằng các thao tác duyệt qua các bit của các số trên mảng a.

2.11. Flash Sort

2.11.1. Ý tưởng

Thuật toán Flash Sort với ý tưởng là ta sẽ chọn ra m được gọi là số lượng phân lớp. Sau đó sẽ phân hoạch lại các phần tử của mảng a về đúng phân lớp của nó.

Cụ thể hơn phần tử a_i sẽ thuộc phân lớp thứ $\lfloor (m-1) \cdot \frac{a_i - min_a}{max_a - min_a} \rfloor$ trong đó min_a và max_a lần lượt là số lớn nhất và số nhỏ nhất của mảng a.

Sau khi phân hoạch cách phần tử về đúng phân lớp thì với mỗi phân lớp ta sẽ sử dụng sắp xếp chèn (Insertion Sort) để sắp xếp lại các phần tử trong cùng phân lớp.

2.11.2. Các bước thuật toán

Với $n = 7$ ta chọn $m = 3$.

Mảng a	5	7	2	4	6	1	3
Phân lớp	1	2	0	1	1	0	0

Sau đó ta tiến hành phân hoạch lại mảng a sao cho các phần tử về đúng phân lớp của nó

Mảng a	3	2	1	6	4	5	7
Phân lớp	0	0	0	1	1	1	2

Cuối cùng ta thực hiện Insertion Sort các đoạn con có cùng phân lớp.

2.11.3. Đánh giá độ phức tạp

Thời gian:

Bước phân hoạch ta chỉ cần duyệt qua các phần tử 1 lần. Do đó độ phức tạp sẽ là $O(n)$.

Trung bình thì mỗi phân lớp sẽ có $\frac{n}{m}$ phần tử. Do đó sắp xếp mỗi phân lớp trung sẽ là $O(\frac{n^2}{m^2})$ và có m phân lớp cần được sắp xếp do đó độ phức tạp là $O(\frac{n^2}{m^2} \cdot m)$

Với cách chọn $m = 0.43n$ thì ta thấy rằng độ phức tạp trong trường hợp tốt nhất và trung bình sẽ là $O(n)$.

Tuy nhiên với bộ dữ liệu phân bố không đều thì có thể có một vài phân lớp kích quá lớn so với phần còn lại sẽ dẫn đến trường hợp tệ nhất lên tới $O(n^2)$.

Không gian: $O(m)$

3. Thực nghiệm

3.1. Kết quả thực nghiệm

Thời gian thực thi được tính trên millisecond

3.1.1. Sorted data

Data order: Sorted data						
Data size	10000		30000		50000	
Result	Time	Comparison	Time	Comparison	Time	Comparison
Selection	101.760400	100010001	918.576400	900030001	2555.402900	2500050001
Insertion	0.000000	29998	0.000000	89998	1.000300	149998
Bubble	0.000000	20000	0.000000	60000	0.000000	100000
Shell	0.000000	360042	1.994200	1170050	1.996300	2100049
Shaker	0.000000	20001	0.000000	60001	1.000000	100001
Heap	1.994800	533373	5.996000	1781548	10.005600	3127047
Merge	1.031800	321628	4.022600	1052684	5.019000	1838364
Quick	1.030700	346280	3.027900	1219305	5.018100	1981438
Counting	0.000000	60003	0.000000	180003	1.030100	300003
Radix	0.000000	100054	2.017900	360067	3.016400	600067
Flash	0.000000	112804	1.024900	338404	2.027300	564004

Data order: Sorted data						
Data size	100000		300000		500000	
Result	Time	Comparison	Time	Comparison	Time	Comparison
Selection	10179.101200	10000100001	92676.114400	90000300001	256428.225900	250000500001
Insertion	1.000100	299998	0.996900	899998	1.994500	1499998
Bubble	0.000000	200000	0.000000	600000	0.997000	1000000
Shell	4.986800	4500051	15.965100	15300061	25.930700	25500058
Shaker	0.000000	200001	0.950300	600001	0.997300	1000001
Heap	20.975600	6664226	69.857900	21852070	115.690200	37799186
Merge	11.004300	3876732	38.916100	12570236	63.831200	21542140
Quick	11.003600	4311329	40.888900	17150903	70.764400	33489049
Counting	1.024200	600003	4.035000	1800003	5.976400	3000003
Radix	6.017900	1200067	21.937900	4200080	37.901400	7000080
Flash	5.021000	1128004	15.001100	3384004	23.938800	5640004

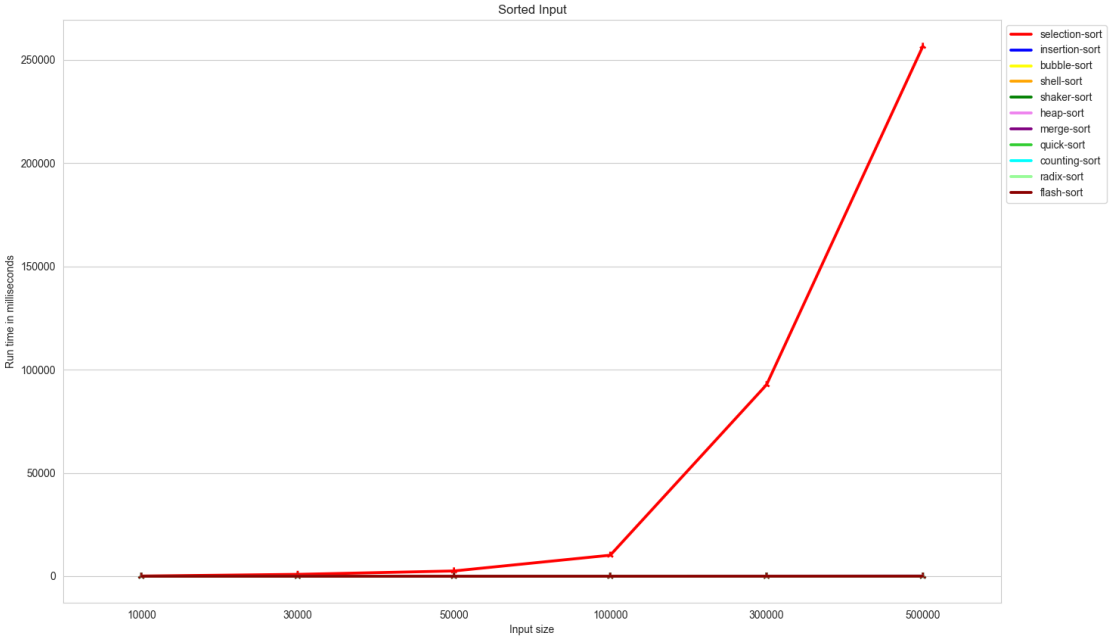

Figure 1: Biểu đồ đường thể hiện thời gian thực thi của các thuật toán sắp xếp

Tuy nhiên biểu đồ trên thời gian chạy của Selection Sort quá lớn nên ta không thể thấy được các thuật toán còn lại, do đó ta sẽ loại bỏ thuật toán này ra khỏi biểu đồ để dễ quan sát hơn

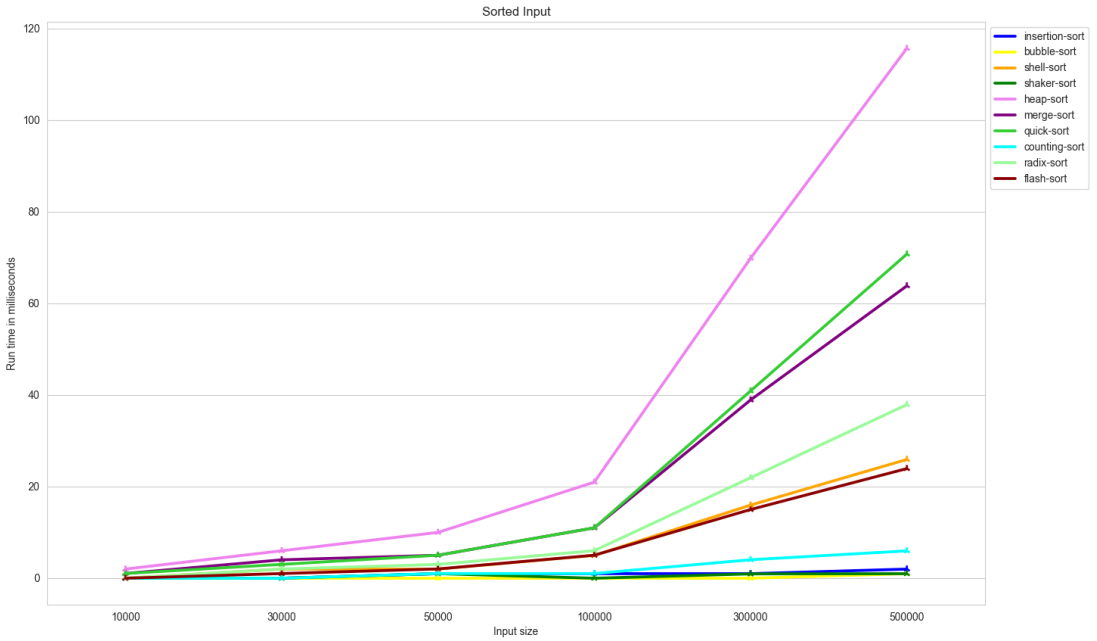

Figure 2: Biểu đồ đường thể hiện thời gian thực thi của các thuật toán sắp xếp (trừ Selection Sort)

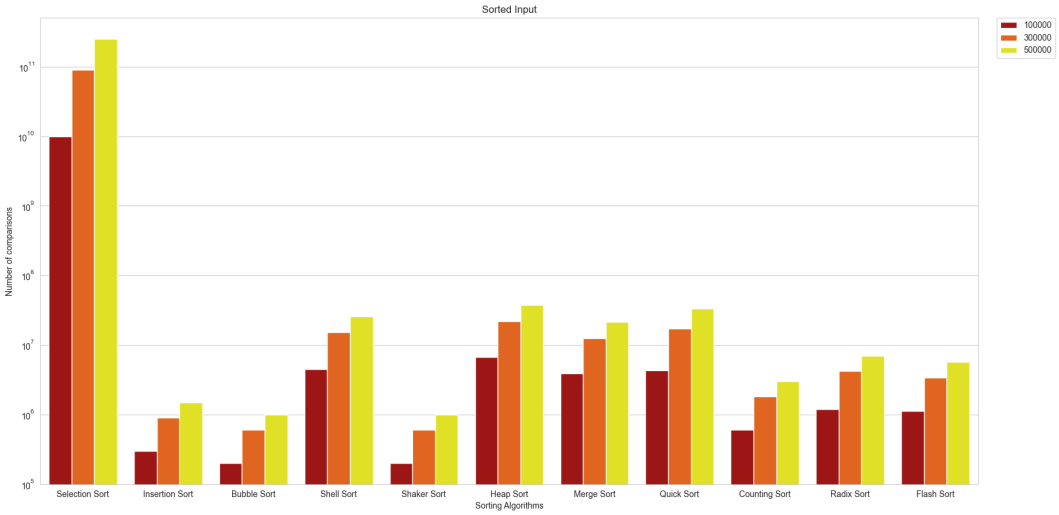

Figure 3: Biểu đồ cột thể hiện số bước so sánh của các thuật toán sắp xếp

3.1.2. Nearly sorted data

Data order: Nearly sorted data						
Data size	10000		30000		50000	
Result	Time	Comparison	Time	Comparison	Time	Comparison
Selection	101.729100	100010001	918.633400	900030001	2540.240000	2500050001
Insertion	0.000000	150602	0.000000	545502	1.014000	628054
Bubble	104.753000	95281550	880.679300	808760638	1995.695900	1841075430
Shell	1.000000	401176	2.034700	1280746	2.991500	2282915
Shaker	0.000000	219901	1.041100	779857	1.996000	1299857
Heap	2.023100	533008	5.036200	1781430	9.974800	3126859
Merge	2.023700	345483	3.039400	1131168	4.988100	1945072
Quick	1.025000	327316	3.037400	1232740	5.008900	2022064
Counting	0.971900	60003	0.000000	180003	0.000000	300003
Radix	0.997800	100054	2.991700	360067	4.004100	600067
Flash	0.997800	112807	0.988700	338399	2.018100	563997

Data order: Nearly sorted data						
Data size	100000		300000		500000	
Result	Time	Comparison	Time	Comparison	Time	Comparison
Selection	10231.119900	10000100001	92382.155700	90000300001	257027.954300	250000500001
Insertion	1.030200	761010	1.015000	1190354	2.027900	1923538
Bubble	4064.166700	3752383194	12436.337500	11472287788	23854.820900	22097855728
Shell	5.017800	4692152	16.998200	15449049	27.896700	25672095
Shaker	5.018400	2999805	8.974600	7799857	18.948000	16999745
Heap	20.984500	6664037	69.848100	21852098	116.696800	37799226
Merge	11.001100	3992784	36.917200	12668958	61.839400	21662449
Quick	11.995500	4282767	38.940400	17235880	71.806300	35143286
Counting	2.026200	600003	3.998800	1800003	5.987000	3000003
Radix	7.015900	1200067	23.935900	4200080	37.897700	7000080
Flash	5.019300	1128007	14.921400	3384002	25.930000	5639997

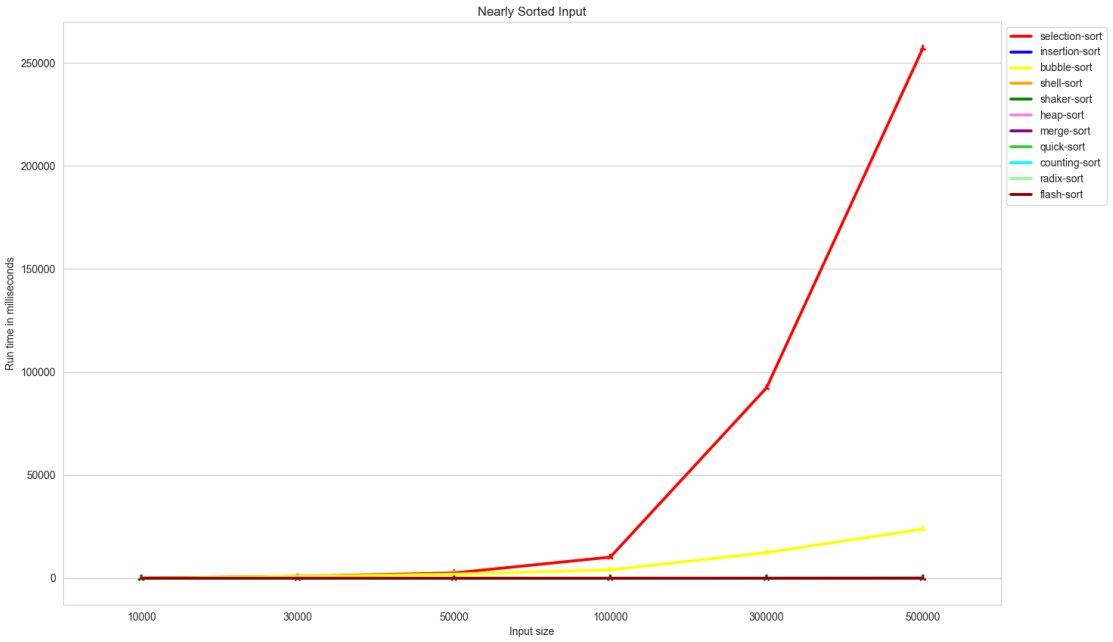

Figure 4: Biểu đồ đường thể hiện thời gian thực thi của các thuật toán sắp xếp

Giống như bộ dữ liệu được sắp thì ở đây ta chỉ thấy được đường của Selection Sort và Bubble Sort, do đó ta thử bỏ 2 thuật toán này ra để dễ dàng quan sát.

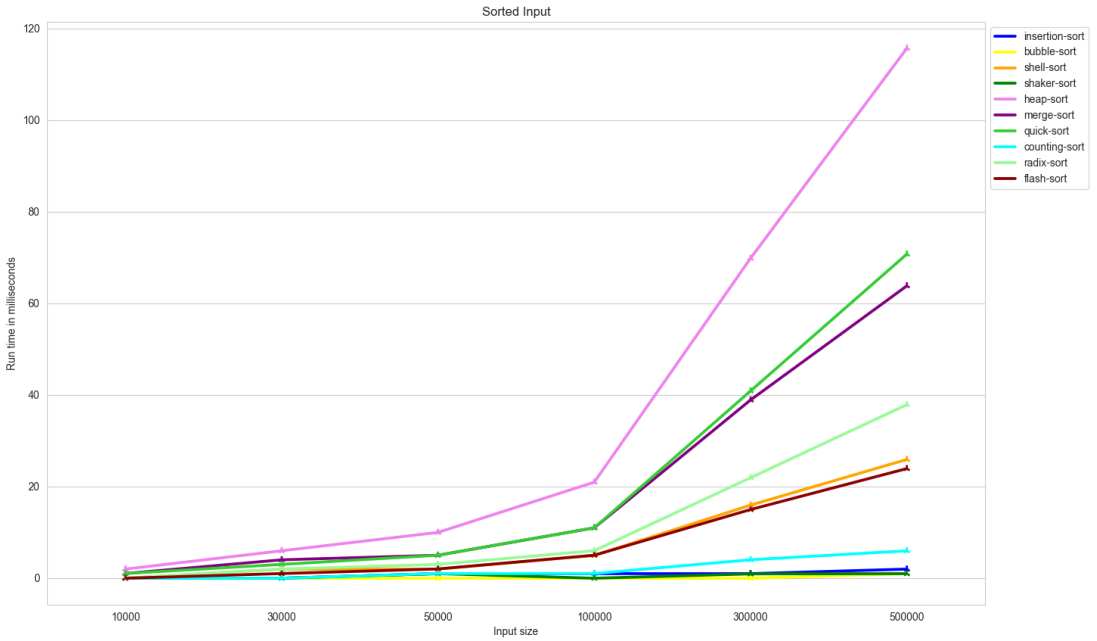

Figure 5: Biểu đồ đường thể hiện thời gian thực thi của các thuật toán sắp xếp (không có Selection Sort và Bubble Sort)

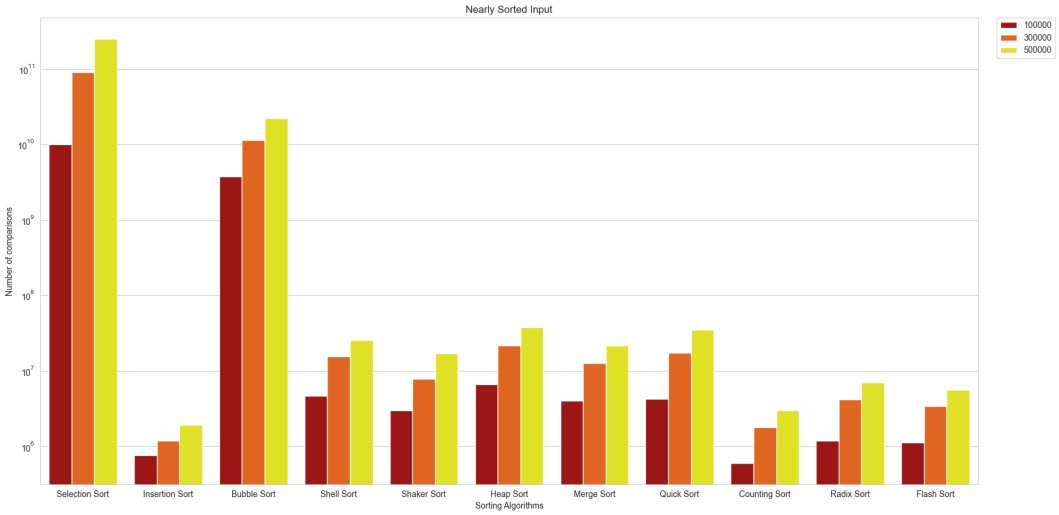

Figure 6: Biểu đồ cột thể hiện số bước so sánh của các thuật toán sắp xếp

3.1.3. Reverse sorted data

Data order: Reverse sorted data						
Data size	10000		30000		50000	
Result	Time	Comparison	Time	Comparison	Time	Comparison
Selection	98.781300	100010001	875.693500	900030001	2449.476400	2500050001
Insertion	116.720400	100009999	1042.259900	900029999	2884.335300	2500049999
Bubble	376.936500	100009999	3474.737700	900029999	9369.314400	2500049999
Shell	1.000000	475175	1.020800	1554051	3.024400	2844628
Shaker	355.050600	100019998	3212.445300	900059998	8956.292100	2500099998
Heap	1.032600	485075	5.011100	1649112	8.976600	2888832
Merge	1.042600	322827	4.030400	1066235	6.028800	1849483
Quick	1.963100	355887	3.990300	1118587	6.028300	1992785
Counting	0.000000	60003	0.000000	180003	0.000000	300003
Radix	0.000000	100054	2.021400	360067	3.031800	600067
Flash	0.000000	112806	1.993900	338406	2.033900	564006

Data order: Reverse sorted data						
Data size	100000		300000		500000	
Result	Time	Comparison	Time	Comparison	Time	Comparison
Selection	9739.807600	10000100001	87986.550400	90000300001	244436.608300	250000500001
Insertion	11549.059100	10000099999	103900.978300	90000299999	288641.209500	250000499999
Bubble	37424.488400	10000099999	337116.679700	90000299999	939710.971400	250000499999
Shell	6.016300	6089190	24.933600	20001852	38.903600	33857581
Shaker	35514.477600	10000199998	317620.435600	90000599998	890868.892700	250000999998
Heap	19.979200	6171509	65.856000	20443133	111.701400	35564898
Merge	12.001500	3898971	34.927900	12632603	62.771900	21860699
Quick	12.997800	4220223	47.867600	16108000	103.755100	33127904
Counting	0.998200	600003	3.990300	1800003	5.987600	3000003
Radix	6.016300	1200067	21.938600	4200080	36.921000	7000080
Flash	5.020300	1128006	13.960500	3384006	23.935800	5640006

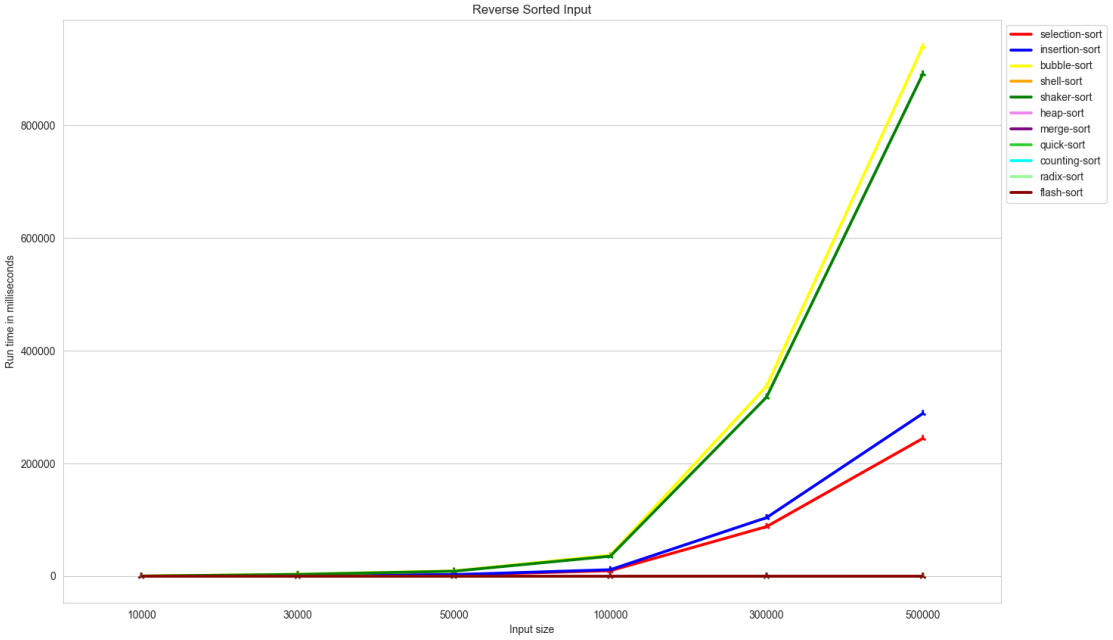

Figure 7: Biểu đồ đường thể hiện thời gian thực thi của các thuật toán sắp xếp

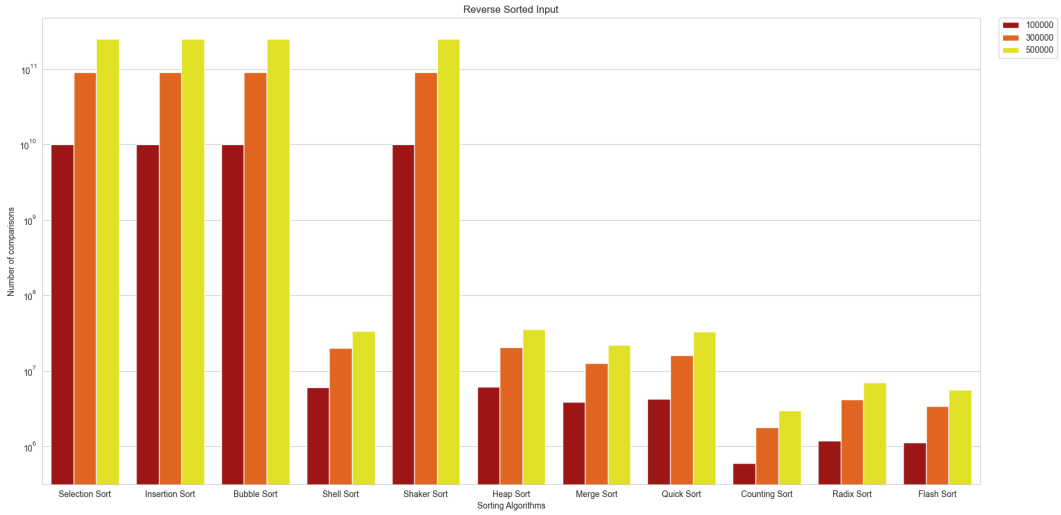

Figure 8: Biểu đồ cột thể hiện số bước so sánh của các thuật toán sắp xếp

3.1.4. Randomized data

Data order: Randomized data						
Data size	10000		30000		50000	
Result	Time	Comparison	Time	Comparison	Time	Comparison
Selection	103.719400	100010001	898.629100	900030001	2459.457100	2500050001
Insertion	54.886500	50165046	483.732500	447161338	1353.413400	1245524563
Bubble	338.145800	100004300	3145.736700	900007048	8788.454600	2499972438
Shell	2.034900	633789	5.020600	2397431	9.983200	4407748
Shaker	261.350500	75497697	2367.650800	677095377	6641.250700	1884415032
Heap	1.024000	508766	7.014900	1714805	11.974200	3010363
Merge	1.994700	430067	5.020600	1429766	8.937200	2498200
Quick	1.995600	338967	5.016700	1215532	8.024800	2043671
Counting	0.000000	60003	0.000000	179998	1.043600	282768
Radix	0.000000	100054	2.020700	360067	4.035100	600067
Flash	0.000000	117391	2.014300	354009	3.036100	580938

Data order: Randomized data						
Data size	100000		300000		500000	
Result	Time	Comparison	Time	Comparison	Time	Comparison
Selection	10037.747600	10000100001	90831.397900	90000300001	259809.299300	250000500001
Insertion	5497.162800	4997814737	51826.261300	44938528039	152202.701200	124910775413
Bubble	35808.714800	9999945158	337768.488700	90000215028	950523.529600	249999217444
Shell	22.970300	10202020	79.752100	34443328	136.633900	63041490
Shaker	27289.373400	7543213377	254889.736200	67503599901	698824.190500	187484998977
Heap	26.944300	6420023	92.751400	21142838	160.570400	36692804
Merge	18.982400	5297970	63.829100	17306327	105.718800	29907830
Quick	17.951300	4546739	62.832100	15236551	117.640000	29214941
Counting	0.998800	532770	3.004200	1532771	5.979300	2532771
Radix	5.983000	1200067	19.948900	3600067	31.916200	6000067
Flash	6.985600	1135259	21.906500	3588680	36.868500	6046648

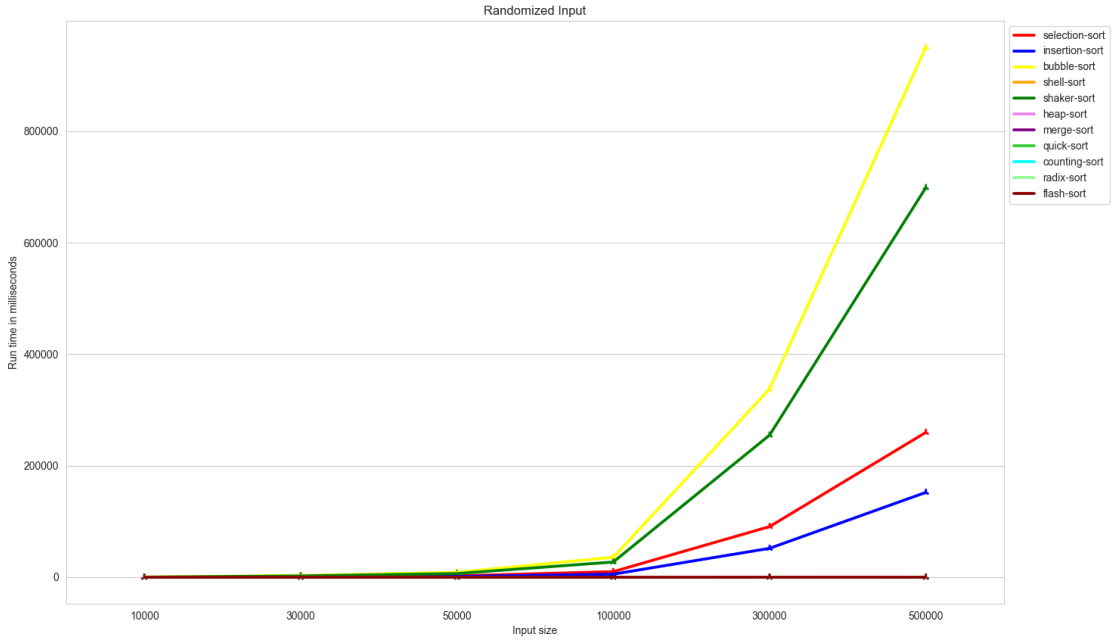

Figure 9: Biểu đồ đường thể hiện thời gian thực thi của các thuật toán sắp xếp

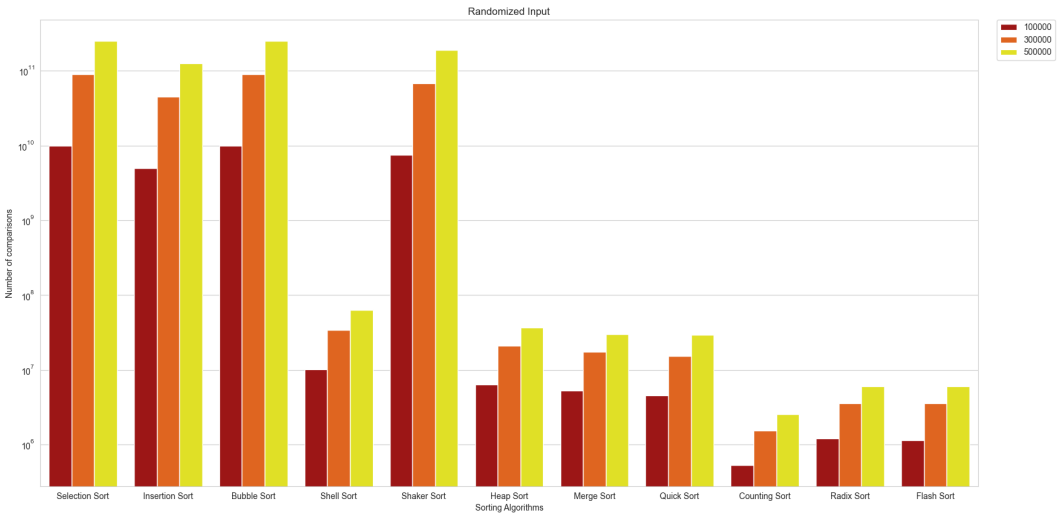

Figure 10: Biểu đồ cột thể hiện số bước so sánh của các thuật toán sắp xếp

3.2. Đánh giá

Dựa vào quá trình thực nghiệm cũng như việc thống kê ở trên, tôi có một vài quan điểm cá nhân về các giải thuật sắp xếp như sau.

Với bộ dữ liệu được sắp: Thì bản cải tiến của Bubble Sort cùng với Shaker Sort chạy rất nhanh vì gần như chỉ tốn 1 lần lặp. Và trong trường hợp này thì Selection Sort là giải thuật có hiệu quả thấp nhất. Còn các giải thuật còn lại thì cũng có sự chênh lệch như không đáng kể (chỉ khoảng 0.1s).

Với bộ dữ liệu gần như được sắp: Lần này ngoài Selection Sort ra thì có vẻ như Bubble Sort cũng không thực hiện hiệu quả lắm mặc dù đã được cải tiến. Và các giải thuật khác vẫn hoạt động khá hiệu quả trong bộ dữ liệu này.

Với bộ dữ liệu ngẫu nhiên và **bộ dữ liệu được sắp ngược** và các thuật toán $O(n^2)$ lúc này thể hiện rõ sự không hiệu quả của mình.

Tổng quan lại thì theo Selection Sort và Bubble Sort theo tôi là 2 thuật toán chạy chậm nhất, tuy nhiên thì Bubble Sort khá là dễ cài đặt.

Counting Sort và Flash Sort theo tôi nhận xét là chạy nhanh nhất, tuy nhiên 2 thuật toán này có một điểm chung là đều phụ thuộc vào sự phân bố của dữ liệu, do đó Heap Sort có vẻ sẽ được lựa chọn nhiều hơn.

Radix Sort cũng là một trong các thuật toán chạy rất nhanh, tuy nhiên nó chỉ hoạt động thực sự hiệu quả trên mảng số nguyên không âm.

Các thuật toán ổn định bao gồm: Selection Sort, Insertion Sort, Bubble Sort, Shaker Sort, Merge Sort, Radix Sort, Heap Sort.

Các thuật toán không ổn định bao gồm: Shell Sort, Quick Sort, Counting Sort, Flash Sort.

Có thể nhận thấy rằng các thuật toán sắp xếp thuộc $O(n \log n)$ và có tính ổn định cao thì không hiệu quả lắm đối với các bộ dữ liệu đã được sắp hoặc gần được sắp. Chẳng hạn như Merge Sort, mặc dù các đoạn đã được sắp rồi thế nhưng với Merge Sort thì nó vẫn chia đôi ra rồi lại trộn lại thành ra dẫn đến tốn chi phí thời gian cho các bước dư thừa.

4. Bài nộp

Bài nộp bao gồm folder **SOURCE** chứa các mã nguồn và file báo cáo (.pdf)

Trong thư mục **SOURCE** bao gồm:

- File main.cpp để thực thi chương trình cũng như xử lý các tham số hàm main.
- File Sort.cpp cũng như SortComp.cpp chứa các thuật toán sort, tuy nhiên file SortComp.cpp mục đích chính là đếm số phép so sánh
- File DataGenerator.cpp để sinh dữ liệu.
- Counter.h đếm số lượng phép so sánh.
- Checker.h kiểm tra xem mảng có được sort đúng hay không.

5. Tài liệu tham khảo

[1] https://www.geeksforgeeks.org/

[2] Introduction to Algorithms Third Edition