BÀI BÁO CÁO CÁC THUẬT TOÁN SẮP XẾP

Lê Duy Thức - 19120037

Mục lục

1 Một số lưu ý

Trong phần trình bày dưới đây, các thuật toán sắp xếp đều được dùng để sắp xếp các phần tử của một mảng theo thứ tự tăng dần.

1.1 Ký hiệu

Trong bài viết này tôi có sử dụng nhiều ký hiệu, dưới đây là một số ký hiệu mà các bạn nên nhớ:

- a: nếu không nói gì thêm thì a chính là mảng đầu vào.
- n: nếu không nói gì thêm thì n chính là độ dài của mảng đầu vào.
- $a[i]$: phần tử ở vị trí i của mảng a. Trong nội dung báo cáo này, mảng mặc định được đánh index 0, nghĩa là phần tử đầu tiên của mảng chính là $a[0]$ và phần tử cuối cùng là $a[n-1]$

1.2 Giới hạn

Tất cả các mảng đầu vào đều thoả giới hạn dưới đây:

- $n \leq 3 * 10^5$
- $0 \leq a[i] < n \; \forall i \in [1; n]$

1.3 Biểu đồ và bảng

Nếu không nói gì thêm, tất cả thời gian ở biểu đồ và bảng đều ở đơn vị milisecond.

1.4 Cấu hình máy tính và bộ dịch

Các thuật toán được chạy và đo với cấu hình máy tính sau đây:

- CPU: Intel i5-8300H @ 2.30GHz. Các thuật toán được chạy đơn luồng.
- Ram: 8192 MB
- OS: Windows 10 Pro 64-bit

Phần cài đặt, chương trình được biên dịch bằng trình biên dịch GNU C++ Compiler: g++ version 10.2.0 với lệnh biên dịch sau đây:

```
g++ main.cpp -std=c++17 -o main.exe
```

2 Các thuật toán đã cài đặt

2.1 Selection Sort

2.1.1 Ý tưởng thuật toán

Thuật toán Selection Sort (sắp xếp chọn) có ý tưởng khá đơn giản. Thuật toán sẽ phân dãy đầu vào thành 2 phần, một phần đã được sắp xếp (thường được đặt ở bên trái) và phần chưa được sắp xếp (thường được đặt ở bên phải). Thuật toán sẽ đi qua $n-1$ giai đoạn, mỗi giai đoạn thì thuật toán sẽ tìm phần tử bé nhất của phần **chưa được sắp xếp** và chuyển nó về phần **đã được sắp xếp**, sau đó tăng độ dài của phần được sắp xếp lên 1 (tương ứng, phần chưa được sắp xếp sẽ bị giảm đi 1).

2.1.2 Các bước hoạt động

Ví dụ ta có mảng đầu vào như sau:

$$a = \{4, 1, 0, 3, 2\}$$

Ở đây tôi dùng ký tự "|" để phân cách 2 phần đã sắp xếp và chưa sắp xếp của mảng. Phần tử có màu đỏ chính là phần tử nhỏ nhất của phần **chưa được sắp xếp**.

Dưới đây là các bước thực hiện thuật toán:

Giai đoạn	Mảng a	Giải thích
1	$\{\mid 4, 1, \mathbf{0}, 3, 2\}$	Ban đầu phần đã được sắp xếp không có gì, và phần tử bé nhất của phần chưa sắp là 0, ta sẽ đem phần tử 0 lên trước và tăng độ dài của phần được sắp xếp lên
2	$\{0 \mid 4, \mathbf{1}, 3, 2\}$	Bây giờ phần tử bé nhất của phần chưa sắp là 1, ta đem 1 lên phần đã được sắp xếp
3	$\{0, 1 \mid 4, 3, \mathbf{2}\}$	Bây giờ phần tử bé nhất của phần chưa sắp là 2, ta đem 1 lên phần đã được sắp xếp
4	$\{0, 1, 2 \mid 4, \mathbf{3}\}$	Bây giờ phần tử bé nhất của phần chưa sắp là 3, ta đem 1 lên phần đã được sắp xếp
5	$\{0, 1, 2, 3 \mid 4\}$	Tới đây thì thuật toán dừng, ta không cần đưa phần tử 4 lên trước vì nó đã ở đúng vị trí.

2.1.3 Phân tích thuật toán

2.1.3.1 Độ phức tạp thuật toán

Ta thấy, ở mỗi giai đoạn, ta cần lấy ra phần tử nhỏ nhất của phần chưa được sắp xếp, ban đầu phần này có n phần tử và ta cần $n-1$ phép so sánh, sau đó đưa phần tử này lên đầu thì phần chưa được sắp xếp chỉ còn $n-1$ phần tử, và cứ thế giảm dần về 1. Ta có số phép so sánh cần dùng là:

$$(n-1)+(n-2)+(n-3)+\cdots+2+1=\frac{n*(n-1)}{2}$$

Do đó, số phép so sánh cần thực hiện là $O(n^2)$. Mỗi giai đoạn ta cần một phép hoán vị, vì vậy ta cần $n-1$ phép hoán vị, hay $O(n)$ phép hoán vị.

Vậy độ phức tạp thời gian của thuật toán sắp xếp chọn là $O(n^2)$. Bộ nhớ cần dùng là $O(1)$.

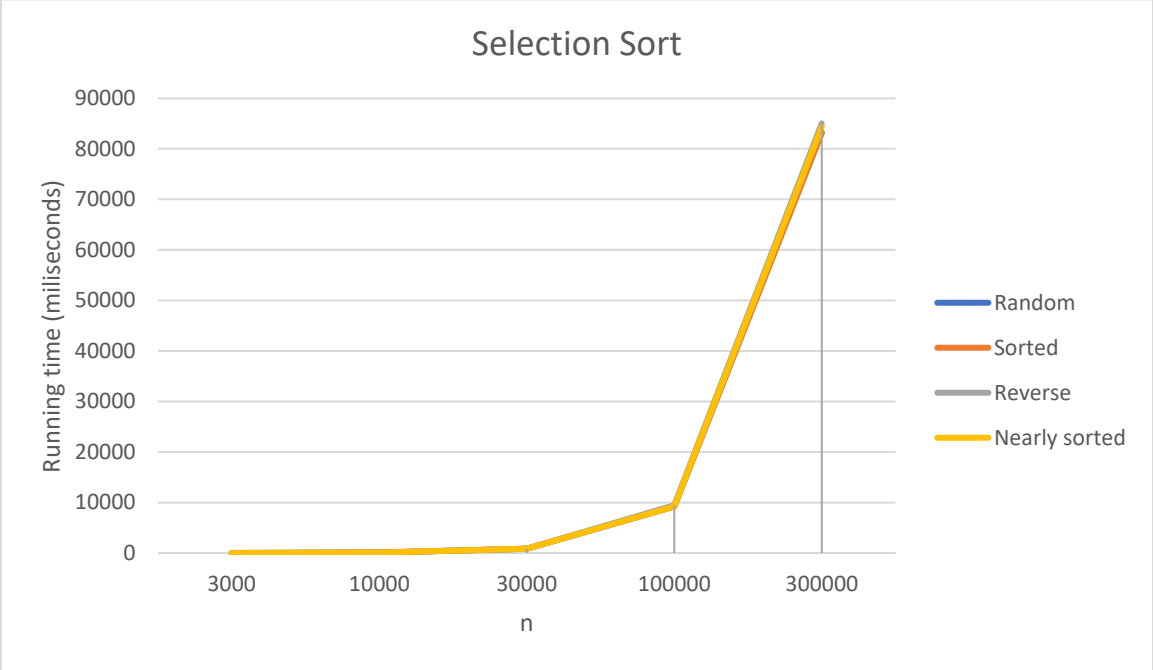
Thuật toán sắp xếp chọn hoàn toàn không phụ thuộc vào cách dữ liệu đầu vào, nghĩa là thời gian chạy không bị ảnh hưởng bởi cách dữ liệu được xếp. Do đó độ phức tạp trong các trường hợp tốt nhất, tệ nhất hay trung bình đều là $O(n^2)$

2.1.3.2 Thời gian chạy thực tế

Thời gian chạy với từng loại dữ liệu và n:

n	3000	10000	30000	100000	300000
Random	8.9588	92.7395	833.9812	9186.2415	83252.1522
Sorted	8.2168	91.3316	828.1996	9226.0128	83163.2184
Reverse	8.4407	94.9947	850.1531	9449.8258	85038.9836
Nearly sorted	8.7651	92.0205	827.0627	9235.1915	84352.3961

Biểu đồ tương ứng:



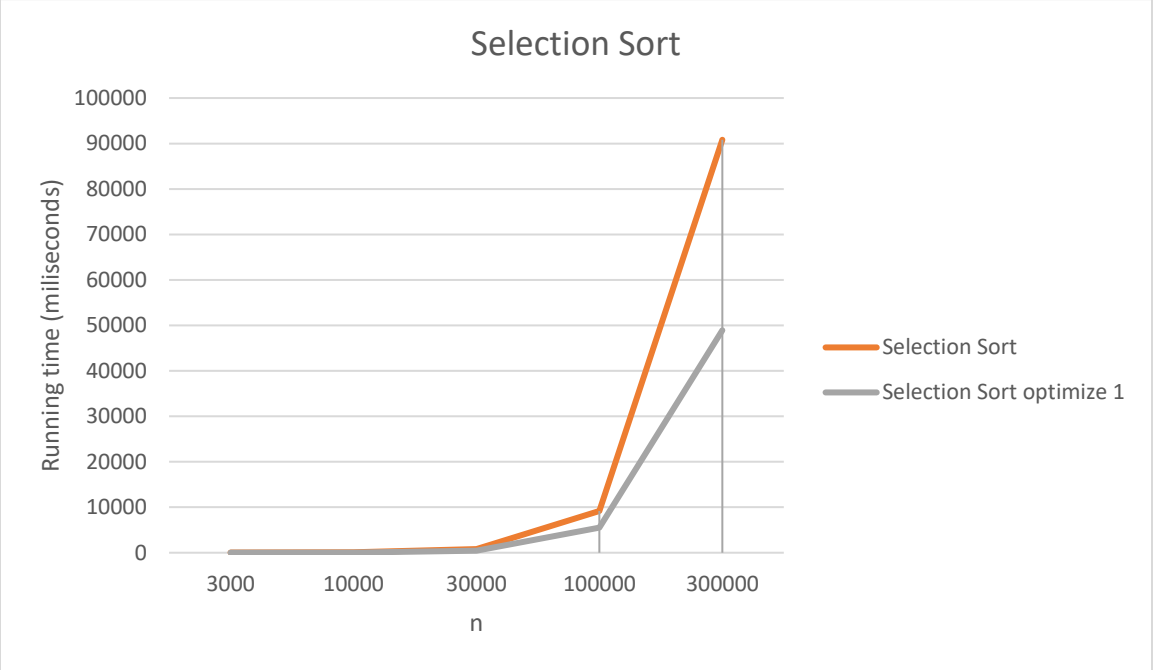
Ta có thể thấy, do thuật toán không phụ thuộc vào loại dữ liệu nên rằng thời gian chạy gần như là tương đương nhau.

2.1.4 Cải tiến

Một cách cải tiến thuật toán này chính là ở mỗi giai đoạn, thay vì ta chỉ đưa phần tử nhỏ nhất ra trước, ta có thể đưa thêm phần tử lớn nhất ra phía sau. Về chi tiết tôi sẽ không nói tới vì nó hoàn toàn giống ý tưởng ban đầu, với tối ưu này thì thời gian chạy trên **dữ liệu ngẫu nhiên** có thay đổi như sau:

n	3000	10000	30000	100000	300000
selection_sort	8.9588	92.7395	833.9812	9186.2415	83252.1522
selection_sort_optimize1	5.0227	54.8072	606.54	5483.5791	50352.2169

Biểu đồ tương ứng:



Mặc dù cải tiến này không giảm độ phức tạp của thuật toán nhưng ta có thể thấy rằng việc cải tiến này giúp giảm thời gian chạy đi khoảng $\frac{2}{5}$ cho tới $\frac{1}{2}$ so với thời gian chạy của sắp xếp chọn. Việc này hoàn toàn dễ hiểu vì bây giờ số giai đoạn của ta chỉ còn $\frac{n}{2}$ thay vì $n-1$ giai đoạn như ban đầu.

2.2 Insertion Sort

2.2.1 Ý tưởng thuật toán

Thuật toán sắp xếp chèn cũng phân chia mảng thành 2 phần giống Selection Sort, phần đã được sắp xếp và phần chưa được sắp xếp. Thuật toán cũng qua $n-1$ giai đoạn, ở mỗi giai đoạn, thuật toán sẽ lấy ra một phần tử ở phần chưa được sắp xếp và tìm vị trí "thích hợp" cho phần tử này ở phần đã được sắp xếp để chèn vào mà vẫn giữ được tính đã sắp xếp.

2.2.2 Các bước hoạt động

Ví dụ ta có mảng đầu vào như sau:

$$a = \{4, 1, 0, 3, 2\}$$

Ở đây tôi dùng ký tự "|" để phân cách 2 phần đã sắp xếp và chưa sắp xếp của mảng. Phần tử có màu đỏ chính là phần tử sẽ được chèn lên phần đã được sắp xếp. Vị trí có ký tự "*" chính là vị trí thích hợp cho phần tử này,

Dưới đây là các bước thực hiện thuật toán:

Giai đoạn	Mảng a	Giải thích
1	$\{*, 4 \mid \mathbf{1}, 0, 3, 2\}$	Với thuật toán sắp xếp chèn, phần đã được sắp xếp sẽ bắt đầu với 1 phần tử, và phần tử tiếp theo sẽ được thêm vào là phần tử 1, nó được chèn vào trước số 4.
2	$\{*, 1, 4 \mid \mathbf{0}, 3, 2\}$	Ở giai đoạn này, ta sẽ đưa phần tử 0 lên trước, và vị trí thích hợp là trước số 1.
3	$\{0, 1, *, 4 \mid \mathbf{3}, 2\}$	Ở giai đoạn này, ta sẽ đưa phần tử 3 lên trước, và vị trí thích hợp là trước số 4.
4	$\{0, 1, *, 3, 4 \mid \mathbf{2}\}$	Ở giai đoạn này, ta sẽ đưa phần tử 2 lên trước, và vị trí thích hợp là trước số 3.
	$\{0, 1, 2, 3, 4 \mid\}$	Tới đây thì thuật toán dừng.

2.2.3 Phân tích thuật toán

2.2.3.1 Độ phức tạp thuật toán

Để đánh giá độ phức tạp của thuật toán sắp xếp chèn, ta cần xem có bao nhiêu phép so sánh và phép gán được thực hiện.

Ta thấy, ở mỗi giai đoạn, ta tìm vị trí thích hợp cho một phần tử ở phần chưa được sắp xếp. Việc tìm vị trí thích hợp này cần nhiều nhất là $O(n)$ phép so sánh và gán (nếu vị trí thích hợp nằm ở đầu phần đã sắp), và ở trường hợp tốt nhất là $O(1)$ phép so sánh và gán (nếu vị trí thích hợp nằm ở cuối phần đã sắp).

Ở trường hợp tốt nhất, khi dãy đã được sắp xếp thì ta cần tổng cộng $O(n)$ phép so sánh và $O(n)$ phép gán. Vậy, độ phức tạp của sắp xếp chèn ở trường hợp tốt nhất là $O(n)$.

Ở trường hợp xấu nhất, khi dãy đã được sắp xếp giảm dần, thì mọi giai đoạn ta đều cần $O(n)$ phép gán và $O(n)$ phép so sánh, từ đó suy ra độ phức tạp ở trường hợp xấu nhất là $O(n^2)$

Trung bình thì độ phức tạp của thuật toán sắp xếp chèn là $O(n^2)$.

Thuật toán sắp xếp chèn phụ thuộc rất nhiều vào dữ liệu đầu vào, khi dữ liệu đầu vào đã được sắp xếp hoặc gần như được sắp xếp thì thuật toán sẽ chạy rất nhanh (Độ phức tạp $O(n)$) , nhưng nếu dữ liệu ngẫu nhiên hoặc dữ liệu được sắp xếp ngược thì thuật toán sẽ chạy rất lâu ví độ phức tạp ở trường hợp tệ nhất là $O(n^2)$

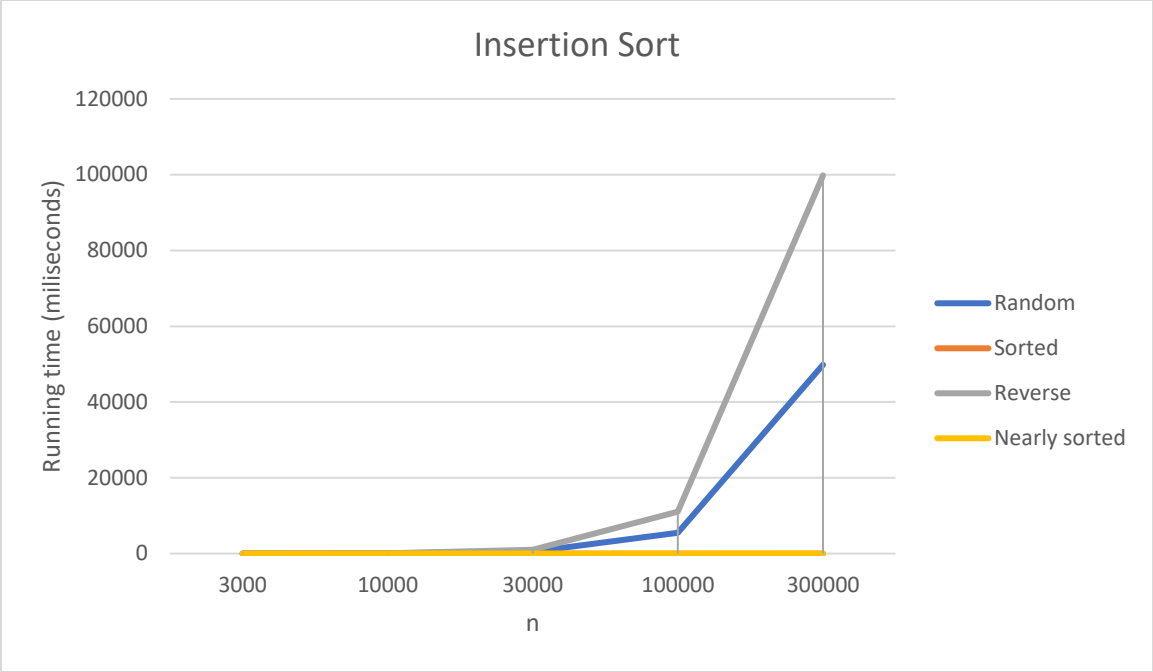
Độ phức tạp bộ nhớ của thuật toán là $O(1)$.

2.2.3.2 Thời gian chạy thực tế

Thời gian chạy với từng loại dữ liệu và n:

n	3000	10000	30000	100000	300000
Random	4.8686	54.9648	502.9288	5514.0076	49826.2318
Sorted	0.0127	0.0298	0.0892	0.3055	0.9078
Reverse	9.7241	109.9858	987.9799	11069.9714	99803.3046
Nearly sorted	0.0421	0.15	0.4991	0.6604	1.9701

Biểu đồ tương ứng:



Ta có thể thấy, do thuật toán phụ thuộc nhiều vào cách dữ liệu được sắp xếp trước nên thời gian chạy với từng bộ dữ liệu rất khác nhau. Khi n còn nhỏ thì ta không thấy quá nhiều sự khác biệt, nhưng khi n lớn, do độ phức tạp là $O(n^2)$ ở trường hợp xấu nhất (là mảng được sắp xếp ngược) nên đồ thị tăng rất nhanh. Còn ở trường hợp dữ liệu đã được sắp xếp hoặc gần sắp xếp thì thời gian chạy có thể nói là tuyến tính $O(n)$.

2.3 Binary-Insertion Sort

2.3.1 Ý tưởng thuật toán

Thuật toán sắp xếp chèn chị phân là một cải tiến trực tiếp từ thuật toán sắp xếp chèn.

Thay đổi lớn nhất của nó so với sắp xếp chèn truyền thống đó là thay vì duyệt hết mọi phần tử ở phần đã sắp xếp để tìm ta vị trí thích hợp, thì giờ đây là dùng thuật toán tìm kiếm nhị phân để làm việc này.

2.3.2 Các bước hoạt động

Ví dụ ta có mảng đầu vào như sau:

$$a = \{4, 1, 0, 3, 2\}$$

Ở đây tôi dùng ký tự "|" để phân cách 2 phần đã sắp xếp và chưa sắp xếp của mảng. Phần tử có màu đỏ chính là phần tử sẽ được chèn lên phần đã được sắp xếp. Vị trí có ký tự "*" chính là vị trí thích hợp cho phần tử này.

Dưới đây là các bước thực hiện thuật toán:

Giai đoạn	Mảng a	Giải thích
1	$\{*, 4 \mid \mathbf{1}, 0, 3, 2\}$	Với thuật toán sắp xếp chèn, phần đã được sắp xếp sẽ bắt đầu với 1 phần tử, và phần tử tiếp theo sẽ được thêm vào là phần tử 1, nó được chèn vào trước số 4, *vị trí này được tìm bằng tìm kiếm nhị phân.*
2	$\{*, 1, 4 \mid \mathbf{0}, 3, 2\}$	Ở giai đoạn này, ta sẽ đưa phần tử 0 lên trước, và vị trí thích hợp là trước số 1, *vị trí này được tìm bằng tìm kiếm nhị phân.*
3	$\{0, 1, *, 4 \mid \mathbf{3}, 2\}$	Ở giai đoạn này, ta sẽ đưa phần tử 3 lên trước, và vị trí thích hợp là trước số 4, *vị trí này được tìm bằng tìm kiếm nhị phân.*
4	$\{0, 1, *, 3, 4 \mid \mathbf{2}\}$	Ở giai đoạn này, ta sẽ đưa phần tử 2 lên trước, và vị trí thích hợp là trước số 3, *vị trí này được tìm bằng tìm kiếm nhị phân.*
	$\{0, 1, 2, 3, 4 \mid\}$	Tới đây thì thuật toán dừng.

2.3.3 Phân tích thuật toán

2.3.3.1 Độ phức tạp thuật toán

Để đánh giá độ phức tạp của thuật toán sắp xếp chèn nhị phân, ta cần xem có bao nhiêu phép so sánh và phép gán được thực hiện.

Ta thấy, ở mỗi giai đoạn, ta tìm vị trí thích hợp cho một phần tử ở phần chưa được sắp xếp bằng thuật toán tìm kiếm nhị phân. Việc tìm vị trí thích hợp này cần $O(logn)$ phép so sánh. Sau đó ta cần chèn phần tử vào đúng vị trí của nó, việc chèn này nếu ở trường hợp tốt nhất là $O(1)$ phép gán (nếu vị trí thích hợp nằm ở đầu phần đã sắp), còn trường hợp xấu nhất là $O(n)$ phép gán (nếu vị trí thích hợp nằm ở cuối phần đã sắp).

Ở trường hợp tốt nhất, khi dãy đã được sắp xếp thì ta cần tổng cộng $O(nlogn)$ phép so sánh do tìm kiếm nhị phân và $O(n)$ phép gán. Vậy, độ phức tạp của sắp xếp chèn ở trường hợp tốt nhất là $O(nlogn)$.

Ở trường hợp xấu nhất, khi dãy đã được sắp xếp giảm dần, thì mọi giai đoạn ta đều cần $O(n)$ phép gán và có $O(logn)$ phép so sánh từ chặt nhị phân, từ đó suy ra độ phức tạp ở trường hợp xấu nhất là $O(n^2 + nlogn) = O(n^2)$

Trung bình thì độ phức tạp của thuật toán sắp xếp chèn nhị phân là $O(n^2)$.

Giống như thuật toán sắp xếp chèn, thuật toán sắp xếp chèn nhị phân phụ thuộc rất nhiều vào dữ liệu đầu vào, khi dữ liệu đầu vào đã được sắp xếp hoặc gần như được sắp xếp thì thuật toán sẽ chạy rất nhanh (Độ phức tạp $O(nlogn)$) , nhưng nếu dữ liệu ngẫu nhiên hoặc dữ liệu được sắp xếp ngược thì thuật toán sẽ chạy rất lâu vì độ phức tạp ở trường hợp tệ nhất là $O(n^2)$

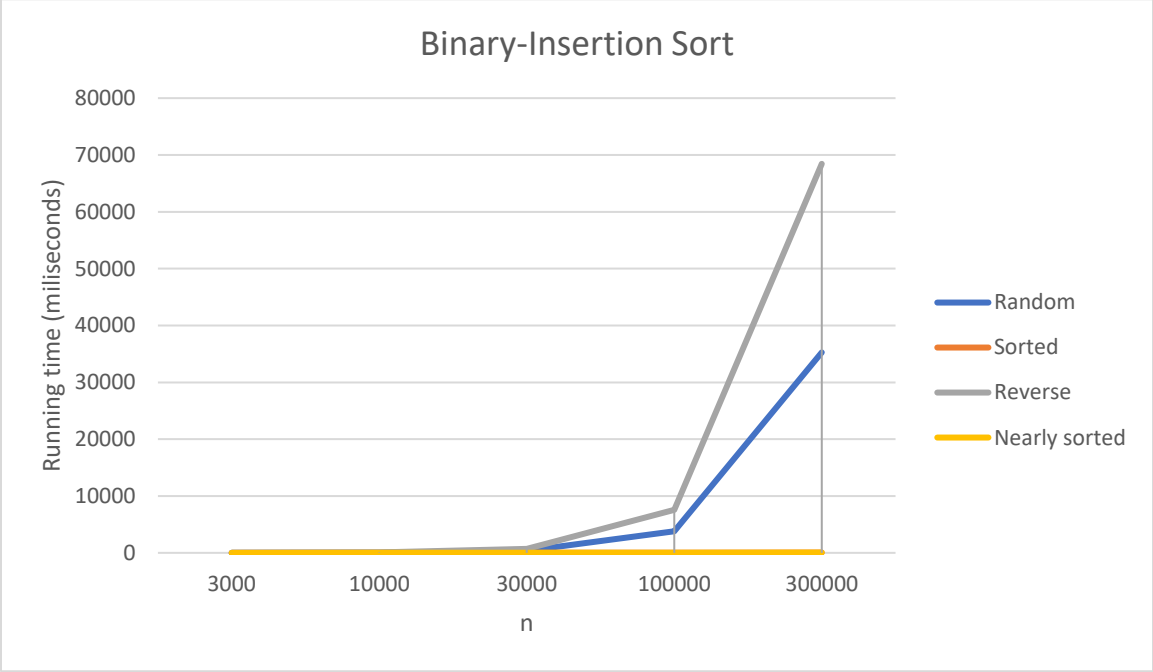
Độ phức tạp bộ nhớ của thuật toán là $O(1)$.

2.3.3.2 Thời gian chạy thực tế

Thời gian chạy với từng loại dữ liệu và n:

n	3000	10000	30000	100000	300000
Random	3.836	39.5626	350.8574	3790.6557	35262.0056
Sorted	0.3049	1.1979	3.9584	15.194	51.8683
Reverse	7.0634	75.7567	685.1251	7580.7619	68446.748
Nearly sorted	0.3737	1.3818	4.5556	15.7922	52.3756

Biểu đồ tương ứng:



Ta có thể thấy, biểu đồ này có dạng cũng gần giống như biểu đồ của thuật toán sắp xếp chèn. Khi n còn nhỏ thì ta không thấy quá nhiều sự khác biệt, nhưng khi n lớn, do độ phức tạp là $O(n^2)$ ở trường hợp xấu nhất (là mảng được sắp xếp ngược) nên đồ thị tăng rất nhanh. Còn ở trường hợp dữ liệu đã được sắp xếp hoặc gần sắp xếp thì thời gian chạy có thể nói là $O(nlogn)$.

Về so sánh thời gian chạy của 2 thuật toán, tôi sẽ trình bày ở phần sau.

2.4 Bubble Sort

2.4.1 Ý tưởng thuật toán

Thuật toán sắp xếp nổi bọn có ý tưởng đơn giản. Thuật toán này sẽ duyệt qua mảng $n-1$ lần, với mỗi lần duyệt, nếu thấy 2 phần tử kề nhau mà phần tử trước lớn hơn phần tử sau thì thuật toán sẽ hoán vị 2 phần tử đó cho nhau. Lưu ý là sau mỗi lần duyệt thì phần tử lớn nhất hiện tại sẽ được đưa về cuối mảng, và ta có thể không duyệt phần tử này vào lần sau, phần này giống như Selection Sort.

2.4.2 Các bước hoạt động

Ví dụ ta có mảng đầu vào như sau:

$$a = \{4, 1, 0, 3, 2\}$$

Ở đây, tôi đánh dấu phần tử có màu đỏ chính là hai phần tử đang được so sánh. Ký tự "|" để phân chia phần được đã sắp xếp và chưa được sắp xếp

Dưới đây là các bước thực hiện thuật toán:

Lần lặp	Mảng a	Giải thích
1	$\{4, 1, 0, 3, 2 \mid \}$	4 và 1 đang đứng sai vị trí, ta sẽ hoán vị 2 phần tử này
1	$\{1, 4, 0, 3, 2 \mid \}$	4 và 0 đang đứng sai vị trí, ta sẽ hoán vị 2 phần tử này
1	$\{1, 0, 4, 3, 2 \mid \}$	4 và 3 đang đứng sai vị trí, ta sẽ hoán vị 2 phần tử này
1	$\{1, 0, 3, 4, 2 \mid \}$	4 và 2 đang đứng sai vị trí, ta sẽ hoán vị 2 phần tử này
1	$\{1, 0, 3, 2, \mid 4\}$	Ta kết thúc lần lặp đầu tiên ở đây, 4 là phần tử lớn nhất đã được đưa về cuối dãy.
2	$\{1, 0, 3, 2, \mid 4\}$	1 và 0 đang đứng sai vị trí, ta sẽ hoán vị 2 phần tử này
2	$\{0, 1, 3, 2, \mid 4\}$	1 và 3 đã đứng đúng vị trí, ta không làm gì
2	$\{0, 1, 3, 2, \mid 4\}$	3 và 2 đang đứng sai vị trí, ta sẽ hoán vị 2 phần tử này
2	$\{0, 1, 2, \mid 3, 4\}$	Ta kết thúc vòng lặp thứ 2 ở đây
3	$\{0, 1, 2, \mid 3, 4\}$	0 và 1 đã đứng đúng vị trí, ta không làm gì
3	$\{0, 1, 2, \mid 3, 4\}$	1 và 2 đã đứng đúng vị trí, ta không làm gì
3	$\{0, 1, \mid 2, 3, 4\}$	Ta kết thúc vòng lặp thứ 3 ở đây
4	$\{0, 1, \mid 2, 3, 4\}$	0 và 1 đã đứng đúng vị trí, ta không làm gì
4	$\{ \mid 0, 1, 2, 3, 4\}$	Ta kết thúc vòng lặp thứ 4 ở đây

2.4.3 Phân tích thuật toán

2.4.3.1 Độ phức tạp thuật toán

Ta thấy, số phép so sánh cần thực hiện là $n-1$ ở lần lặp đầu tiên, sau đó là $n-2, n-3, \dots, 1$. Số phép hoán vị trong trường hợp xấu nhất sẽ bằng số phép so sánh, còn trong trường hợp tốt nhất là 0 phép hoán vị.

Trong trường hợp xấu nhất sẽ có $O(n^2)$ phép so sánh, $O(n^2)$ phép hoán vị. Suy ra độ phức tạp của thuật toán là $O(n^2)$ trong trường hợp xấu nhất.

Trong trường hợp tốt nhất thì ta cần $O(n^2)$ phép so sánh, $O(1)$ phép hoán vị. Suy ra độ phức tạp của thuật toán là $O(n^2)$ trong trường hợp xấu nhất.

Độ phức tạp của thuật toán là $O(n^2)$ trong trường hợp trung bình.

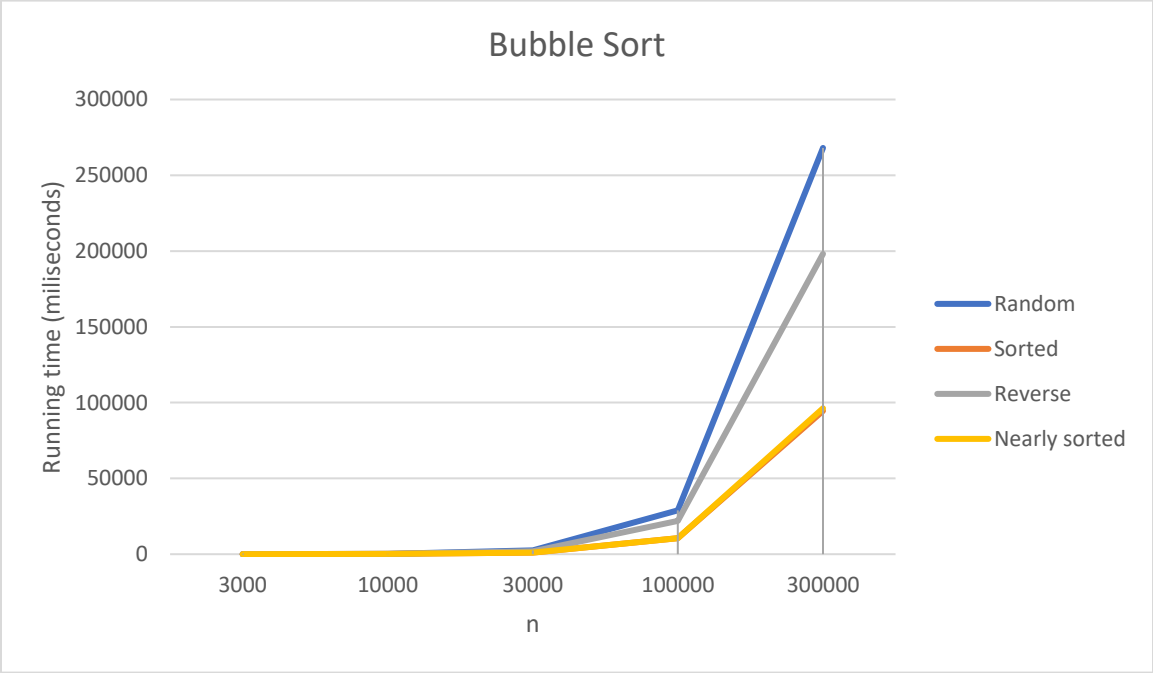
Thuật toán sắp xếp nổi bọt có độ phức tạp bộ nhớ là $O(1)$

2.4.3.2 Thời gian chạy thực tế

Thời gian chạy với từng loại dữ liệu và n:

n	3000	10000	30000	100000	300000
Random	17.6753	243.5058	2456.8087	28943.7081	268048.1359
Sorted	9.3751	103.5523	945.4694	10498.0885	94556.2277
Reverse	19.3244	216.2287	1971.2866	22046.0146	198140.0733
Nearly sorted	9.4183	104.0736	945.5142	10480.3164	96246.7829

Biểu đồ tương ứng:



Khi dữ liệu đã được sắp xếp hoặc gần như được sắp xếp thì số bước hoán vị sẽ thấp (mặc dù số bước so sánh vẫn là $O(n^2)$) cho nên ta thấy thời gian sẽ nhanh gấp đôi so với khi mảng đã được sắp xếp ngược.

Có một điều lạ ở đây đó là khi mảng được sắp xếp ngẫu nhiên thì thời gian chạy lâu hơn so với khi mảng được sắp xếp ngược. Điều này không đúng với lý thuyết, vì theo lý thuyết thì khi mảng bị sắp xếp ngược thì số lần hoán vị sẽ là lớn nhất. Theo suy đoán của tôi, lý do mảng ngẫu nhiên chạy chậm hơn mảng đã được sắp xếp ngược có thể là do tối ưu của CPU, mà ở đây là do **Branch Prediction**[1]**.** Tôi sẽ không đi sâu hơn về Branch Prediction vì nó không liên quan tới chủ đề bài báo cáo nhưng ta có thể tạm suy đoán rằng khi mảng được sắp xếp ngược thì các câu lệnh rẻ nhánh sẽ dễ "dự đoán" hơn là khi mảng được sắp xếp ngẫu nhiên.

2.4.4 Cải tiến

2.4.4.1 Cải tiến 1

Một cách cải tiến thuật toán này chính là nhờ nhận xét ta không cần duyệt qua mảng đúng $n - 1$ lần, mà ta sẽ ngừng việc duyệt ngay khi mảng đã được sắp xếp.

Dưới đây là thời gian chạy của thuật toán cải tiến này với:

Bộ dữ liệu ngẫu nhiên

n	3000	10000	30000	100000	300000
Bubble Sort	17.6753	243.5058	2456.809	28943.7081	268048.1
Bubble Sort optimize 1	18.8777	253.1162	2497.213	29417.849	269035.6

Bộ dữ liệu đã được sắp xếp

n	3000	10000	30000	100000	300000
Bubble Sort	9.3751	103.5523	945.4694	10498.09	94556.23
Bubble Sort optimize 1	0.0063	0.021	0.0623	0.2356	0.6369

Bộ dữ liệu được sắp xếp ngược

n	3000	10000	30000	100000	300000
Bubble Sort	19.3244	216.2287	1971.2866	22046.01	198140.1
Bubble Sort optimize 1	20.0365	223.4281	2043.349	22671.4783	204460.8

Bộ dữ liệu gần như được sắp xếp

n	3000	10000	30000	100000	300000
Bubble Sort	9.4183	104.0736	945.5142	10480.32	96246.78
Bubble Sort optimize 1	8.0653	83.9896	914.4556	3343.7823	17548.45

[1]Parihar, Raj: **"Branch Prediction Techniques and Optimizations"**

Ta thấy, với cải tiến này thì kết quả không thay đổi quá nhiều, tập chí còn tệ hơn nếu mảng được sắp xếp ngược hoặc là ngẫu nhiên. Còn khi mảng được sắp xếp hoặc gần như được sắp xếp thì thời gian nhanh lên đáng kể, vì lúc này số lần lặp lại mảng sẽ rất ít.

Với cải tiến này, độ phức tạp ở trường hợp tốt nhất sẽ là $O(n)$ khi mảng đã được sắp xếp tăng dần.

2.4.4.2 Cải tiến 2

Một cải tiến rất nổi tiếng của thuật toán sắp xếp nổi bọt là Shaker Sort mà tôi sẽ trình bày ngay sau đây.

2.5 Shaker Sort

2.5.1 Ý tưởng thuật toán

Shaker sort là một cải tiến từ bubble sort. Ta thấy, ở thuật toán sắp xếp nổi bọt, mỗi khi sắp xếp thì ta chỉ đưa các phần tử hướng về phía sau. Shaker sort thì sau khi duyệt từ đầu mảng tới cuối, ta sẽ thực hiện một lần duyệt từ cuối về đầu mảng, với cách làm này thì các phần tử có thể được di chuyển theo hướng đi về trước.

Sau mỗi lần lặp như thế, phần tử lớn nhất sẽ được đưa về cuối, phần tử nhỏ nhất sẽ được đầu, vì vậy ta cũng không cần duyệt hết lại từ đầu mảng, một nhận xét rất giống phần cải tiến của thuật toán sắp xếp chọn mà tôi đã trình bày ở trên.

2.5.2 Các bước hoạt động

Ví dụ ta có mảng đầu vào như sau:

$$a = \{4, 1, 0, 3, 2\}$$

Ở đây, tôi đánh dấu phần tử có màu đỏ chính là hai phần tử đang được so sánh. Tôi dùng ký tự "|" để phân cách phần đã sắp xếp và chưa sắp xếp của mảng, sẽ có 2 ký tự "|" vì sẽ có 2 phần đã sắp xếp (là phần đầu mảng và cuối mảng).

Dưới đây là các bước thực hiện thuật toán:

Lần lặp	Mảng a	Giải thích
1	$\{ \|4, 1, 0, 3, 2 \| \}$	4 và 1 đang đứng sai vị trí, ta sẽ hoán vị 2 phần tử này
1	$\{ \| 1, 4, 0, 3, 2 \| \}$	4 và 0 đang đứng sai vị trí, ta sẽ hoán vị 2 phần tử này
1	$\{ \| 1, 0, 4, 3, 2 \| \}$	4 và 3 đang đứng sai vị trí, ta sẽ hoán vị 2 phần tử này
1	$\{ \| 1, 0, 3, 4, 2 \| \}$	4 và 2 đang đứng sai vị trí, ta sẽ hoán vị 2 phần tử này
1	$\{ \| 1, 0, 3, 2, \| 4 \}$	Ta hoàn thành việc duyệt xuôi từ đầu về cuối, lúc này ta thêm 4 vào phần đã sắp xếp ở sau. Và ta bắt đầu duyệt theo thứ tự ngược lại
1	$\{ \| 1, 0, 3, 2 \| 4 \}$	3 và 2 đang đứng sai vị trí, ta sẽ hoán vị 2 phần tử này

1	$\{ \mid 1, 0, 2, 3 \mid 4 \}$	0 và 2 đã đứng đúng vị trí, ta không làm gì.
1	$\{ \mid 1, 0, 2, 3 \mid 4 \}$	1 và 0 đang đứng sai vị trí, ta sẽ hoán vị 2 phần tử này
1	$\{ 0 \mid 1, 2, 3 \mid 4 \}$	Ta hoàn thành việc duyệt ngược từ cuối về đầu, lúc này ta thêm 0 vào phần đã sắp xếp ở sau. Ta kết thúc lần lặp 1
2	$\{ 0 \mid 1, 2, 3 \mid 4 \}$	1 và 2 đã đứng đúng vị trí, ta không làm gì.
2	$\{ 0 \mid 1, 2, 3 \mid 4 \}$	2 và 3 đã đứng đúng vị trí, ta không làm gì.
2	$\{ 0 \mid 1, 2 \mid 3, 4 \}$	Ta hoàn thành việc duyệt xuôi từ đầu về cuối, lúc này ta thêm 3 vào phần đã sắp xếp ở sau. Và ta bắt đầu duyệt theo thứ tự ngược lại
2	$\{ 0 \mid 1, 2 \mid 3, 4 \}$	1 và 2 đã đứng đúng vị trí, ta không làm gì.
2	$\{ 0, 1 \mid 2 \mid 3, 4 \}$	Ta hoàn thành việc duyệt ngược từ cuối về đầu, lúc này ta thêm 1 vào phần đã sắp xếp ở sau. Ta kết thúc lần lặp 2
3	$\{ 0, 1 \mid \mid 2, 3, 4 \}$	Ta hoàn thành việc duyệt xuôi từ đầu về cuối, lúc này ta thêm 3 vào phần đã sắp xếp ở sau. Lúc này ta ngừng thuật toán vì mảng đã được sắp xếp.

2.5.3 Phân tích thuật toán

2.5.3.1 Độ phức tạp thuật toán

Độ phức tạp của thuật toán Shaker sort hoàn toàn giống với Bubble sort, chỉ khác duy nhất là thời gian chạy sẽ nhanh hơn vì số bước thực hiện sẽ ít hơn, nhưng nhìn chung, khi nói về độ phức tạp thì nó vẫn là $O(n^2)$ trong mọi trường hợp.

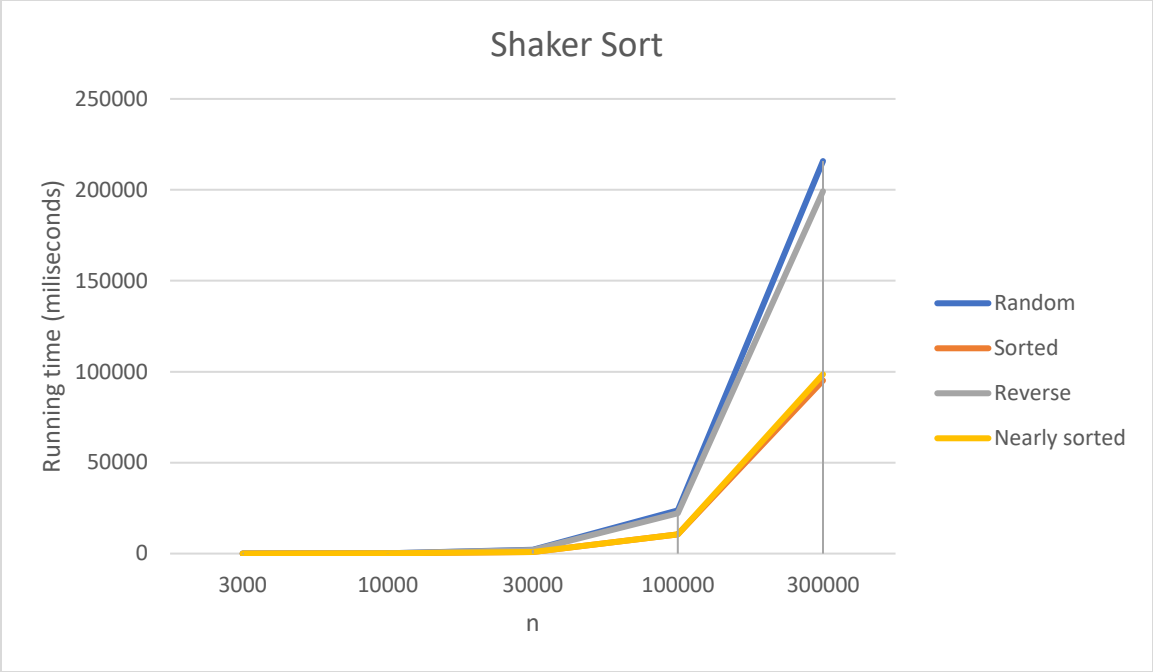
Thuật toán Shaker Sort có độ phức tạp bộ nhớ là $O(1)$

2.5.3.2 Thời gian chạy thực tế

Thời gian chạy với từng loại dữ liệu và n:

n	3000	10000	30000	100000	300000
Random	18.1968	223.8228	2084.4841	23692.7625	215826.0308
Sorted	9.4021	104.4242	955.0801	10564.075	95155.2853
Reverse	19.7438	219.491	1985.3185	22099.0394	199228.314
Nearly sorted	9.4238	107.5545	949.2447	10559.4304	98573.5853

Biểu đồ tương ứng:



Cũng giống với sắp xếp nổi bọt, khi dữ liệu đã được sắp xếp hoặc gần như được sắp xếp thì số bước hoán vị sẽ thấp (mặc dù số bước so sánh vẫn là $O(n^2)$) cho nên ta thấy thời gian sẽ nhanh gấp đôi so với khi mảng đã được sắp xếp ngược.

Cùng một vấn đề với sắp xếp nổi bọt, ta thấy khi mảng được sắp xếp ngẫu nhiên thì thời gian chạy lâu hơn so với khi mảng được sắp xếp ngược. Tôi nghĩ lý do cũng tương tự, đó là do Branch Prediction.

2.5.4 Cải tiến

2.5.4.1 Cải tiến 1

Cũng như sắp xếp nổi bọt, ta có một cách cải tiến thuật toán này chính là ta sẽ ngừng việc duyệt ngay khi mảng đã được sắp xếp.

Dưới đây là thời gian chạy của thuật toán cải tiến này:

Bộ dữ liệu ngẫu nhiên

n	3000	10000	30000	100000	300000
Shaker Sort	18.1968	223.8228	2084.484	23692.7625	215826
Shaker Sort optimize 1	18.7474	210.39	1872.808	21868.2865	194324.5

Bộ dữ liệu đã được sắp xếp

n	3000	10000	30000	100000	300000
Shaker Sort	9.4021	104.4242	955.0801	10564.075	95155.29
Shaker Sort optimize 1	0.0064	0.0205	0.0629	0.2106	0.6347

Bộ dữ liệu được sắp xếp ngược

n	3000	10000	30000	100000	300000
Shaker Sort	19.7438	219.491	1985.319	22099.0394	199228.3
Shaker Sort optimize 1	20.8997	235.0072	2057.075	22803.776	205858.9

Bộ dữ liệu gần như được sắp xếp

n	3000	10000	30000	100000	300000
Shaker Sort	9.4238	107.5545	949.2447	10559.4304	98573.59
Shaker Sort optimize 1	0.1042	0.4015	1.3023	3.228	9.1263

Giống với cải tiến ở thuật toán sắp xếp nổi bọt, cải tiến này chỉ tối ưu hơn khi mảng đã được sắp xếp hoặc là gần như sắp xếp. Ngược lại, khi mảng ngẫu nhiên hoặc được sắp xếp ngược thì cải tiến này sẽ làm chương trình chạy lâu hơn, dù không đáng kể.

Với cải tiến này, độ phức tạp ở trường hợp tốt nhất sẽ là $O(n)$ khi mảng đã được sắp xếp tăng dần.

2.5.5 So sánh với Bubble Sort

Dưới đây tôi sẽ so sánh Cải tiến 1 của Bubble Sort với Cải tiến 1 của Shaker Sort qua từng bộ dữ liệu:

Bộ dữ liệu ngẫu nhiên

n	3000	10000	30000	100000	300000
Bubble Sort optimize 1	18.8777	253.1162	2497.213	29417.849	269035.6
Shaker Sort optimize 1	18.7474	210.39	1872.808	21868.2865	194324.5

Bộ dữ liệu đã được sắp xếp

n	3000	10000	30000	100000	300000
Bubble Sort optimize 1	0.0063	0.021	0.0623	0.2356	0.6369
Shaker Sort optimize 1	0.0064	0.0205	0.0629	0.2106	0.6347

Bộ dữ liệu được sắp xếp ngược

n	3000	10000	30000	100000	300000
Bubble Sort optimize 1	20.0365	223.4281	2043.349	22671.4783	204460.8
Shaker Sort optimize 1	20.8997	235.0072	2057.075	22803.776	205858.9

Bộ dữ liệu gần như được sắp xếp

n	3000	10000	30000	100000	300000
Bubble Sort optimize 1	8.0653	83.9896	914.4556	3343.7823	17548.45
Shaker Sort optimize 1	0.1042	0.4015	1.3023	3.228	9.1263

Với đại đa số bộ dữ liệu thì thuật toán Shaker sort sẽ chạy nhanh hơn Bubble Sort, chỉ trừ khi bộ dữ liệu được sắp xếp ngược, nhưng chênh lệch không quá đáng kể. Một điều đáng chú ý là khi bộ dữ liệu gần như được sắp xếp thì thuật toán Shaker Sort chạy nhanh hơn hẳn so với Bubble Sort.

Điều này khá dễ hiểu, ta có thể lấy mảng $a = \{2, 3, 4, 5, 1\}$ làm ví dụ. Ở mảng này thì thuật toán Bubble Sort cần 4 lần duyệt mảng, trong khi thuật toán Shaker Sort chỉ cần 1 lần duyệt xuôi và 1 lần duyệt ngược là mảng đã được sắp xếp.

2.6 Shell Sort

2.6.1 Ý tưởng thuật toán

Thuật toán Shell Sort có thể nói là một cải tiến của thuật toán Insertion Sort, thuật toán này cho phép so sánh và hoán vị các phần tử nằm xa nhau thay vì chỉ được xét những phần tử kề nhau như thuật toán Insertion Sort. Ban đầu, ta sẽ có một dãy các số gọi là các "***gaps***", ta lần lượt duyệt qua các ***gaps*** này theo thứ tự giảm dần và ta sẽ chỉ so sánh các phần tử cách nhau một đoạn đúng bằng ***gap*** tương ứng.

2.6.2 Các bước hoạt động

Ví dụ ta có mảng đầu vào như sau:

$$a = \{4, 1, 0, 3, 2\}$$

Và có mảng các ***gaps*** là:

$$g = \{3, 2, 1\}$$

Phần tử có màu đỏ chính là phần tử đang được so sánh.

Dưới đây là các bước thực hiện thuật toán:

Gap	Mảng a	Giải thích
3	$\{\mathbf{4}, 1, 0, \mathbf{3}, 2\}$	Bắt đầu với ***gap=3,*** ta so sánh phần tử 4 và 3, ta thấy phần tử 3 đang đứng sai vị trí nên sẽ được đưa lên trước.
3	$\{3, \mathbf{1}, 0, 4, \mathbf{2}\}$	Với ***gap=3,*** ta so sánh phần tử 1 và 2, ta thấy phần tử 2 đã đứng đúng vị trí nên ta không thực hiện gì. Tới đây ta cũng kết thúc việc duyệt mảng với ***gap=3.***
2	$\{\mathbf{3}, 1, \mathbf{0}, 4, 2\}$	Với ***gap=2,*** ta so sánh phần tử 3 và 0, ta thấy phần tử 0 đang đứng sai vị trí nên sẽ được đưa lên trước.

2	$\{0, \mathbf{1}, 3, \mathbf{4}, 2\}$	Với ***gap=2,*** ta so sánh phần tử 3 và 4, ta thấy phần tử 4 đã đứng đúng vị trí nên ta không thực hiện gì.
2	$\{0, 1, \mathbf{3}, 4, \mathbf{2}\}$	Với ***gap=2,*** ta so sánh phần tử 3 và 2, ta thấy phần tử 2 đang đứng sai vị trí nên sẽ được đưa lên trước.
2	$\{\mathbf{0}, 1, \mathbf{2}, 4, 3\}$	Ta tiếp tục so sánh 0 và 2, , ta thấy phần tử 2 đã đứng đúng vị trí nên ta không thực hiện gì. Tới đây ta cũng kết thúc việc duyệt mảng với ***gap=2.***
1	$\{\mathbf{0}, \mathbf{1}, 2, 4, 3\}$	Với ***gap=1,*** ta so sánh phần tử 0 và 1, ta thấy phần tử 1 đã đứng đúng vị trí nên ta không thực hiện gì.
1	$\{0, \mathbf{1}, \mathbf{2}, 4, 3\}$	Với ***gap=1,*** ta so sánh phần tử 1 và 2, ta thấy phần tử 2 đã đứng đúng vị trí nên ta không thực hiện gì.
1	$\{0, 1, \mathbf{2}, \mathbf{4}, 3\}$	Với ***gap=1,*** ta so sánh phần tử 2 và 4, ta thấy phần tử 4 đã đứng đúng vị trí nên ta không thực hiện gì.
1	$\{0, 1, 2, \mathbf{4}, \mathbf{3}\}$	Với ***gap=1,*** ta so sánh phần tử 4 và 3, ta thấy phần tử 3 đang đứng sai vị trí nên sẽ được đưa lên trước.
1	$\{0, 1, \mathbf{2}, \mathbf{3}, 4\}$	Ta tiếp tục so sánh 2 và 3 , ta thấy phần tử 3 đã đứng đúng vị trí nên ta không thực hiện gì. Tới đây ta cũng kết thúc việc duyệt mảng với ***gap=1*** và kết thúc thuật toán.

2.6.3 Phân tích thuật toán

2.6.3.1 Độ phức tạp thuật toán

Độ phức tạp của thuật toán Shell Sort phụ thuộc phần lớn vào cách ta chọn dãy ***gaps***. Với một số dãy, việc đánh giá độ phức tạp của Shell Sort vẫn là một bài toán mở, chưa có lời giải [1].

Bộ nhớ cần dùng của thuật toán là $O(1 + m)$, với m là độ dài dãy ***gaps***.

Trong phần cài đặt thuật toán này, tôi chọn dãy ***gaps*** có dạng:

$$g = \{1750, 701, 301, 132, 57, 23, 10, 4, 1\}$$

Dãy này được đề xuất bởi Ciura vào năm 2001 [1].

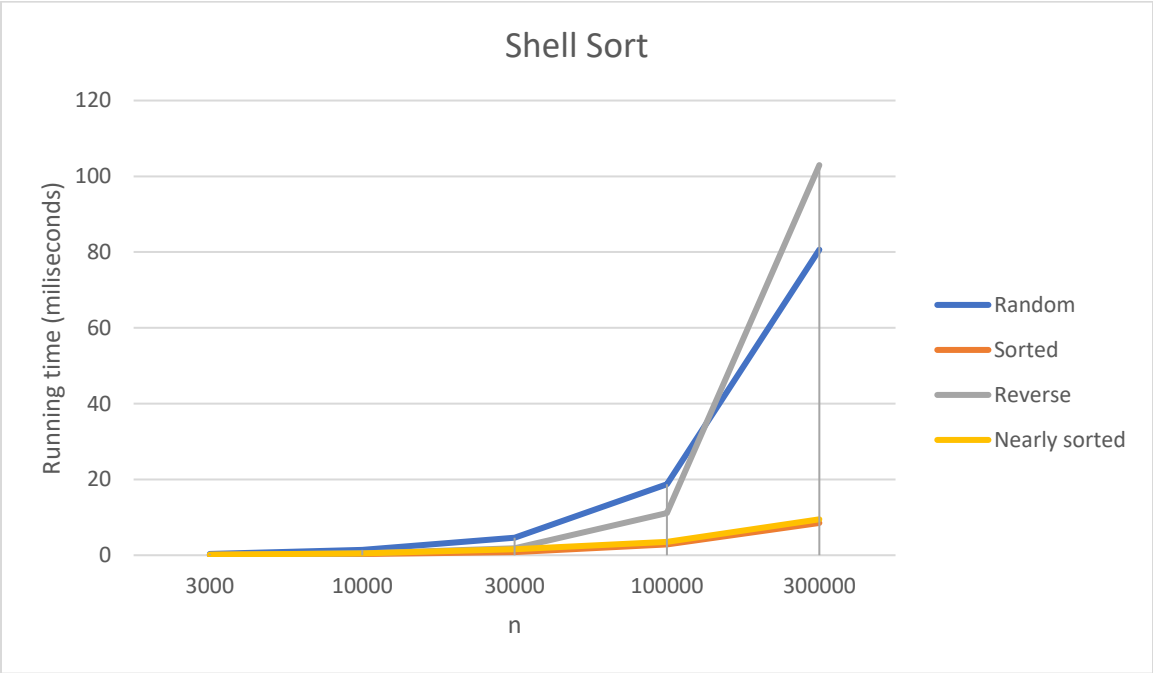
2.6.3.2 Thời gian chạy thực tế

Thời gian chạy với từng loại dữ liệu và n:

n	3000	10000	30000	100000	300000
Random	0.3429	1.3835	4.5902	18.7067	80.6386
Sorted	0.0744	0.3373	0.8357	2.834	8.5114

Reverse	0.1188	0.4574	1.8611	11.1159	102.9577
Nearly sorted	0.1464	0.4764	1.5458	3.4942	9.4775

Biểu đồ tương ứng:



Dù là cải tiến từ thuật toán sắp xếp chèn với độ phức tạp trung bình $O(n^2)$ nhưng thuật toán Shell Sort hoạt động rất hiệu quả, nhanh hơn hẳn so với các thuật toán có độ phức tạp $O(n^2)$ khác như Bubble Sort hay Selection Sort.

2.7 Heap Sort

2.7.1 Ý tưởng thuật toán

Thuật toán Heap Sort có ý tưởng cơ bản là tách mảng ra làm 2 phần, một phần đã được sort tăng dần và một phần duy trì nó là một max-heap.

Max-heap được định nghĩa là một heap với giá trị lớn nhất nằm ở đỉnh heap. Max-heap là một cây nhị phân, với giá trị của nút cha lớn hơn giá trị của 2 nút con.

Thuật toán sẽ đi qua 2 giai đoạn chính:

- Giai đoạn 1: Dựng max-heap.
- Giai đoạn 2: Lặp lại việc đem phần tử đầu max-heap ra sau mảng, điều chỉnh lại heap.

Ở giai đoạn 1, ta sẽ mảng theo thứ tự từ cuối về đầu, với mỗi phần tử ta sẽ đẩy dần nó xuống vị trí thích hợp trong max-heap bằng cách so sánh nó với 2 con, nếu nó có giá trị nhỏ hơn 2 con thì ta sẽ đẩy nó xuống và đẩy con nó lên và tiếp tục thực hiện.

Ở giai đoạn 2, khi đem phần tử đầu max-heap ra sau mảng, ta lại đem phần tử ở cuối mảng hiện tại lên đầu heap, và đẩy nó xuống giống ở trong giai đoạn 1.

2.7.2 Các bước hoạt động

Ví dụ ta có mảng đầu vào như sau:

$$a = \{4, 1, 0, 3, 2\}$$

Ở đây, khi cài đặt heap bằng mảng, tôi xem phần tử đứng ở vị trí i sẽ có con trái đứng tại vị trí $2 * i + 1$ và con phải tại vị trí $2 * i + 2$.

Dưới đây là các bước thực hiện thuật toán ở giai đoạn dựng max-heap. Phần tử màu đỏ chính là phần tử đang được xét, phần tử màu xanh chính là con của phần tử đang xét.

Vị trí đang xét	Mảng a	Giải thích
1	$\{4, \mathbf{1}, 0, \mathbf{3}, \mathbf{2}\}$	Do các phần tử với vị trí thuộc đoạn $[\frac{n}{2}; n-1]$ sẽ không có con nên ta bỏ qua không cần xét, ta sẽ xét từ phần tử ở vị trí $\frac{n}{2} - 1$. Phần tử 1 có hai con là 3 và 2, do ta dựng max-heap nên ta cần đưa 3 lên thay vị trí 1.
0	$\{\mathbf{4}, \mathbf{3}, \mathbf{0}, 1, 2\}$	Ta xét phần tử ở vị trí 0, phần tử này có 2 con có giá trị là 3 và 0 đều nhỏ hơn 4, vì vậy ta không thực hiện gì. Tới đây ta hoàn tất giai đoạn xây dựng max-heap.

Tiếp theo là giai đoạn 2, ở đây tôi sẽ dùng ký tự "|" để phân tách mảng thành 2 phần, phần max-heap và phần đã được sắp xếp.

Lần lặp	Mảng a	Giải thích
1	$\{\mathbf{4}, 3, 0, 1, 2, \mid\}$	Đầu tiên ta đem phần tử 4 về cuối mảng, sau đó đem phần tử 2 lên thay thế phần tử 4.
1	$\{\mathbf{2}, \mathbf{3}, \mathbf{0}, 1, \mid 4\}$	Ta sẽ đẩy phần tử 2 xuống vị trí thích hợp để đảm bảo max heap.
1	$\{3, \mathbf{2}, 0, \mathbf{1}, \mid 4\}$	Tới đây phần tử 2 đã nằm đúng vị trí, ta dừng lần lặp 1. Lưu ý là lúc này con của phần tử này chỉ có 1, vì phần tử 4 đã nằm ngoài vùng quản lý hiện tại.

2	$\{\mathbf{3}, 2, 0, 1, \mid 4\}$	Ta đưa 3 về sau, đem phần tử 1 lên thay 3.
2	$\{\mathbf{1}, \mathbf{2}, \mathbf{0}, \mid 3, 4\}$	Ta cần đưa phần tử 1 tới vị trí thích hợp, ta sẽ thay thế vị trí của phần tử 1 và 2.
2	$\{2, \mathbf{1}, 0, \mid 3, 4\}$	Tới đây phần tử 1 đã nằm đúng vị trí, ta dừng lần lặp 2.
3	$\{\mathbf{2}, 1, 0, \mid 3, 4\}$	Ta đưa 2 về sau, đem phần tử 0 lên thay 2.
3	$\{\mathbf{0}, \mathbf{1} \mid 2, 3, 4\}$	Ta cần đưa phần tử 0 tới vị trí thích hợp, ta sẽ thay thế vị trí của phần tử 0 và 1.
3	$\{1, \mathbf{0}, \mid 2, 3, 4\}$	Phần tử 0 đã nằm đúng vị trí, ta kết thúc lần lặp 3.
4	$\{\mathbf{1}, 0, \mid 2, 3, 4\}$	Ta đưa phần tử 1 về sau, đem phần tử 0 lên thay.
4	$\{0, \mid 1, 2, 3, 4\}$	Phần tử 0 đã nằm đúng vị trí, ta kết thúc vòng lặp 4 và kết thúc thuật toán.

2.7.3 Phân tích thuật toán

2.7.3.1 Độ phức tạp thuật toán

Độ phức tạp của Heap Sort ở giai đoạn 1 sẽ phụ thuộc vào cách dựng max-heap. Với cách dựng tôi trình bày ở trên thì giai đoạn 1 có độ phức tạp $O(n)$ [2].

Độ phức tạp ở giai đoạn 2 của Heap Sort là $O(nlogn)$ vì ta cần thực hiện n lần điều chỉnh heap, mỗi lần điều chỉnh sẽ mất chi phí lớn nhất là độ cao của cây nhị phân, hay $O(logn)$.

Trong trường hợp tốt nhất, thuật toán heap sort có độ phức tạp $O(n)$, và xấu nhất là $O(nlogn)$.

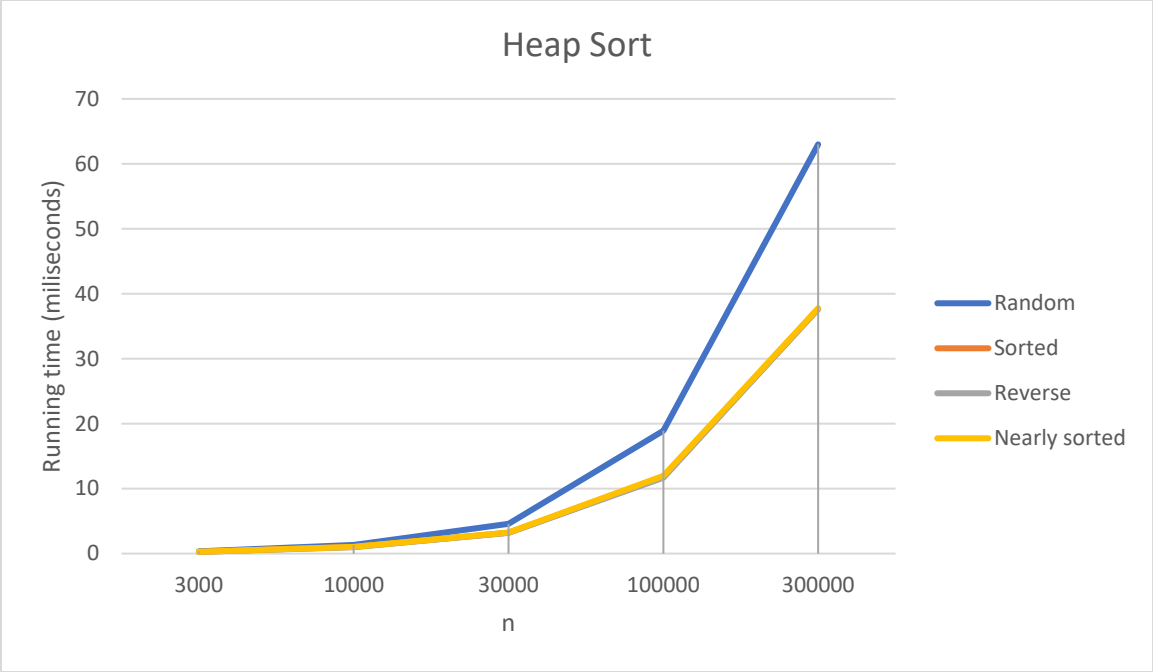
Bộ nhớ cần dùng của thuật toán là $O(1)$.

2.7.3.2 Thời gian chạy thực tế

Thời gian chạy với từng loại dữ liệu và n:

n	3000	10000	30000	100000	300000
Random	0.3471	1.3228	4.5569	18.9012	63.0007
Sorted	0.2737	1.0443	3.2226	11.8128	37.6452
Reverse	0.2686	0.9956	3.2013	11.6675	37.6676
Nearly sorted	0.2761	1.0103	3.2286	11.9746	37.7667

Biểu đồ tương ứng:



Ta thấy với những bộ dữ liệu đã được sắp xếp hoặc gần như sắp xếp thì thời gian chạy của thuật toán heap sort gần như là tương đương nhau chỉ trừ khi dữ liệu ngẫu nhiên thì mới có sự khác biệt rõ rệt về thời gian chạy. Tôi vẫn chưa giải thích được tại sao lại có sự tương đồng này.

2.8 Merge Sort

2.8.1 Ý tưởng thuật toán

Thuật toán sắp xếp trộn là một thuật toán chia để trị, thuật toán này hoạt động một cách đệ quy như sau:

- Nếu mảng có ít hơn 2 phần tử thì ta không làm gì.
- Ngược lại, ta phân mảng ra làm 2 phần có số lượng phần tử chênh lệch không quá một, sau đó ta sắp xếp 2 phần đó, rồi trộn lại với nhau.

2.8.2 Các bước hoạt động

Ví dụ ta có mảng đầu vào như sau:

$$a = \{4, 1, 0, 3, 2\}$$

Ở đây tôi dùng các cặp ngoặc "[]" để thể hiện các mảng đang được gọi đệ quy.

Dưới đây là các bước thực hiện thuật toán:

Mảng a	Giải thích
$[4, 1, 0, 3, 2]$	Ta sẽ chia mảng ra làm 2 phần [4, 1, 0] và [3, 2]

$[[4, 1, 0], [3, 2]]$	Ta chia [4, 1, 0] làm 2 phần [4, 1] và [0]
$[\left[[4, 1], [0]\right], [3, 2]]$	Ta chia [4, 1] làm 2 phần [4] và [1], sau đó ta trộn về thành [1, 4]
$[[1,4], [0]], [3, 2]]$	Ta trộn [1, 4] và [0] thành [0, 1, 4]
$[[0,1,4], [3, 2]]$	Ta chia [3,2] làm 2 phần [3] và [2] sau đó trộn thành [2, 3]
$[[0,1,4], [2, 3]]$	Ta trộn [0, 1, 4] và [2, 3] thành [0, 1, 2, 3, 4]
$[0, 1, 2, 3, 4]$	Thuật toán dừng.

2.8.3 Phân tích thuật toán

2.8.3.1 Độ phức tạp thuật toán

Độ phức tạp của thuật toán sắp xếp trộn là $O(nlogn)$ ở mọi trường hợp. Ta có thể chứng minh nó như sau:

- Tạm gọi $T(n)$ là thời gian thực thi của thuật toán sắp xếp trộn cho mảng có n phần tử.
- Ta có hệ thức đệ quy sau: $T(n) = 2 * T\left(\frac{n}{2}\right) + n$. Vì ta cần giải bài toán con cho 2 mảng con với độ dài $\frac{n}{2}$, và tốn n cho phép trộn 2 mảng con.
- $T(n) = 4 * T\left(\frac{n}{4}\right) + n + \left(\frac{n}{2}\right) * 2 = 4 * T\left(\frac{n}{4}\right) + n + n = 2^{logn} * T\left(\frac{n}{2^{logn}}\right) + nlogn = n + nlogn = O(nlogn)$

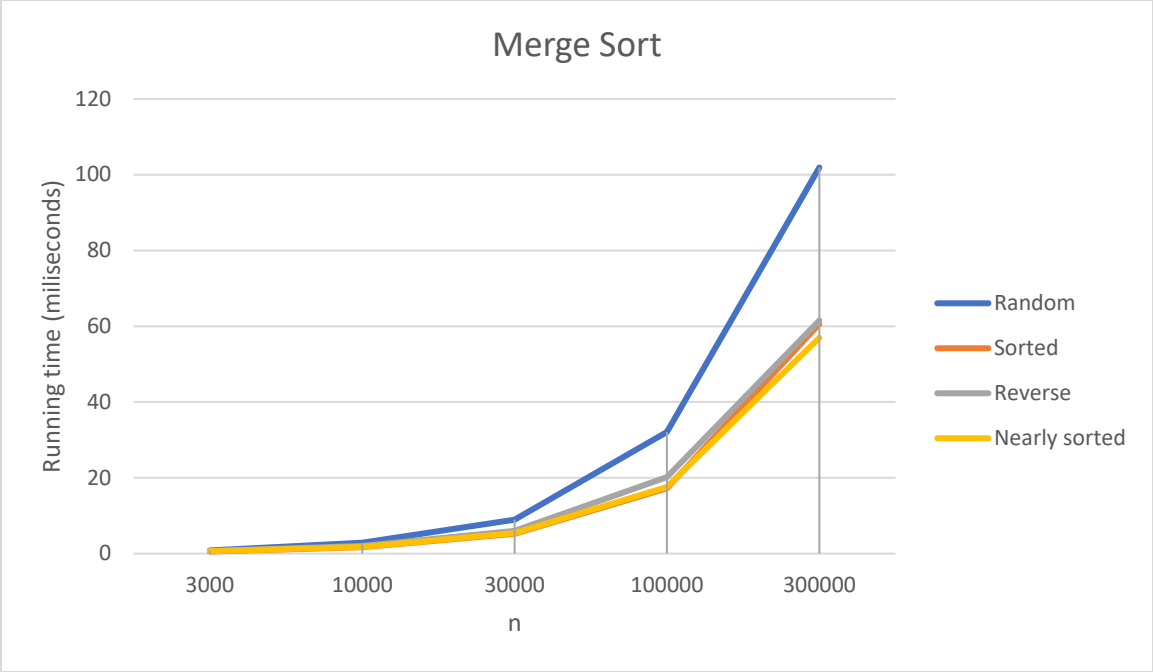
Bộ nhớ cần dùng của thuật toán là $O(n + logn) = O(n)$, với n là độ dài mảng, và $logn$ tầng đệ quy.

2.8.3.2 Thời gian chạy thực tế

Thời gian chạy với từng loại dữ liệu và n:

n	3000	10000	30000	100000	300000
Random	0.8323	2.8312	8.9335	32.1014	101.8882
Sorted	0.537	1.7154	5.2519	17.242	60.6117
Reverse	0.6683	2.0351	6.0133	20.1755	61.5252
Nearly sorted	0.6236	1.8125	5.3858	17.5332	56.9513

Biểu đồ tương ứng:



2.9 Quick Sort

2.9.1 Ý tưởng thuật toán

Giống với sắp xếp trộn, thuật toán sắp xếp nhanh cũng là một thuật toán chia để trị, thuật toán này hoạt động một cách đệ quy như sau:

- Nếu mảng có ít hơn 2 phần tử thì ta không làm gì.
- Ngược lại, ta chọn một phần tử (gọi là phần tử chốt), ta phân mảng ra làm 2 phần, một phần gồm các phần tử bé hơn phần tử chốt (nằm bên trái), và các phần tử còn lớn hơn phần tử chốt.

2.9.2 Các bước hoạt động

Ví dụ ta có mảng đầu vào như sau:

$$a = \{4, 1, 0, 3, 2\}$$

Ở đây tôi dùng các cặp ngoặc "[]" để thể hiện các mảng đang được gọi đệ quy. Màu đỏ là phần tử chốt.

Dưới đây là các bước thực hiện thuật toán:

Mảng a	Giải thích
$[4, 1, 0, 3, \mathbf{2}]$	Chọn 2 làm phần tử chốt. Phân làm 2 mảng $[1, 0]$ và $[4, 3]$

$[[\mathbf{1}, 0], 2, [4, 3]]$	Ở [1, 0] ta chọn [1] làm phần tử chốt. Phân ra 2 mảng [0] và [1], tới đây ta thực hiện xong một phần.
$[0, 1, 2, [\mathbf{4}, 3]]$	Ta chọn 4 làm phần tử chốt, phân làm 2 mảng [3] và 4.

2.9.3 Phân tích thuật toán

2.9.3.1 Độ phức tạp thuật toán

Độ phức tạp của thuật toán sắp xếp nhanh phụ thuộc nhiều vào cách ta chọn phần tử chốt. Nếu ta chọn tốt, khi phần tử chốt chia mảng ra làm 2 phần có độ dài xấp xỉ bằng nhau thì độ phức tạp của sắp xếp nhanh là $O(nlogn)$. Ta có thể chứng minh nó như sau:

- Tạm gọi $T(n)$ là thời gian thực thi của thuật toán sắp xếp nhanh cho mảng có n phần tử.
- Ta có hệ thức đệ quy sau: $T(n) = 2 * T\left(\frac{n}{2}\right) + n$. Vì ta cần giải bài toán con cho 2 mảng con với độ dài $\frac{n}{2}$, và tốn n việc phân 2 mảng con.
- $T(n) = 4 * T\left(\frac{n}{4}\right) + n + \left(\frac{n}{2}\right) * 2 = 4 * T\left(\frac{n}{4}\right) + n + n = 2^{logn} * T\left(\frac{n}{2^{logn}}\right) + nlogn = n + nlogn = O(nlogn)$

Nhưng điều trên chỉ đúng khi ta chọn được các phần tử chốt tốt (khi phần tử là trung vị của dãy), nhưng nếu chọn không tốt thì độ phức tạp ở trường hợp xấu nhất là $O(n^2)$ (khi mà mỗi lần chọn phần tử chốt thì phân ra làm 2 dãy có độ dài 1 và $n - 1$).

Ở trường hợp tốt nhất, thuật toán có độ phức tạp $O(nlogn)$.

Bộ nhớ cần dùng của thuật toán là $O(logn)$ ở trường hợp trung bình, vì ta gọi để quy nên sẽ tốn chi phí bộ nhớ stack. Ở trường hợp tệ nhất, khi chọn chốt tệ, bộ nhớ cần dùng có thể lên tới $O(n)$.

Nếu ta chọn phần tử chốt một cách cố định thì rất dễ rơi vào trường hợp xấu, chẳng hạn nếu ta chọn phần tử đầu tiên làm phần tử chốt và mảng được sắp xếp giảm dần thì sẽ dẫn tới trường hợp mà ta đệ quy xâu n bước và dẫn tới tràn stack trong quá trình đệ quy. Do đó, trong cách cài đặt của tôi, tôi chọn phần tử chốt một cách ngẫu nhiên để giảm khả năng rơi vào trường hợp xấu.

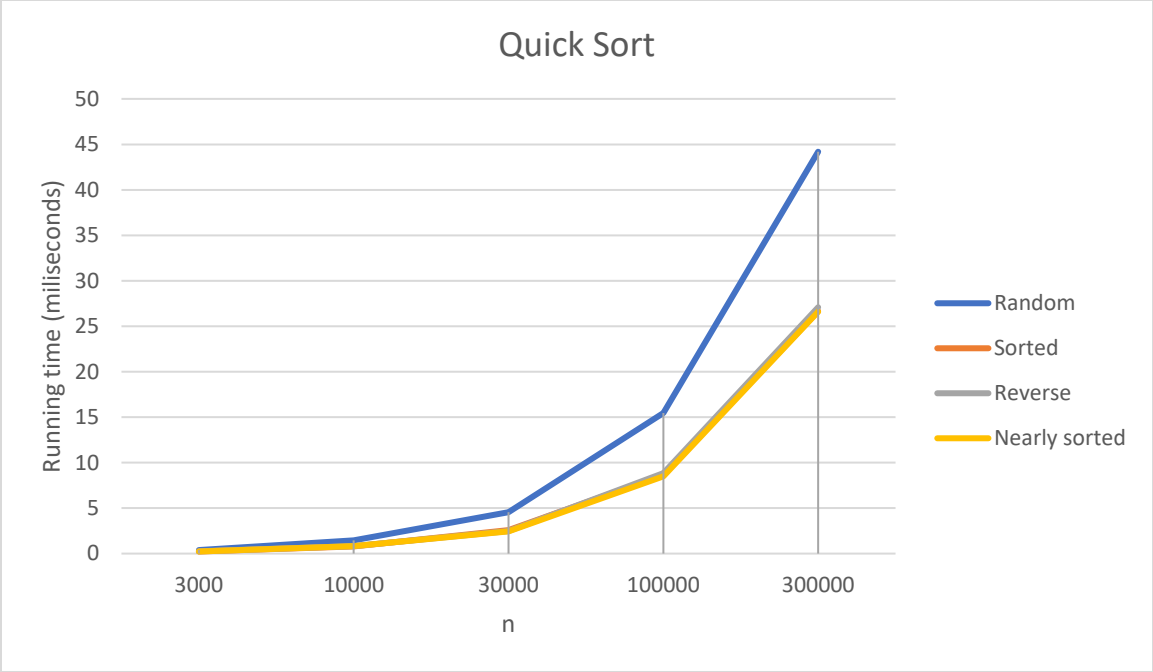
2.9.3.2 Thời gian chạy thực tế

Thời gian chạy với từng loại dữ liệu và n:

n	3000	10000	30000	100000	300000
Random	0.3867	1.4415	4.554	15.459	44.1892
Sorted	0.2298	0.7917	2.5861	8.6606	26.6561
Reverse	0.2545	0.8229	2.4592	8.833	27.0834

Nearly sorted	0.232	0.8046	2.4427	8.5029	26.5956

Biểu đồ tương ứng:



Khi mảng đã được sắp xếp hoặc gần được sắp xếp thì số lần hoán vị 2 phần tử sẽ ít, nên thời gian chạy chủ yếu ở phần gọi đệ quy và duyệt lại từng phần. Tương tự khi mảng đã được sắp xếp ngược thì sau lần phân hoặc đầu tiên, mảng cũng có dạng giống như đã được sắp xếp, do đó số phép hoán vị sẽ ít. Với trường hợp mảng ngẫu nhiên thì số lần hoán vị sẽ nhiều hơn, do đó thời gian chạy lớn hơn hẳn so với 3 kiểu dữ liệu còn lại.

Dưới đây là bảng thống kê số lần hoán vị:

n	3000	10000	30000	100000	300000
Random	9207	34169	115783	436608	1499355
Sorted	2197	7364	22069	73576	220588
Reverse	3682	12340	37097	123537	370609
Nearly sorted	2219	7373	22124	73511	220635

Ta có thể thấy khi mảng được cho ngẫu nhiên thì số lần hoán vị rất lớn, đó là lí do làm thời gian chạy lâu hơn khi mảng ngẫu nhiên.

2.10 Couting Sort

2.10.1 Ý tưởng thuật toán

Thuật toán sắp xếp đếm phân phối có ý tưởng khá đơn giản, ta sẽ đếm số lượng phần tử có giá trị x, sau đó từ mảng đếm phân phối này ta có thể đưa ra mảng đã được sắp xếp.

2.10.2 Các bước hoạt động

Ví dụ ta có mảng đầu vào như sau:

$$a = \{3, 1, 0, 3, 2\}$$

Thì ta sẽ có mảng đếm phân phối như sau:

- $cnt[0] = 1; cnt[1] = 1; cnt[2] = 1; cnt[3] = 2;$

Tới đây ta có thể đưa ra mảng gồm 1 số 0, rồi 1 số 1, rồi 1 số 2, rồi 2 số 3:

$$a = \{0, 1, 2, 3, 3\}$$

2.10.3 Phân tích thuật toán

2.10.3.1 Độ phức tạp thuật toán

Độ phức tạp của thuật toán sắp xếp đếm phân phối phụ thuộc vào độ lớn của mảng đầu vào. Nó có độ phức tạp $O(n + k)$ với n là độ dài mảng và k là chênh lệch của phần tử lớn nhất và phần tử bé nhất của mảng đầu vào.

Với bộ dữ liệu tôi dùng, do các phần tử đều nằm trong đoạn $[0, n - 1]$ nên trong trường hợp này, độ phức tạp của thuật toán sắp xếp đếm phân phối là $O(n + k)$ với $k \leq n$.

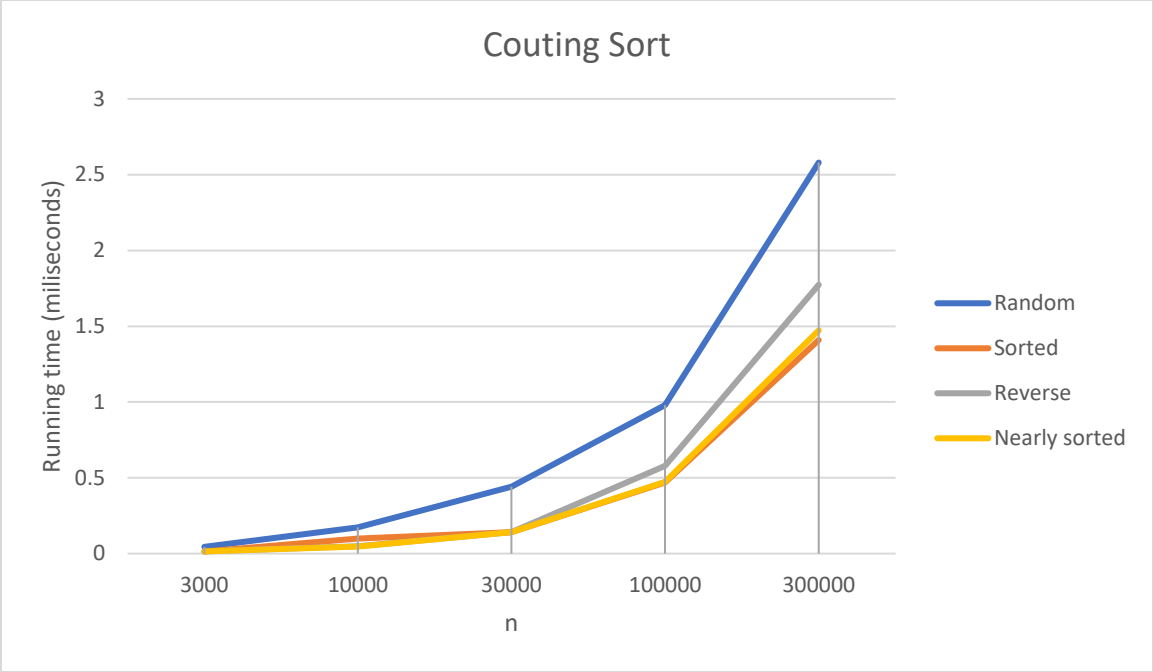
Bộ nhớ cần dùng của thuật toán là $O(k)$ với $k \leq n$.

2.10.3.2 Thời gian chạy thực tế

Thời gian chạy với từng loại dữ liệu và n:

n	3000	10000	30000	100000	300000
Random	0.0445	0.1738	0.4404	0.9786	2.58
Sorted	0.0139	0.0985	0.1405	0.4685	1.4089
Reverse	0.0146	0.0475	0.1408	0.5788	1.7736
Nearly sorted	0.0144	0.0462	0.1424	0.4729	1.4725

Biểu đồ tương ứng:



Lại một lần nữa, với dữ liệu được sinh ngẫu nhiên thì sẽ có thời gian chạy lâu hơn những dữ liệu khác nhưng không quá đáng kể (chưa tới 1ms). Thuật toán Counting Sort có độ phức tạp tuyến tính cùng với hằng số lập trình thấp nên có thời gian chạy nhanh hơn hẳn các thuật toán khác (chưa tới 3ms cho trường hợp chậm nhất).

2.11 Radix Sort

2.11.1 Ý tưởng thuật toán

Tương tự đếm phân phối, sắp xếp cơ số là một thuật toán sắp xếp không dựa vào so sánh mà dựa vào phân bố của dữ liệu. Thuật toán này, ở mỗi bước sẽ phân dữ liệu thành từng nhóm khác nhau dựa trên cơ số của nó trong một hệ cơ số nào đó.

Thuật toán sắp xếp cơ số có 2 trường phái đó là sắp xếp từ hàng đơn vị lên (least significant digit) hoặc sắp xếp từ hàng cao nhất xuống (most significant digit).

Ở bài báo cáo này, tôi sẽ trình bày về sắp xếp theo hàng cao nhất xuống trong hệ cơ số 2.

Với hệ cơ số 2, ta có thể xem mỗi bước chính là phân hoạch dãy làm 2 phần, phần chỉ chứa các số có bit tại hàng đang xét là 0, và chứa các số có bit tại hàng đang xét là 1. Phần phân hoạch này có thể cài đặt giống với cách phân hoạch ở thuật toán sắp xếp nhanh.

2.11.2 Các bước hoạt động

Ví dụ ta có mảng đầu vào như sau:

$$a = \{4, 1, 0, 3, 2\}$$

Ta có 3 bước để xét từ bit 2, bit 1 và bit 0. Ở đây tôi dùng ký tự "|" để phân cách các phần của mảng.

Bit	Mảng a	Giải thích
2	$\{4\mid 1, 0, 3, 2\}$	Do 4 là số duy nhất có bit thứ 2 là 1, nên nó sẽ được phân hoạch về phần sau. Ta sẽ gọi đệ quy để phân hoạch phần [1, 0, 3, 2]
1	$\{1, 0, 3, 2 \mid 4\}$	Số 3 và 2 có bit 1 là 1 nên sẽ được phân hoạch về sau. Ta gọi đệ quy để phân hoạch [1, 0] và [3, 2]
0	$\{1, 0 \mid 3, 2 \mid 4\}$	Số 1 có bit 0 là 1 nên được phân hoạch về sau.
0	$\{0 \mid 1 \mid 3, 2 \mid 4\}$	Số 3 có bit 0 là 1 nên được phân hoạch về sau
	$\{0 \mid 1 \mid 2 \mid 3 \mid 4\}$	Thuật toán dừng

2.11.3 Phân tích thuật toán

2.11.3.1 Độ phức tạp thuật toán

Độ phức tạp theo cách cài đặt ở hệ cơ số 2 và sắp xếp từ hàng cao xuống đó là $O(nlogk)$ với n là độ dài mảng, và k là số lớn nhất trong mảng. Có độ phức tạp này là vì ta cần $O(logk)$ tầng gọi đệ quy, và mỗi tầng gọi ta đều phải xử lý $O(n)$ phần tử.

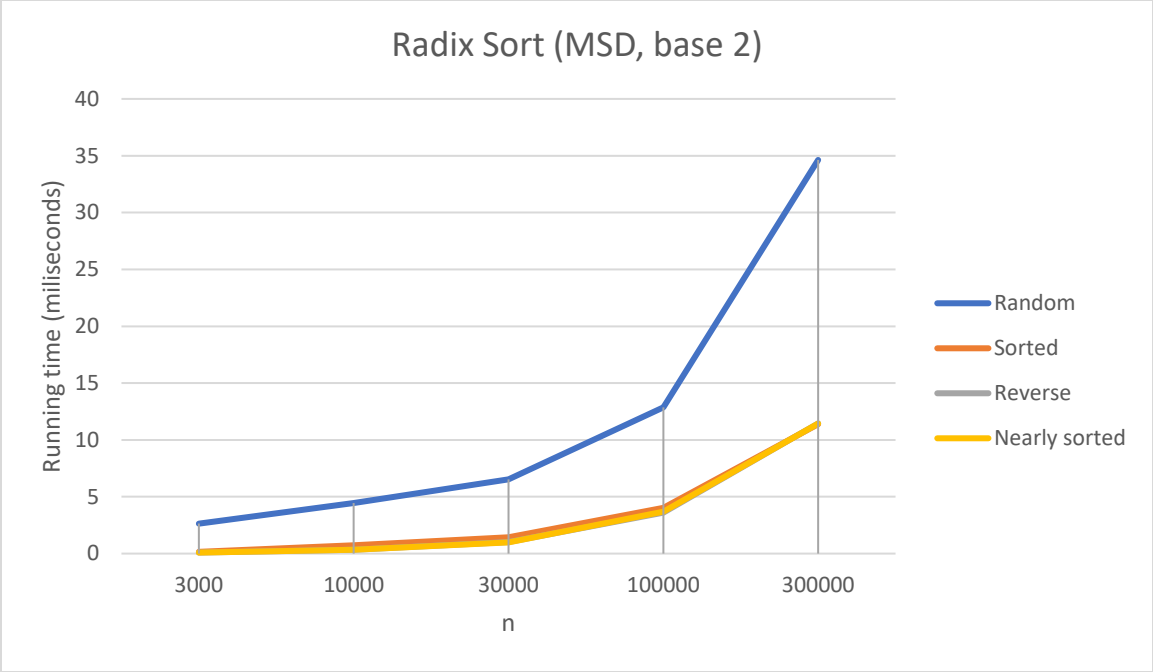
Bộ nhớ cần dùng của thuật toán là $O(logk)$, với k là số lớn nhất trong mảng vì ta có k tầng đệ quy.

2.11.3.2 Thời gian chạy thực tế

Thời gian chạy với từng loại dữ liệu và n:

n	3000	10000	30000	100000	300000
Random	2.6345	4.4547	6.5164	12.874	34.63
Sorted	0.1529	0.7359	1.4421	4.0386	11.3833
Reverse	0.0873	0.3401	1.02	3.6129	11.4525
Nearly sorted	0.0887	0.3321	0.9954	3.6829	11.4399

Biểu đồ tương ứng:



Lại một lần nữa, với dữ liệu được sinh ngẫu nhiên thì sẽ có thời gian chạy lâu hơn những dữ liệu khác. Tôi nghĩ lý do cũng tương tự như ở thuật toán sắp xếp nhanh: khi dữ liệu ngẫu nhiên thì ta sẽ cần nhiều phép hoán vị hơn, mặc dù số lần duyệt tương tự nhau, nhưng số phép hoán vị khác nhau.

Dưới đây là bảng thống kê số lần hoán vị:

n	3000	10000	30000	100000	300000
Random	7941	30645	103684	366645	1116088
Sorted	0	0	0	0	0
Reverse	1500	5000	15000	50000	150000
Nearly sorted	22	22	22	22	22

Đúng như tôi dự đoán, số lần hoán vị khi mảng ngẫu nhiên là lớn hơn hẳn so với những loại dữ liệu khác.

2.12 Flash Sort

2.12.1 Ý tưởng thuật toán

Flash sort là một thuật toán vừa dựa vào phân phối của dữ liệu vừa dựa vào các phép so sánh.

Ý tưởng của thuật toán là chia các phần tử vào m phân đoạn khác nhau, phần tử $a[i]$ sẽ nằm ở phân đoạn thứ $1 + \left\lfloor (m-1) * \frac{a[i]-a_{min}}{a_{max}-a_{min}} \right\rfloor$.

Sau khi phân hoạch xong thì thuật toán sẽ thực hiện sắp xếp các phần tử trong cùng một phân đoạn bằng thuật toán sắp xếp chèn.

2.12.2 Các bước hoạt động

Ví dụ ta có mảng đầu vào như sau:

$$a = \{4, 1, 0, 3, 2\}$$

Và $m = 3$ (số phân đoạn)

Bảng phân hoạch:

Phần tử	Phân đoạn
4	3
1	1
0	1
2	2
3	3

Sau khi phân hoạch:

$$a = \{1, 0 \mid 3, 2 \mid 4\}$$

Ở đây tôi dùng ký tự "|" để phân cách các phân đoạn.

Bước tiếp theo, ta sẽ dùng sắp xếp chèn để sắp xếp từng phân đoạn:

$$a = \{0,1 \mid 2, 3 \mid 4\}$$

Tới đây thì thuật toán dừng.

2.12.3 Phân tích thuật toán

2.12.3.1 Độ phức tạp thuật toán

Giai đoạn phân hoạch có độ phức tạp là $O(n)$ vì mỗi phần tử ta chỉ duyệt qua duy nhất một lần.

Giai đoạn sắp xếp: trung bình thì mỗi phân đoạn sẽ có $\frac{n}{m}$ phần tử và do sắp xếp chèn cần có độ phức tạp $O(n^2)$ với n là số phần tử nên mỗi phân đoạn ta sắp xếp sẽ mất $O\left(\frac{n^2}{m^2}\right)$.

Ta có m phân đoạn nên độ phức tạp để sắp xếp m phân đoạn là: $O\left(\frac{n^2}{m^2} * m\right) = O\left(\frac{n^2}{m}\right)$.

Theo thực nghiệm, ta nên chọn $m = 0.43n$ [3]

Nếu thay $m = 0.43n$ vào ta sẽ có độ phức tạp của giai đoạn sắp xếp là: $O\left(\frac{n^2}{0.43n}\right) = O(n)$.

Tuy nhiên đó chỉ là trường hợp trung bình, ở trường hợp tệ nhất, khi mà các dữ liệu phân bố không đều thì độ phức tạp của flash sort có thể lên tới $O(n^2)$, tuy nhiên trường hợp này rất hiếm khi xảy ra nếu dữ liệu hoàn toàn ngẫu nhiên.

Bộ nhớ cần dùng là $O(m) = O(n)$

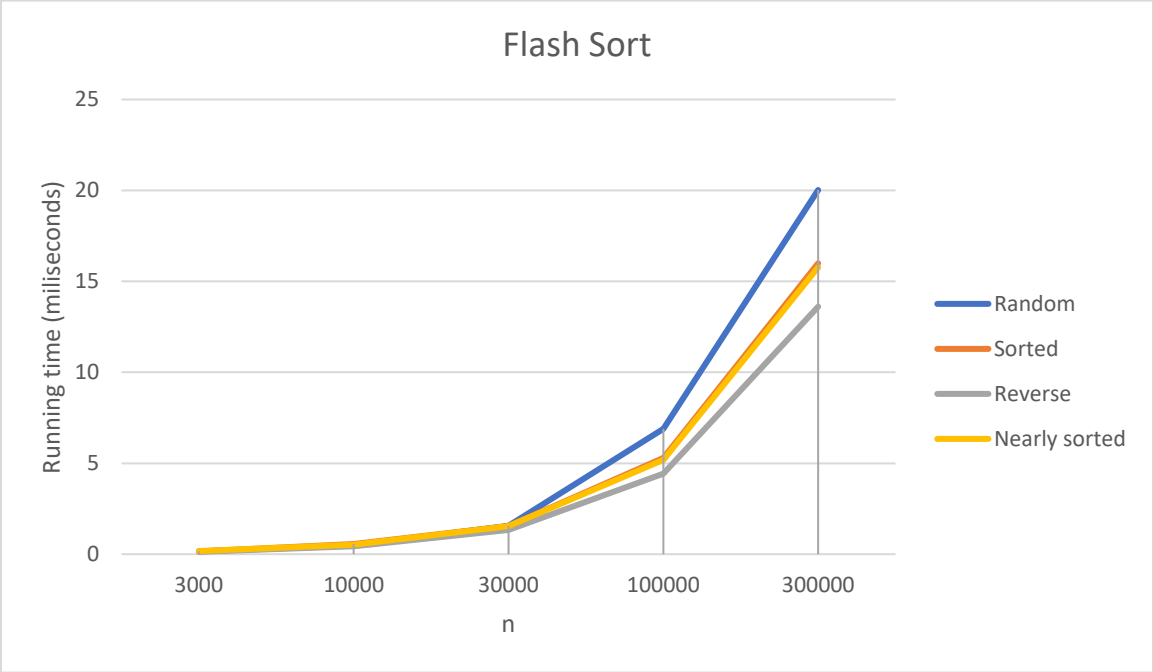
Tuy vậy, do giới hạn của các dữ liệu có tính đặc thù, tôi đã chọn $m = n$. Điều này sẽ giúp tôi đảm bảo được rằng mọi phân đoạn đều chứa các phần tử giống nhau (vì các phần tử $a[i]$ nhỏ hơn n), do đó thuật toán gần như chỉ tốn chi phí ở giai đoạn phân hoạch. Lưu ý tôi chỉ chọn $m = n$ vì giới hạn của dữ liệu cho phép.

2.12.3.2 Thời gian chạy thực tế

Thời gian chạy với từng loại dữ liệu và n:

n	3000	10000	30000	100000	300000
Random	0.157	0.4712	1.5617	6.8922	20.0238
Sorted	0.1538	0.5577	1.547	5.2871	15.9952
Reverse	0.1323	0.4432	1.3256	4.4227	13.609
Nearly sorted	0.1711	0.5202	1.5505	5.1865	15.7739

Biểu đồ tương ứng:



Thuật toán Flash Sort đã loại bỏ được yếu điểm của Counting Sort bằng phân hoạch các phần tử bằng chênh lệch so với phần tử nhỏ nhất. Sau khi phân hoạch thì từng phần có số lượng phần tử nhỏ, Flash Sort lại áp dụng Insertion Sort vào những phần nhỏ này,

điều này tận dụng được thời gian chạy tốt của Insertion Sort ở những trường hợp số lượng phần tử ít. Do có những ưu điểm này nên thuật toán Flash Sort có thời gian chạy rất tốt so với các thuật toán khác.

3 So sánh

3.1 So sánh độ phức tạp

Dưới đây là bảng so sánh về độ phức tạp của các thuật toán mà tôi đã trình bày. Lưu ý độ phức tạp dưới đây phụ thuộc vào giới hạn của dữ liệu mà tôi đã trình bày ở phần đầu báo cáo.

Thuật toán	Tốt nhất	Trung bình	Tệ nhất	Bộ nhớ	Ổn định	Ghi chú
Selection Sort	n^2	n^2	n^2	1	Không	
Selection Sort Optimize1	n^2	n^2	n^2	1	Không	
Insertion Sort	n	n^2	n^2	1	Có	
Binary Insertion Sort	$nlogn$	n^2	n^2	1	Có	Thuật toán ổn định hay không phụ thuộc vào bước tìm nhị phân.
Bubble Sort	n^2	n^2	n^2	1	Có	
Bubble Sort Optimize1	n	n^2	n^2	1	Có	
Shaker Sort	n^2	n^2	n^2	1	Có	
Shaker Sort Optimize1	n	n^2	n^2	1	Có	
Shell Sort	.	.	.	1	Không	Độ phức tạp phụ thuộc vào cách chọn **gaps.**
Heap Sort	$nlogn$	$nlogn$	$nlogn$	1	Không	
Merge Sort	$nlogn$	$nlogn$	$nlogn$	n	Có	
Quick Sort	$nlogn$	$nlogn$	n^2	$logn$	Không	
Couting Sort	$n+k$	$n+k$	$n+k$	k	Có	k là giới hạn của các phần tử, do ở trường hợp dữ liệu này ta có $k = O(n)$ nên độ phức tạp có thể nói là $O(n)$
Radix Sort	$nlogk$	$nlogk$	$nlogk$	$logk$	Tuỳ	Ở đây thuật toán Radix Sort của tôi được cài đặt trên hệ cơ số 2, most significant digit. Có

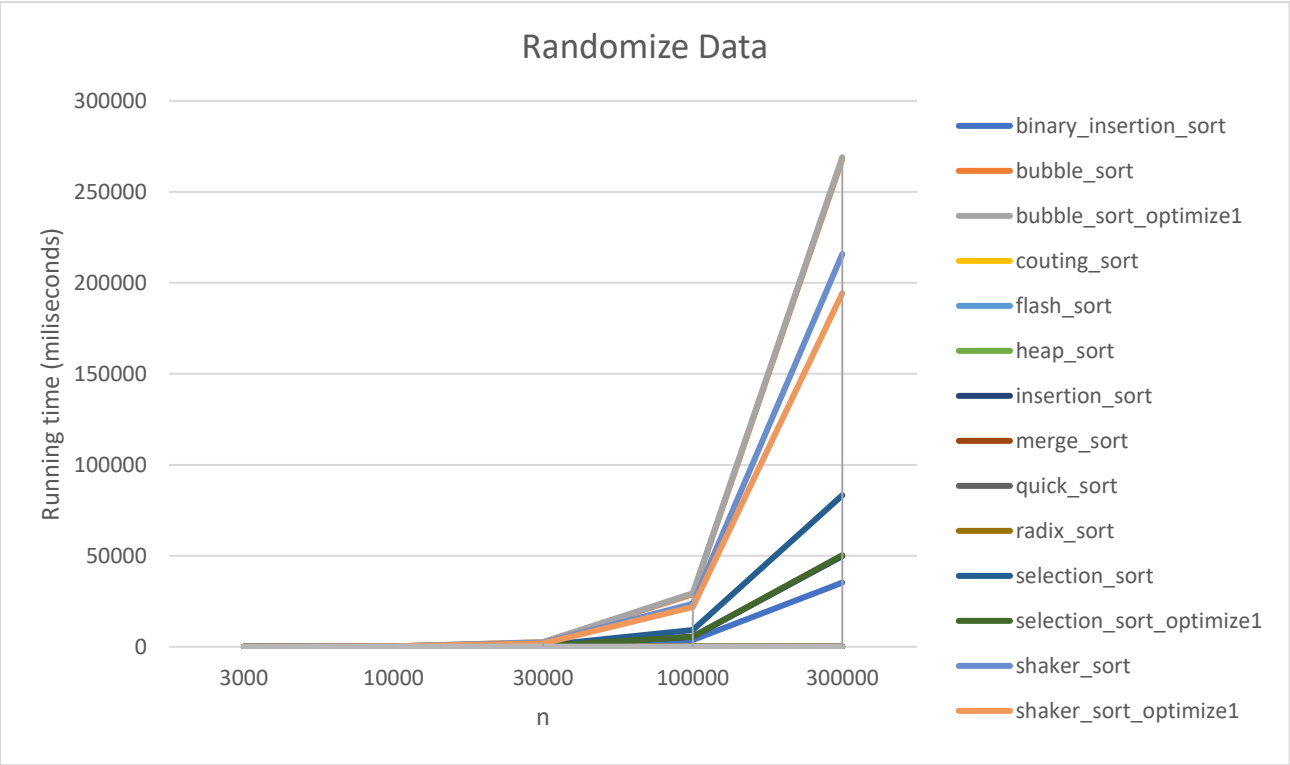
						cách cài đặt để Radix sort ổn định, nhưng ở cách cài đặt của tôi thì nó không ổn định.
Flash Sort	$n+m$	$n+m$	n^2	m	Không	Với m chính là số lượng phân đoạn. Về mặt tổng quát thì độ phức tạp ở trường hợp tệ nhất của Flash Sort là $O(n^2)$ trong trường hợp mà việc phân đoạn không đều. Tuy nhiên với dữ liệu hiện tại thì trường hợp này không xảy ra, và độ phức tạp tệ nhất là $O(n+m)$.

3.2 Thời gian chạy với bộ dữ liệu ngẫu nhiên

Bảng thời gian chạy

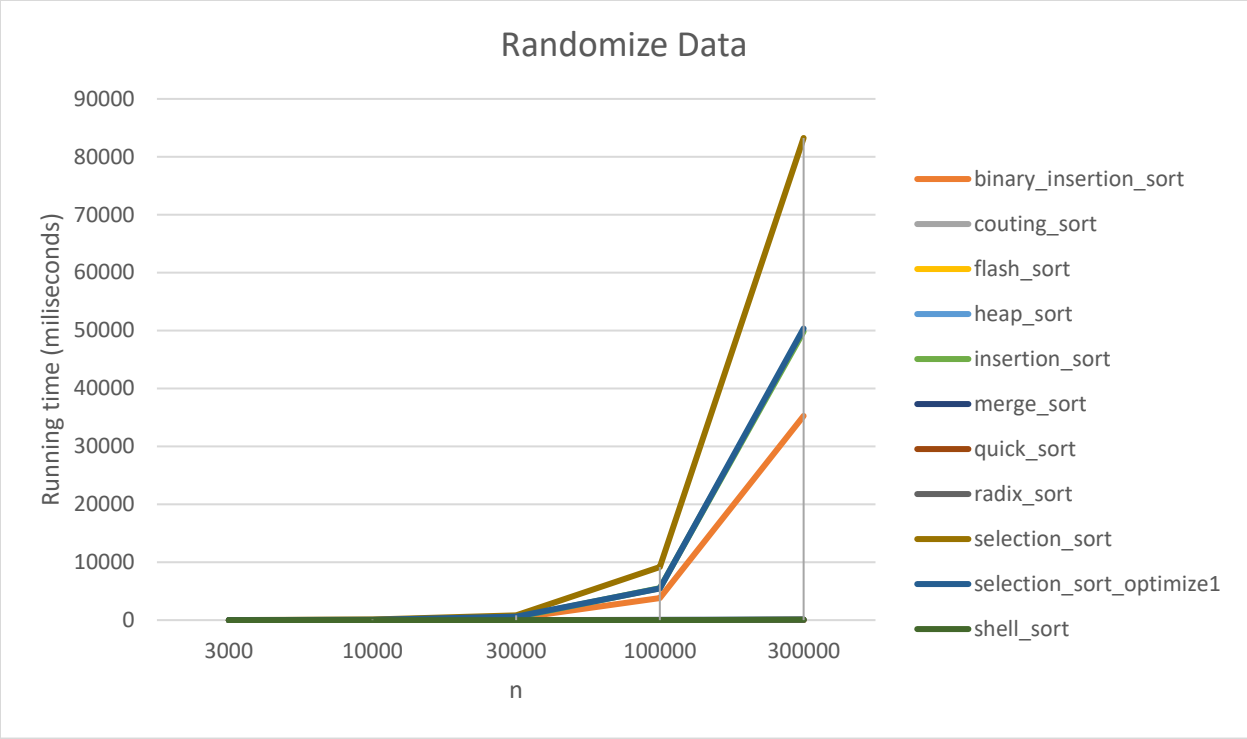
n	3000	10000	30000	100000	300000
binary_insertion_sort	3.836	39.5626	350.8574	3790.6557	35262.0056
bubble_sort	17.6753	243.5058	2456.8087	28943.7081	268048.1359
bubble_sort_optimize1	18.8777	253.1162	2497.2126	29417.849	269035.6476
couting_sort	0.0445	0.1738	0.4404	0.9786	2.58
flash_sort	0.157	0.4712	1.5617	6.8922	20.0238
heap_sort	0.3471	1.3228	4.5569	18.9012	63.0007
insertion_sort	4.8686	54.9648	502.9288	5514.0076	49826.2318
merge_sort	0.8323	2.8312	8.9335	32.1014	101.8882
quick_sort	0.3867	1.4415	4.554	15.459	44.1892
radix_sort	2.6345	4.4547	6.5164	12.874	34.63
selection_sort	8.9588	92.7395	833.9812	9186.2415	83252.1522
selection_sort_optimize1	5.0227	54.8072	606.54	5483.5791	50352.2169
shaker_sort	18.1968	223.8228	2084.4841	23692.7625	215826.0308
shaker_sort_optimize1	18.7474	210.39	1872.8078	21868.2865	194324.4909

shell_sort	0.3429	1.3835	4.5902	18.7067	80.6386



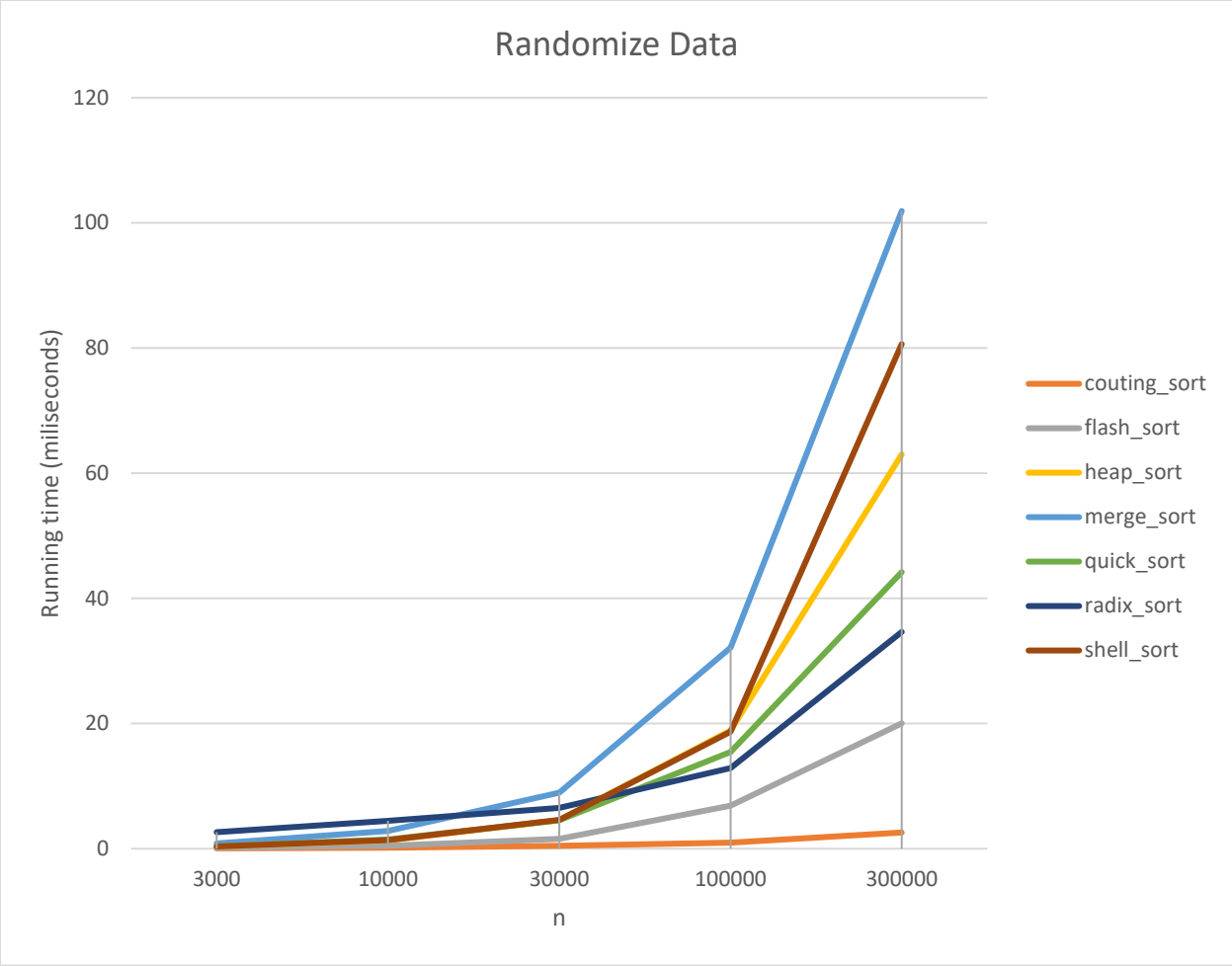
Đồ thị này có phần khó xem vì một số thuật toán với độ phức tạp $O(n^2)$ như Bubble Sort, Shaker Sort hay Selection Sort có thời gian chạy quá lớn, làm ta khó nhìn thấy phần còn lại. Qua đây ta thấy trong các thuật toán sắp xếp cơ bản với độ phức tạp $O(n^2)$ thì Bubble sort là thuật toán tệ nhất, dù có tối ưu nó bằng Shaker Sort cũng không cải thiện tốc độ lắm.

Tôi sẽ loại bỏ đi các thuật toán này (cũng như những tối ưu của nó) trong phần biểu đồ sau nhằm xem chi tiết hơn các thuật toán khác.



Qua đồ thị này ta có thể thấy, thuật toán Selection Sort sẽ là lâu nhất trong các thuật toán còn lại, chậm gần gấp đôi so với Insertion Sort. Ta cũng thấy là Selection Sort Optimize 1 sẽ có tốc độ tương đương với Insertion Sort (đồ thị của 2 thuật toán này nằm chồng lên nhau). Thuật toán Binary Insertion Sort khá tốt, có thể nói là thuật toán chạy nhanh nhất trong các thuật toán $O(n^2)$ ngoại trừ Shell Sort.

Tôi tiếp tục loại bỏ thuật toán Selection Sort, Insertion Sort, Binary Insertion Sort để tiện quan sát:



Trong các thuật toán còn lại, đa số đều là các thuật toán có độ phức tạp $O(nlogn)$ hoặc tuyến tính. Rất bất ngờ đó là thuật toán Shell Sort có tốc độ nhanh hơn Merge Sort.

Thuật toán Radix Sort ở các bộ dữ liệu nhỏ $(n \leq 30\,000)$ thì chạy chậm hơn đa số các thuật toán khác, nhưng khi n lớn thì chỉ chạy chậm hơn Flash Sort và Counting Sort.

Trong 3 thuật toán Merge Sort, Heap Sort, Quick Sort thì Quick Sort có tốc độ nhanh nhất, rồi tới Heap Sort và sau cùng là Merge Sort. Thuật toán Quick Sort nếu cài đặt tốt, không bị rơi vào trường hợp chọn chốt tệ thì sẽ có tốc độ thực thi rất nhanh chóng.

Thuật toán nhanh nhất chính là Counting Sort, với độ phức tạp tuyến tính cùng hằng số lập trình nhỏ, thuật toán có thời gian chạy nhanh, dễ cài đặt nhưng chỉ khả dụng nếu giới hạn dữ liệu nhỏ.

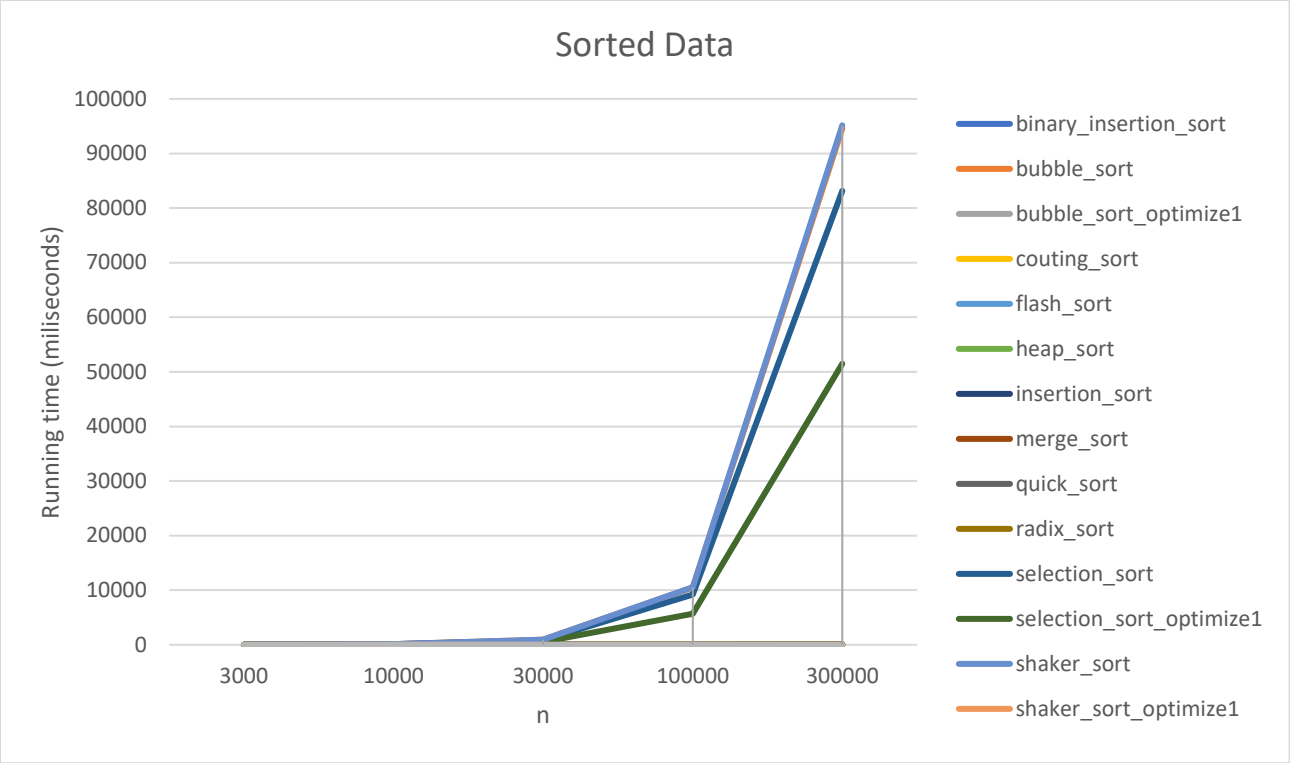
Thuật toán nhanh tiếp theo là Flash Sort, về trung bình cũng có độ phức tạp tuyến tính, tuy nhiên do Flash Sort có hằng số lập trình cao nên vẫn chậm hơn Counting Sort rất nhiều, tuy vậy FlashSort có thể dùng được khi dữ liệu lớn và có tính ngẫu nhiên cao. Một điểm trừ của nó là khó cài đặt.

Trong các thuật toán ổn định, thuật toán Counting sort có tốc độ nhanh nhất, Bubble Sort có tốc độ chậm nhất. Với các thuật toán không ổn định thì Flash Sort có tốc độ nhanh nhất và Selection Sort chậm nhất.

3.3 Thời gian chạy với bộ dữ liệu đã sắp xếp

Bảng thời gian chạy

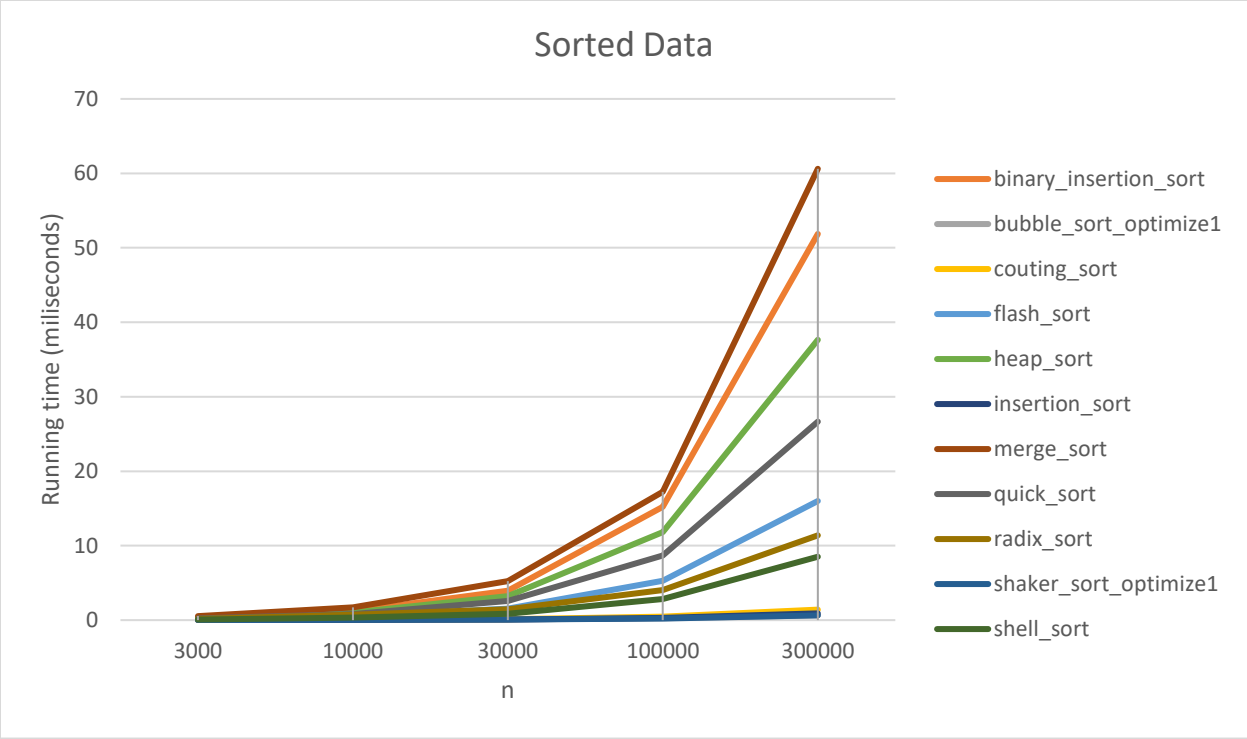
n	3000	10000	30000	100000	300000
binary_insertion_sort	0.3049	1.1979	3.9584	15.194	51.8683
bubble_sort	9.3751	103.5523	945.4694	10498.0885	94556.2277
bubble_sort_optimize1	0.0063	0.021	0.0623	0.2356	0.6369
couting_sort	0.0139	0.0985	0.1405	0.4685	1.4089
flash_sort	0.1538	0.5577	1.547	5.2871	15.9952
heap_sort	0.2737	1.0443	3.2226	11.8128	37.6452
insertion_sort	0.0127	0.0298	0.0892	0.3055	0.9078
merge_sort	0.537	1.7154	5.2519	17.242	60.6117
quick_sort	0.2298	0.7917	2.5861	8.6606	26.6561
radix_sort	0.1529	0.7359	1.4421	4.0386	11.3833
selection_sort	8.2168	91.3316	828.1996	9226.0128	83163.2184
selection_sort_optimize1	5.082	57.2673	510.6755	5707.7995	51489.0053
shaker_sort	9.4021	104.4242	955.0801	10564.075	95155.2853
shaker_sort_optimize1	0.0064	0.0205	0.0629	0.2106	0.6347
shell_sort	0.0744	0.3373	0.8357	2.834	8.5114



Đồ thị này có phần khó xem vì một số thuật toán với độ phức tạp $O(n^2)$ như Bubble Sort hay Shaker Sort có thời gian chạy quá lớn dù dữ liệu đã sắp xếp, làm ta khó nhìn thấy phần còn lại. Qua đây ta thấy trong các thuật toán sắp xếp cơ bản với độ phức tạp $O(n^2)$ thì Bubble sort là thuật toán tệ nhất, dù có tối ưu nó bằng Shaker Sort cũng không cải thiện tốc độ lắm, nhưng nếu ta dùng cải tiến rằng khi mảng đã được sắp xếp thì tốc độ sẽ rất nhanh.

Với thuật toán Selection Sort có thời gian chạy nhanh hơn Bubble Sort và Shaker Sort một chút, trong khi đó Selection Sort optimize 1 có thời gian chạy khoảng 2/3 so với Selection Sort, đây quả là một cải tiến rất giá trị mặc dù cách cài đặt không thay đổi nhiều.

Tôi sẽ loại bỏ đi các thuật toán Bubble Sort, Shaker Sort, Selection Sort, Selection Sort optimie 1 trong phần biểu đồ sau nhằm xem chi tiết hơn các thuật toán khác.



Những thuật toán còn lại chủ yếu đều có độ phức tạp $O(nlogn)$ hay $O(n)$ ở trường hợp mảng đã được sắp xếp. Trong đó chậm nhất chính là Merge Sort với độ phức tạp $O(nlogn)$. Tiếp theo chính là Binary-Insertion Sort, thuật toán này chạy nhanh hơn Merge Sort một chút bởi vì có hằng số lập trình thấp hơn Merge sort nhiều.

Tiếp theo là Heap Sort và Quick Sort, cũng giống với dữ liệu ngẫu nhiên, thuật toán Heap Sort chạy nhanh hơn Merge Sort nhưng chậm hơn Quick Sort.

Sau đó chính là Flash Sort, ta thấy với dữ liệu ngẫu nhiên thì Radix Sort sẽ chạy chậm hơn Flash Sort nhưng ở dữ liệu đã sắp xếp thì Radix Sort sẽ nhỉnh hơn Flash Sort một chút.

Radix Sort và Flash Sort cũng là 2 thuật toán cuối cùng có độ phức tạp $O(nlogn)$ và $O(nlogk)$ trước khi ta tới với những thuật toán có độ phức tạp $O(n)$ trong trường hợp này.

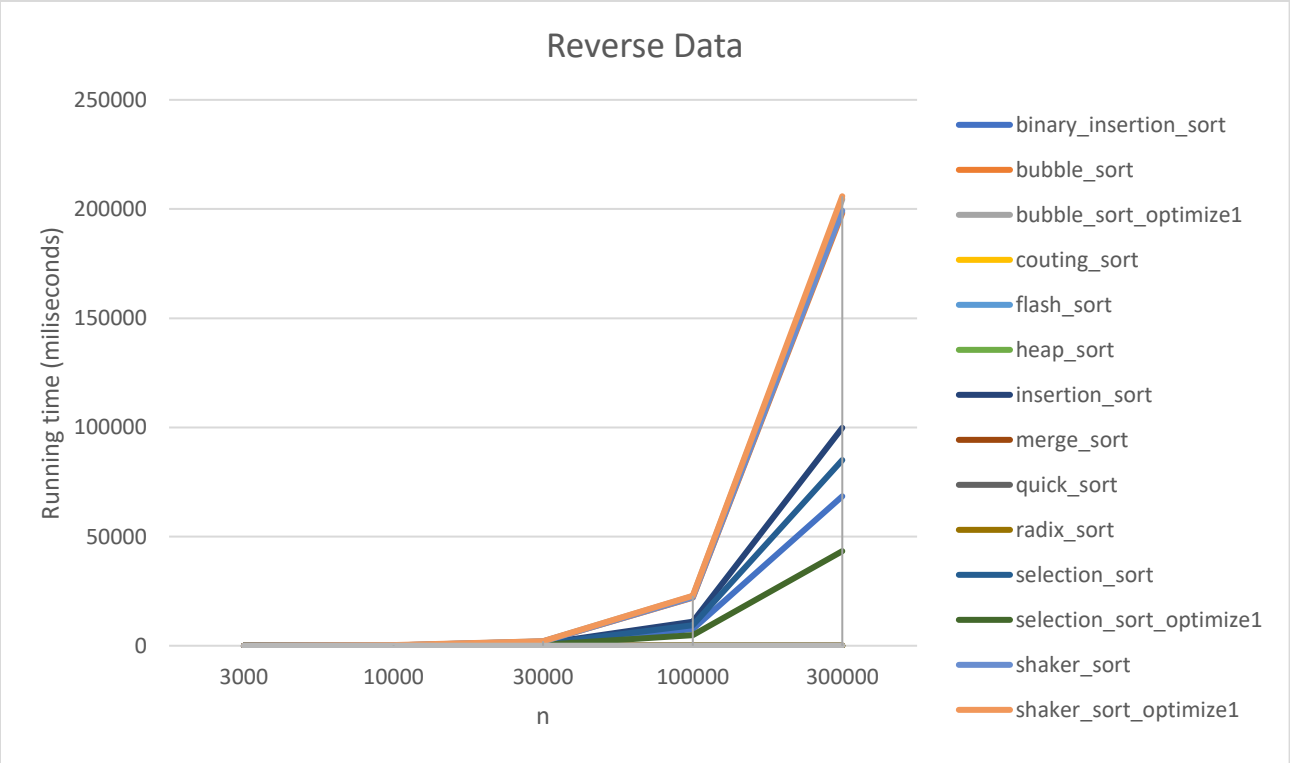
Thuật toán chậm nhất trong các thuật toán còn lại chính là thuật toán Shell Sort, mặc dù không tốn bất kì phép chèn nào nhưng Shell Sort chạy lâu hơn nhiều so với các thuật toán $O(n)$ khác đó là vì Shell Sort phải duyệt qua mảng nhiều lần.

Còn lại là các thuật toán Counting Sort, Bubble Sort Optimize 1, Insertion Sort, Shaker Sort Optimize1 đều có tốc độ không quá chênh lệch nhau.

3.4 Thời gian chạy với bộ dữ liệu sắp xếp ngược

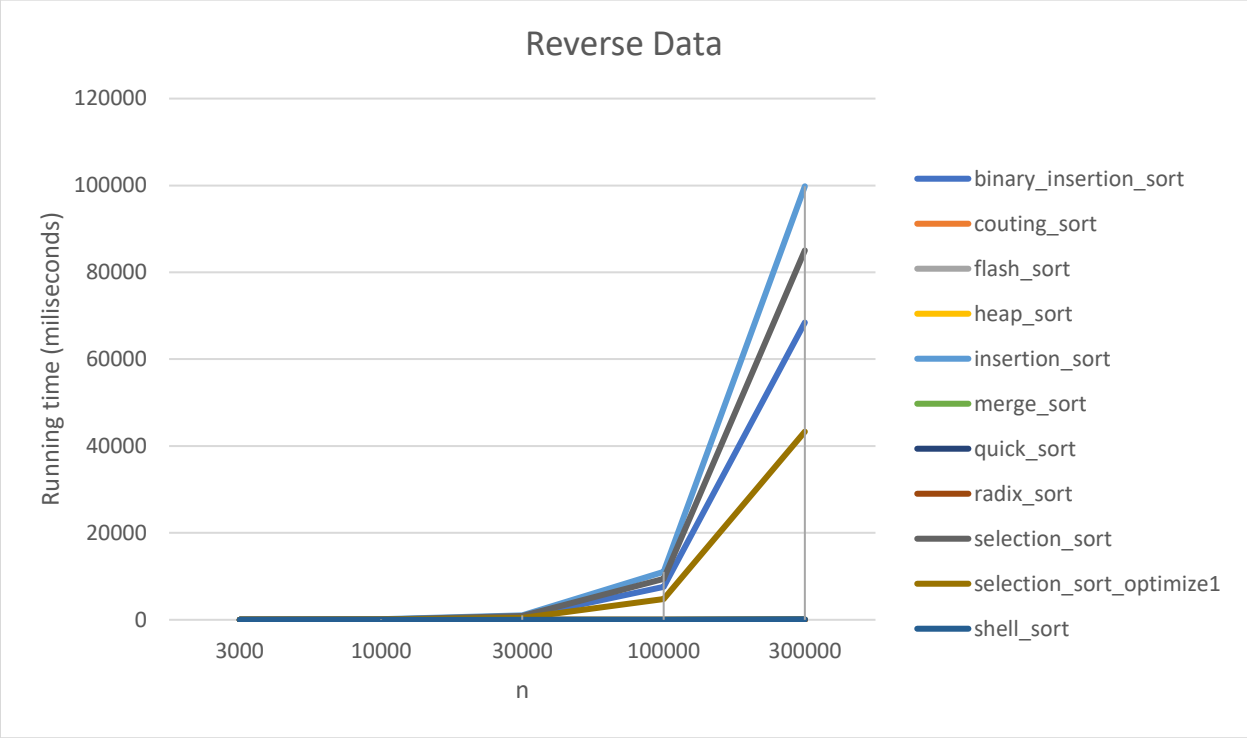
Bảng thời gian chạy

n	3000	10000	30000	100000	300000
binary_insertion_sort	7.0634	75.7567	685.1251	7580.7619	68446.748
bubble_sort	19.3244	216.2287	1971.2866	22046.0146	198140.0733
bubble_sort_optimize1	20.0365	223.4281	2043.3487	22671.4783	204460.7962
couting_sort	0.0146	0.0475	0.1408	0.5788	1.7736
flash_sort	0.1323	0.4432	1.3256	4.4227	13.609
heap_sort	0.2686	0.9956	3.2013	11.6675	37.6676
insertion_sort	9.7241	109.9858	987.9799	11069.9714	99803.3046
merge_sort	0.6683	2.0351	6.0133	20.1755	61.5252
quick_sort	0.2545	0.8229	2.4592	8.833	27.0834
radix_sort	0.0873	0.3401	1.02	3.6129	11.4525
selection_sort	8.4407	94.9947	850.1531	9449.8258	85038.9836
selection_sort_optimize1	4.2525	47.5694	439.4796	4814.6857	43318.2947
shaker_sort	19.7438	219.491	1985.3185	22099.0394	199228.314
shaker_sort_optimize1	20.8997	235.0072	2057.0751	22803.776	205858.9101
shell_sort	0.1188	0.4574	1.8611	11.1159	102.9577



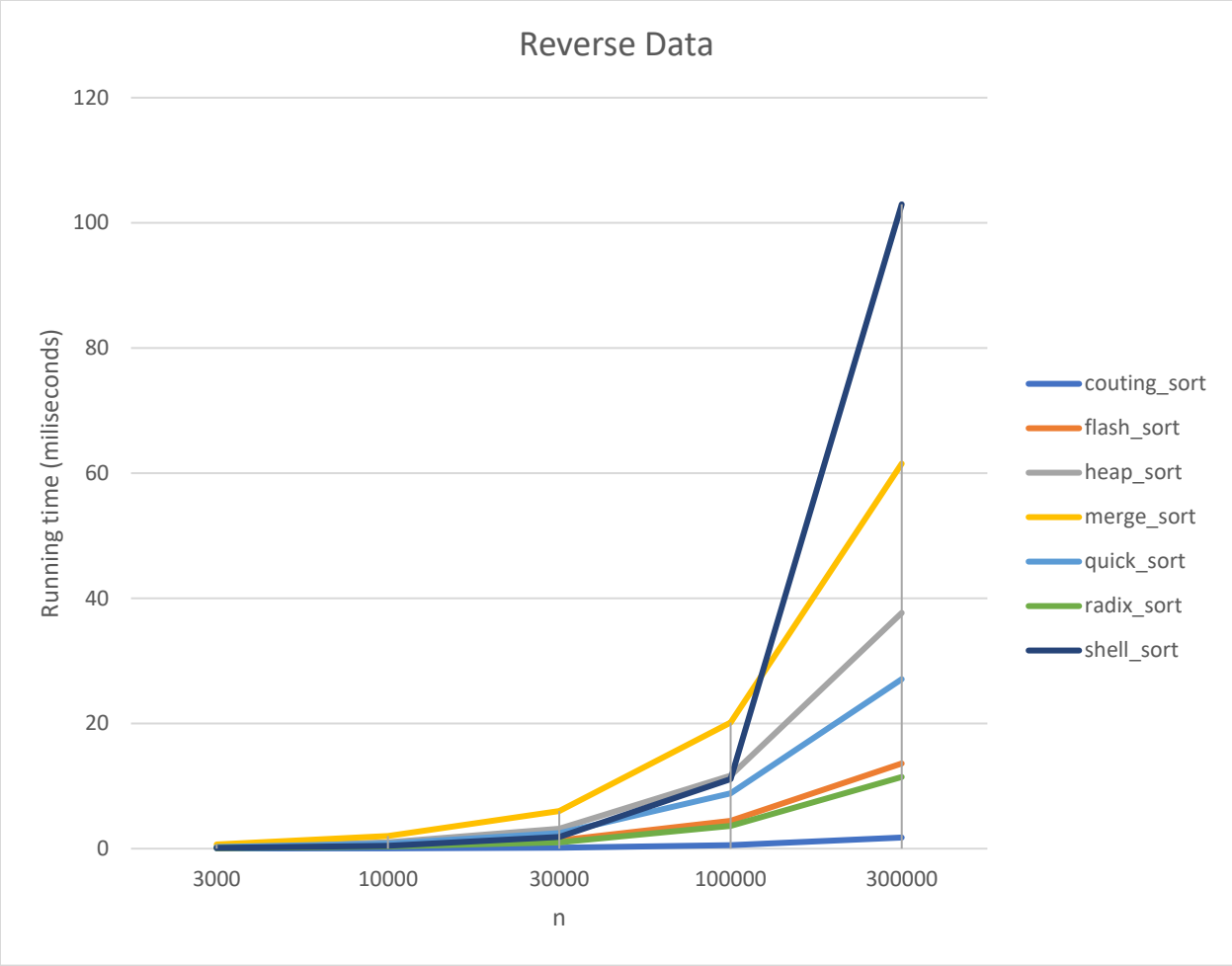
Vẫn dẫn đầu các thuật toán chạy chậm nhất đó chính là Bubble Sort và Shaker Sort, đây cũng là trường hợp khiến cho hai thuật toán này sử dụng nhiều phép hoán vị nhất, đó đó ta thấy thời gian chạy của 2 thuật toán này gấp đôi so với Selection Sort. Một điều đặc biệt đó là cải tiến của cả 2 thuật toán này đều chạy lâu hơn so với khi không được cải tiến.

Tôi sẽ loại bỏ đi thuật toán Bubble Sort và Shaker Sort này (cũng như những tối ưu của nó) trong phần biểu đồ sau nhằm xem chi tiết hơn các thuật toán khác.



Qua đồ thị này ta có thể thấy, thuật toán Insertion Sort chạy lâu nhất, lâu hơn thuật toán Selection Sort. Tiếp theo đó là thuật toán Binary Insertion Sort. Cải tiến của thuật toán Selection Sort là Selection Sort Optimize 1 giảm thời gian chạy đi khoảng một nửa, một cải tiến đáng quan tâm.

Tôi tiếp tục loại bỏ thuật toán trên để tiện quan sát:



Trong các thuật toán còn lại, đa số đều là các thuật toán có độ phức tạp $O(nlogn)$ hoặc tuyến tính. Ở thuật toán Shell Sort ta thấy có một sự nhảy vọt thời gian chạy từ $n = 100\ 000$ và $n = 300\ 000$. Điều này theo tôi nghĩ đó là do dãy **gaps** mà tôi chọn phù hợp với kích thước mảng nhỏ.

Giống với những dữ liệu khác, 3 thuật toán Merge Sort, Heap Sort và Quick Sort vẫn giữ đúng thứ tự thời gian chạy.

Với dữ liệu này, ta thấy thuật toán Radix Sort có tốc độ chạy nhanh hơn thuật toán Flash Sort một chút.

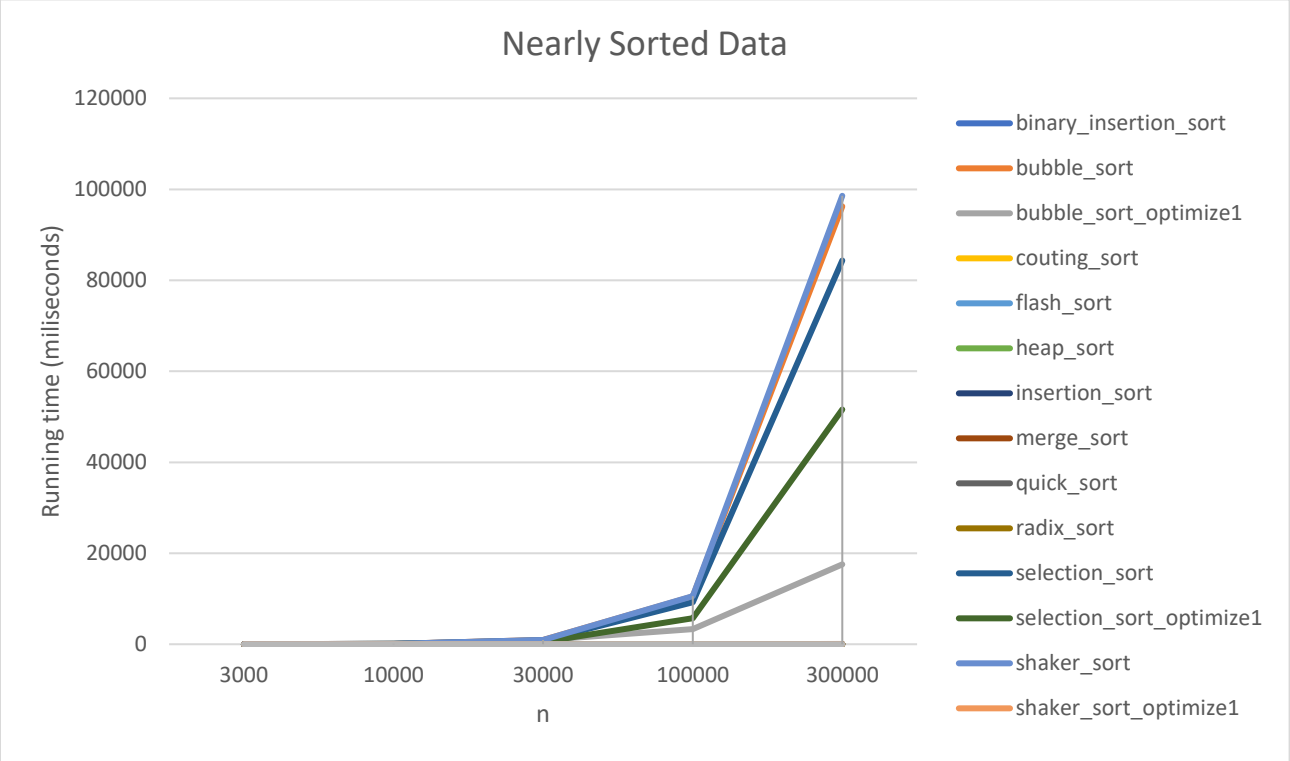
Thuật toán nhanh nhất vẫn là Counting Sort.

3.5 Thời gian chạy với bộ dữ liệu gần được sắp xếp

Bảng thời gian chạy

sort	3000	10000	30000	100000	300000
binary_insertion_sort	0.3737	1.3818	4.5556	15.7922	52.3756

bubble_sort	9.4183	104.0736	945.5142	10480.3164	96246.7829
bubble_sort_optimize1	8.0653	83.9896	914.4556	3343.7823	17548.452
couting_sort	0.0144	0.0462	0.1424	0.4729	1.4725
flash_sort	0.1711	0.5202	1.5505	5.1865	15.7739
heap_sort	0.2761	1.0103	3.2286	11.9746	37.7667
insertion_sort	0.0421	0.15	0.4991	0.6604	1.9701
merge_sort	0.6236	1.8125	5.3858	17.5332	56.9513
quick_sort	0.232	0.8046	2.4427	8.5029	26.5956
radix_sort	0.0887	0.3321	0.9954	3.6829	11.4399
selection_sort	8.7651	92.0205	827.0627	9235.1915	84352.3961
selection_sort_optimize1	5.0594	55.1876	507.1752	5698.1522	51595.3187
shaker_sort	9.4238	107.5545	949.2447	10559.4304	98573.5853
shaker_sort_optimize1	0.1042	0.4015	1.3023	3.228	9.1263
shell_sort	0.1464	0.4764	1.5458	3.4942	9.4775

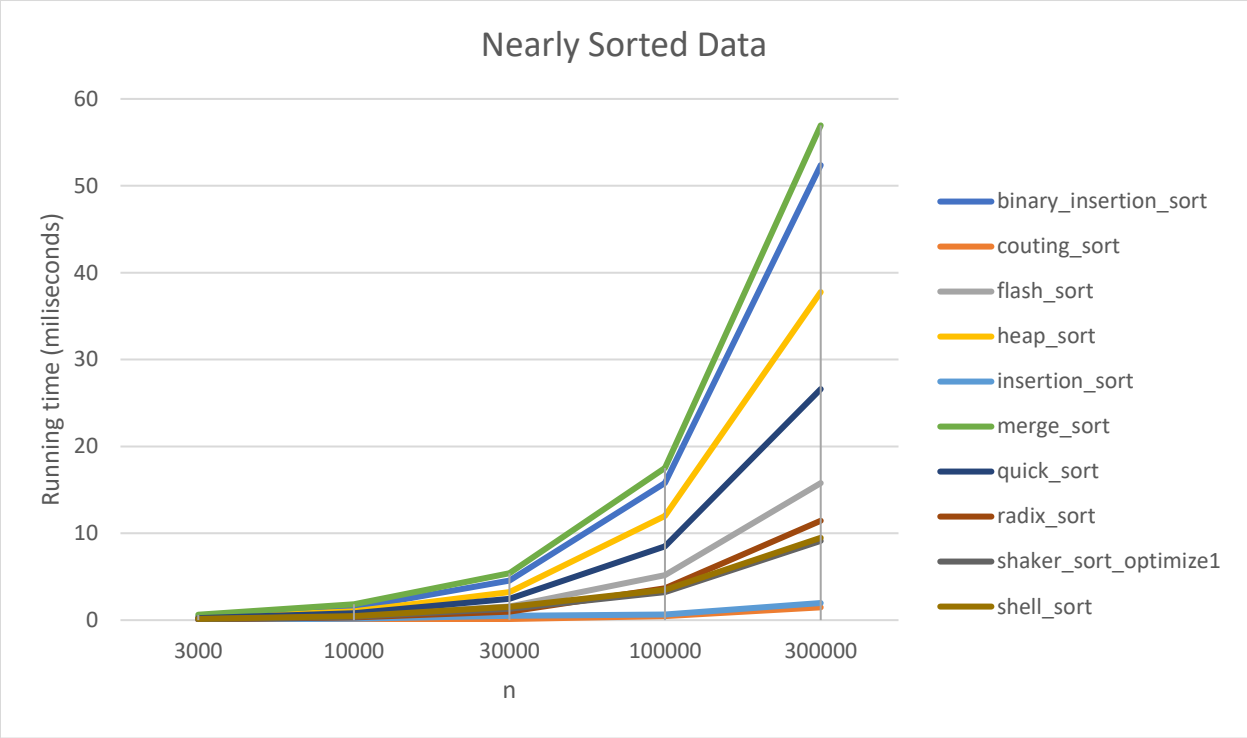


Vẫn dẫn đầu các thuật toán chạy chậm nhất đó chính là Bubble Sort và Shaker Sort chưa được tối ưu. Nhưng khác với trường hợp dãy được sắp xếp ngược, ở trường hợp này thì tốc độ của hai thuật toán này không lớn hơn quá nhiều so với Selection Sort (cách nhau khoảng 10s), còn ở trường hợp trước thì gấp đôi.

Nhanh hơn một chút đó là Selection Sort và cải tiến của nó. Cải tiến của Selection Sort vẫn tiếp tục cho thấy sự hiệu quả với việc giảm thời gian chạy đi gần một nửa.

Theo sau đó là Bubble Sort Optimize 1, ta thấy với việc cải tiến là khi mảng được sắp xếp thì sẽ ngừng việc duyệt đã giảm đáng kể thời gian chạy của thuật toán Bubble Sort.

Tiếp theo, tôi sẽ loại bỏ thuật toán Bubble Sort, Bubble Sort Optimize 1, Shaker Sort, Selection Sort và tối ưu của Section Sort khỏi biểu đồ nằm xem chi tiết hơn:



Trong các thuật toán còn lại, đa số đều là các thuật toán có độ phức tạp $O(nlogn)$ hoặc tuyến tính.

Vẫn như những dữ liệu trước, Merge Sort vẫn chạy chậm nhất khi so với Heap Sort và Quick Sort.

Nhanh hơn Merge Sort một chút là thuật toán Binary Insertion Sort, ở trường hợp này thuật toán có độ phức tạp xấp xĩ $O(nlogn)$ và chạy nhanh hơn Merge Sort do hằng số lập trình thấp hơn.

Sau đó vẫn là Heap Sort rồi tới Quick Sort. Ta có thể thấy trong mọi trường hợp, nếu Quick Sort được cài đặt tốt thì luôn chạy nhanh hơn Heap Sort.

Tôi sẽ loại bỏ các thuật toán kể trên trong biểu đồ sau, nhằm tiện quan sát các thuật toán có thời gian chạy thấp.

Trong các thuật toán còn lại, chậm nhất là Flash Sort, dù có độ phức tạp $O(n)$ trong trường hợp này nhưng với hằng số lập trình cao nên Flash Sort chậm hơn các thuật toán khác.

Nhanh hơn một chút đó là thuật toán Radix Sort.

Hai thuật toán có thời gian chạy xấp xĩ nhau đó là Shell Sort và Shaker Sort Optimize 1.

Ở trường hợp này, thuật toán Insertion Sort có tốc độ chạy nhanh hơn thuật toán Shell Sort. Nó có tốc độ gần như tương đương với Counting Sort.

4 Tổng kết

Các nhận xét dưới đây dựa phần lớn vào thời gian chạy ở trường hợp dữ liệu được sinh ngẫu nhiên.

4.1 Các thuật toán $O(n^2)$

Các thuật toán có độ phức tạp $O(n^2)$ ở trường hợp trung bình gồm các thuật toán: Bubble Sort, Bubble Sort Optimize 1, Shaker Sort, Shaker Sort Optimize 1, Selection Sort, Section Sort Optimize 1, Insertion Sort, Binary Insertion Sort.

Với các thuật toán này, Bubble Sort và Bubble Sort Opitmize có tốc độ chậm nhất, chậm hơn hẳn so với các thuật toán khác.

Với cách cài đặt đơn giản, Insertion Sort và Selection Sort có thời gian chạy tốt hơn nhiều so với Bubble Sort và Shaker Sort. Nếu ta cần sắp xếp với số lượng phần tử ít thì Insertion Sort là một sự lựa chọn hợp lý bởi tính dễ cài đặt của nó. Tuy Binary Insertion Sort có tốc độ nhanh hơn Insertion Sort nhưng cài đặt có phần phức tạp hơn.

4.2 Các thuật toán $O(nlogn)$ hay $O(nlogk)$

Các thuật toán có độ phức tạp $O(nlogn)$ ở trường hợp trung bình gồm có Merge Sort, Heap Sort, Quick Sort. Ở ba thuật toán này, thuật toán Merge Sort có tốc độ chạy chậm nhất và cần dùng thêm $O(n)$ bộ nhớ ngoài. Thuật toán Heap Sort có tốc độ chạy nhanh hơn Merge Sort nhưng chậm hơn Quick Sort trong đa số trường hợp. Tuy vậy, thuật toán Heap Sort luôn đảm bảo có độ phức tạp $O(nlogn)$ trong khi trường hợp xấu nhất thì độ phức tạp của Quick Sort có thể lên tới $O(n^2)$, mặc dù trường hợp này hiếm khi xảy ra khi ta cài đặt Quick Sort với cách chọn phần tử chốt ngẫu nhiên.

Một thuật toán dù khó đánh giá độ phức tạp nhưng thời gian chạy cũng rất tốt đó là Shell Sort, với thời gian chạy thường là nhanh hơn Merge Sort nhưng chậm hơn Heap Sort.

Thuật toán Radix Sort có độ phức tạp $O(nlogk)$, so về tốc độ chạy thì nhanh hơn hẳn 3 thuật toán kể trên, tuy nhiên thuật toán Radix Sort dựa vào cơ số của phần tử nên sẽ gặp chút khó khăn nếu các phần tử không phải số nguyên (số thực, xâu, các kiểu dữ liệu trừu tượng).

4.3 Các thuật toán $O(n)$

Lớp thuật toán này gồm có Flash Sort với độ phức tạp $O(n+m)$ (với m là số phân lớp), Counting Sort với độ phức tạp $O(n+k)$ (với k là phần tử lớn nhất).

Cả hai thuật toán này đều cần dùng thêm bộ nhớ ngoài. Khi giới hạn của các phần tử nhỏ, việc lựa chọn thuật toán Counting Sort sẽ phù hợp hơn Flash Sort bởi vì dễ cài đặt, hằng số lập trình thấp. Tuy nhiên điểm yếu của Counting Sort đó là không sắp xếp được khi các

phần tử có giới hạn quá lớn và Flash Sort đã giải quyết được vấn đề này bằng cách phân hoạch các phần tử ra các nhóm và dùng Insertion Sort ở mỗi nhóm.

4.4 Các thuật toán dựa vào so sánh

Đa số các thuật toán tôi trình bày đều dựa vào việc so sánh các phần tử: Bubble Sort, Shaker Sort, Insertion Sort, Binary Insertion Sort, Selection Sort, Shell Sort, Heap Sort, Merge Sort, Quick Sort.

Với các thuật toán này ta thấy có một điểm chung là độ phức tạp của chúng không thể tốt hơn $O(nlogn)$ [4]

Với những thuật toán này, thuật toán Quick Sort có thời gian chạy nhanh nhất trong đa số trường hợp, nhưng thuật toán Heap Sort sẽ đảm bảo được thời gian chạy là $O(nlogn)$ trong trường hợp tệ nhất.

4.5 Các thuật toán không dựa vào so sánh

Với các thuật toán không dựa vào so sánh, ta có thuật toán Counting Sort dựa vào phân phối dữ liệu, Radix Sort dựa vào cơ số của dữ liệu, Flash Sort dựa một phần vào phân phối và so sánh.

Điểm chung của các thuật toán này là nó có thể có độ phức tạp tốt hơn $O(nlogn)$, tuy nhiên sẽ khó sắp xếp nếu kiểu dữ liệu không phải là số (xâu, kiểu dữ liệu trừu tượng).

4.6 Các thuật toán ổn định

Trong cách cài đặt của tôi, những thuật toán ổn định bao gồm: Insertion Sort, Binary Insertion Sort, Bubble Sort, Shaker Sort, Merge Sort, Counting Sort.

Với các thuật toán này, Counting Sort có thời gian chạy cũng như độ phức tạp thấp nhất, nhưng theo tôi Merge Sort sẽ được dùng nhiều hơn bởi vì không phụ thuộc vào phân phối của dữ liệu.

4.7 Các thuật toán không ổn định

Trong cách cài đặt của tôi, những thuật toán không ổn định gồm: Selection Sort, Shell Sort, Heap Sort, Quick Sort, Radix Sort, Flash Sort. Trong các thuật toán này, thuật toán chạy nhanh nhất vẫn là Flash Sort.

5 Tài liệu tham khảo

[1] M. Ciura, "Best Increments for the Average Case," in *13th International Symposium: Fundamentals of Computation Theory*, London, 2001.

[2] Ryan Hayward, Colin McDiarmid, "Average Case Analysis of Heap Building by Repeated Insertion," *Journal of Algorithms,* vol. 12, p. 126, 1991.

[3] Neubert, Karl-Dietrich, "The Flashsort Algorithm," *Dr. Dobb's Journal,* p. 123, 1998.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, Introduction To Algorithms, Cambridge: The MIT Press, 2009, p. 167.