VNU HCMC-UNIVERSITY OF SCIENCE
FACULTY OF INFORMATION TECHNOLOGY

REPORT
SORTING ALGORITHMS

COURSE: DATA STRUCTURE AND ALGORITHM

21120506 - Nguyễn Thị Kiều Ngân

21120508 - Đặng An Nguyên

21120529 - Nguyễn Gia Phúc

21120530 - Nguyễn Hoàng Phúc

Class: 21CTT5

Mục lục

I Một số lưu ý

1 Cấu hình máy

- CPU: Intel Core I3 3240 @ 3.40Hz
- RAM: 16 GB
- OS: Windows 11 64bit

2 Các ký hiệu

- Nếu không nói gì thêm thì ta quy ước như sau:

- n: Số lượng phần tử của mảng.
- i: Chỉ số của một phần tử trong mảng.
- a: Mảng các phần tử.
- a[i]: Giá trị phần tử tại vị trí i của mảng a.

3 Giới hạn

- Giới hạn số phần tử:

$$10^4 <= n <= 5 * 10^5$$

- Giới hạn giá trị:

$$0 <= a[i] <= n$$

4 Một số lưu ý về cài đặt code

- Số phép so sánh được đếm cả trong vòng lặp. Tức là vị trí nào có phép so sánh thì đếm.

- Class đếm thời gian tôi cài đặt ban đầu là đếm theo đơn vị nanosecond, khi in ra màn hình hoặc ghi vào file tôi chuyển sang đơn vị milisecond.

- Để code được gọn hơn, tôi sử dụng kỹ thuật con trỏ hàm, trỏ đến 11 hàm sort.

- Dãy số được chọn trong thuật toán Shell sort là dãy số của Marcin Ciura đề xuất năm 2001.

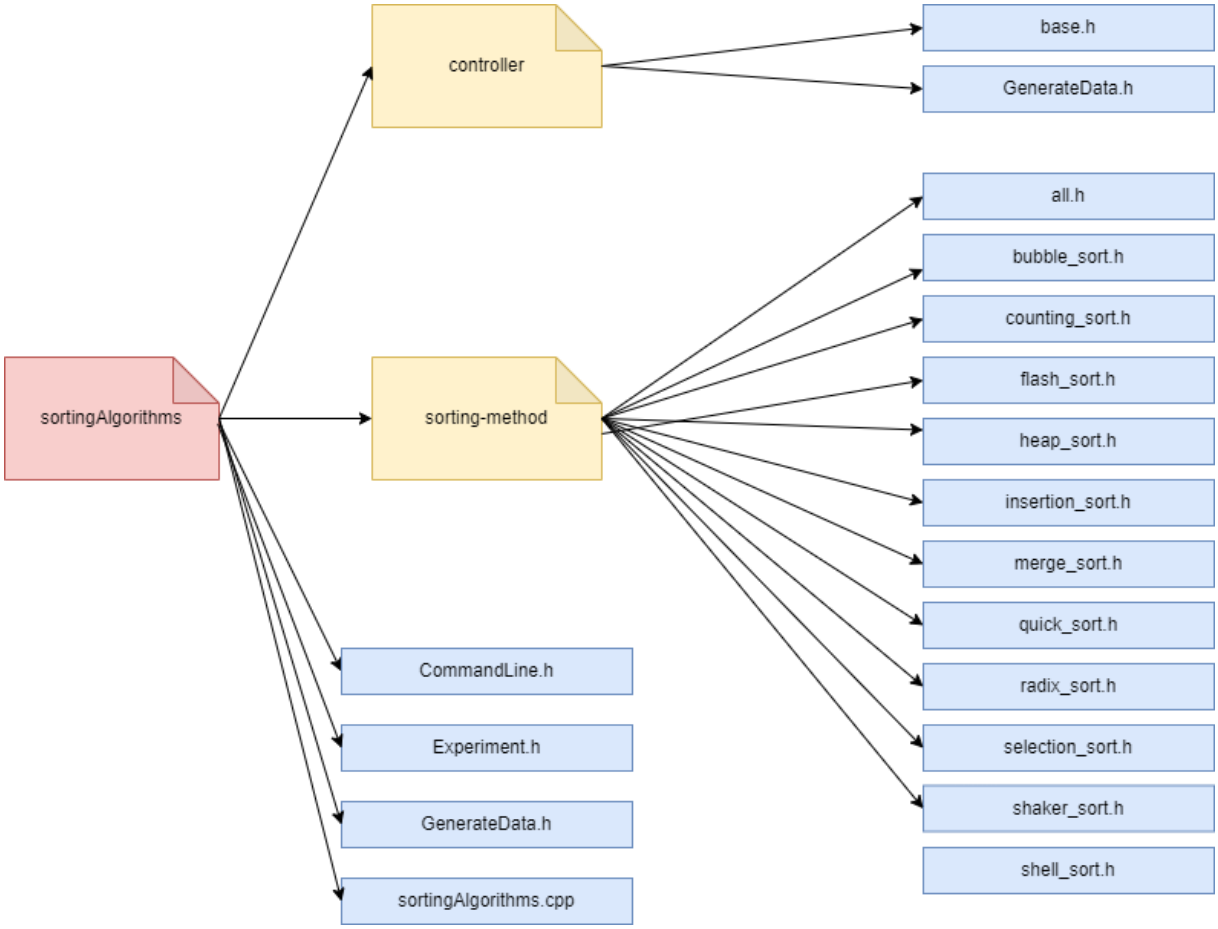
- Các thư viện đã được sử dụng:

- iostream: cin, cout, các câu lệnh cơ bản.

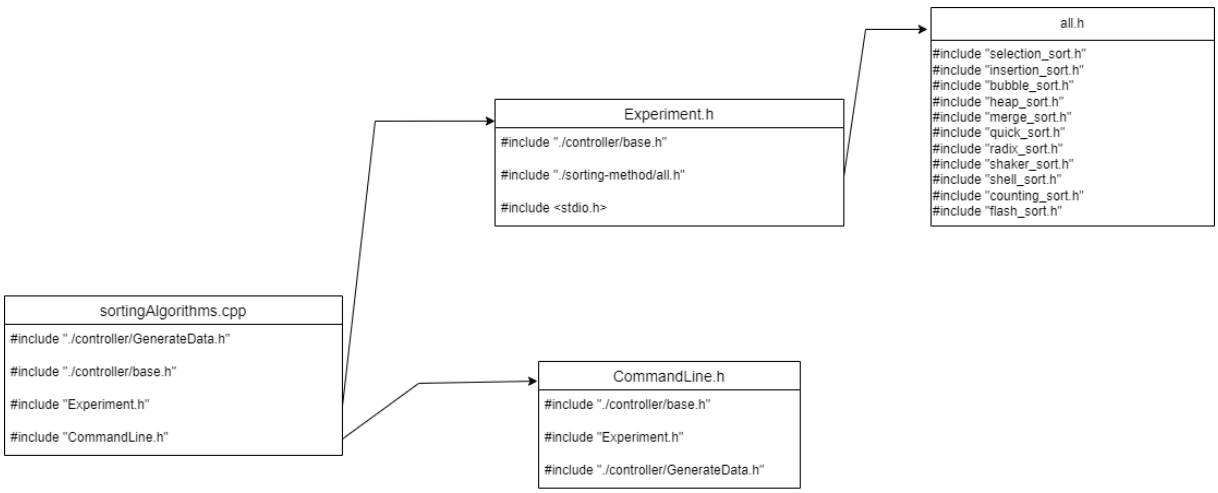
- chrono: tính thời gian chạy.
- stdio.h: định dạng stdout để thuận tiện ghi file csv trong lúc thực nghiệm.
- fstream: Đọc và ghi file txt.
- string: xử lý xâu.

II Tổ chức project

Tổ chức file



Tổ chức chương trình



- Mỗi thuật toán sẽ nằm trong một file .h. Kết nối toàn bộ thuật toán lại bằng file all.h.

- Thư mục sorting-method: chứa mã nguồn các thuật toán sắp xếp.
- Thư mục controller chứa:
  - base.h: Chứa các hàm phụ cho việc cài đặt thuật toán và class tính thời gian.
  - GenerateData.h: Chứa các hàm dùng để tạo dữ liệu.

III Các thuật toán đã cài đặt

1 Shell-sort

1.1 Ý tưởng thuật toán

Thuật toán Shell Sort được coi là một cải tiến so với Selection Sort. Nếu như Selection Sort so sánh các phần tử ở kề nhau thì Shell Sort cho phép so sánh và hoán vị các phần tử ở xa nhau.

Ban đầu ta sẽ có một dãy số được gọi là các "gaps". Các "gaps" này có ý nghĩa như là một khoảng cách giữa 2 phần tử cần sắp xếp mỗi lần duyệt. Mỗi lần duyệt ta sẽ duyệt qua các "gaps" theo thứ tự giảm dần cho đến khi hết dãy "gaps".

1.2 Các bước hoạt động

Ví dụ, ban đầu ta có mảng như sau:

$$A = \{5, 1, 2, 6, 7\}$$

Và các gaps là:

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

Các bước thực hiện thuật toán như sau:

Gaps	Mảng	Ghi chú
3	{5, 1, 2, 6, 7}	Với gaps = 3. Ta so sánh 5 và 6 thấy đã đúng vị trí nên ta bỏ qua.
3	{5, 1, 2, 6, 7}	Tương tự 1 và 7 cũng đã đúng vị trí nên ta bỏ qua.
2	{5, 1, 2, 6, 7}	Với gaps = 2. Ta so sánh thấy 5 và 2 đứng sai vị trí nên ta hoán vị nó và tiếp tục vòng lặp.
2	{2, 1, 5, 6, 7}	1 và 6 đã đúng vị trí nên ta bỏ qua.
2	{2, 1, 5, 6, 7}	5 và 7 đã đúng vị trí nên ta bỏ qua.
2	{2, 1, 5, 6, 7}	Tương tự với 2 và 5 đúng vị trí nên ta tiếp tục với gaps = 1
1	{2, 1, 5, 6, 7}	Với gaps = 1 thì 2 và 1 đứng sai vị trí nên ta hoán vị 2 và 1.
1	{1, 2, 5, 6, 7}	2 và 5 đã đứng đúng vị trí nên ta bỏ qua.
1	{1, 2, 5, 6, 7}	5 và 6 đã đứng đúng vị trí nên ta bỏ qua.
1	{1, 2, 5, 6, 7}	6 và 7 đã đứng đúng vị trí nên ta bỏ qua.
1	{1, 2, 5, 6, 7}	5 và 6 đã đứng đúng vị trí nên ta bỏ qua. Tới đây ta cũng kết thúc việc duyệt mảng với gap=1 và kết thúc thuật toán.

1.3 Phân tích thuật toán

Độ phức tạp thuật toán

- Độ phức tạp thuật toán phụ thuộc phần lớp vào cách ta chọn "gaps". Việc đánh giá độ phức tạp chỉ mang tính tương đối như sau:

- Trường hợp tốt nhất: $O(nlogn)$
- Trường hợp xấu nhất: $O(n^2)$
- Trường hợp trung bình: $O(nlog(n)^2)$

Độ phức tạp không gian

- $O(1+m)$ với m là số phần tử của "gaps"

2 Selection-sort

2.1 Ý tưởng thuật toán

Tìm kiếm vị trí của phần tử nhỏ nhất trong trong mảng n phần tử ban đầu, đưa phần tử này về đúng vị trí của nó trong mảng là đầu mảng. Sau đó thực hiện tương tự với n – 1 phần tử còn lại trong mảng cho tới khi chỉ còn 1 phần tử thì ta được mảng đã được sắp xếp.

2.2 Các bước hoạt động

Ví dụ ta có mảng đầu vào như sau:

$$A = \{3,0,2,8,7\}$$

Giai đoạn	Mô tả	Giải thích
1	{|3, 0, 2, 8, 7}	Ban đầu mảng chưa được sắp xếp, ta tìm phần tử nhỏ nhất trong mảng a là 0 và đưa về đúng vị trí của nó, lúc này mảng chỉ còn lại 4 phần tử chưa được sắp xếp
2	{0|3, 2, 8, 7}	Tiếp tục tìm phần tử nhỏ nhất trong 4 phần tử chưa sắp xếp, đó là 2 và đưa về đúng vị trí của nó, lúc này mảng chỉ còn 3 phần tử chưa sắp xếp
3	{0, 2|3, 8, 7}	Tiếp tục tìm phần tử nhỏ nhất trong 3 phần tử chưa sắp xếp, đó là 3 và đưa về đúng vị trí của nó, lúc này mảng còn 2 phần tử chưa sắp xếp
4	{0, 2, 3|8,7}	Tiếp tục tìm phần tử nhỏ nhất trong 2 phần tử còn lại chưa được sắp xếp, đó là 7 và đưa 7 vè đúng vị trí của nó, lúc này mảng chỉ còn đúng 1 phần tử chưa được sắp xếp
5	{0, 2, 3, 7|8}	Do mảng chỉ còn đúng 1 phần tử chưa được sắp xếp thì phần tử này luôn đúng vị trí của nó => thuật toán dừng

2.3 Phân tích thuật toán

Độ phức tạp thuật toán Trong mọi trường hợp ta đều cần phải duyệt n - 1 lần để tìm giá trị nhỏ nhất của lần lặp 1, duyệt n - 2 lần để tìm giá trị nhỏ nhất của lần lặp 2, và cứ thế cho đến hết. Như vậy ta sẽ có tổng số phép so sánh là:

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

Do đó, độ phức tạp thuật toán sẽ là $O(n^2)$ cho mọi trường hợp.
Độ phức tạp không gian $O(1)$

2.4 Cải tiến

Thay vì chỉ tìm vị trí phần tử nhỏ nhất, ta sẽ đi tìm cả hai phần tử nhỏ nhất và lớn nhất ở mỗi vòng lặp rồi đưa về đúng vị trí của nó.

3 Shaker-sort

3.1 Ý tưởng thuật toán

Shaker sort là một thuật toán cải tiến của bubble sort. Với bubble sort, ta sẽ chỉ đưa phần tử về cuối mảng. Còn đối với shaker sort, sẽ có một lần duyệt từ đầu mảng tới cuối mảng để đưa phần tử về sau như bubble sort sau đó duyệt

từ cuối mảng về đầu mảng để đưa phần tử về phía trước, cứ thế lặp lại cho tới khi mảng đã được sắp xếp.

3.2 Các bước hoạt động

Ví dụ, ban đầu ta có mảng như sau:

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

Các bước thực hiện thuật toán như sau:

Lần lặp	Mô tả	Giải thích
1	{|1, 0, 4, 3, 2|}	1 và 0 đang ở sai vị trí, ta sẽ hoán đổi vị trí chúng.
1	{|0, 1, 4, 3, 2|}	1 và 4 đang ở đúng vị trí, bỏ qua.
1	{|0, 1, 4, 3, 2|}	4 và 3 đang ở sai vị trí, ta sẽ hoán đổi vị trí chúng.
1	{|0, 1, 3, 4, 2|}	4 và 2 đang ở sai vị trí, ta sẽ hoán đổi vị trí chúng.
1	{|0, 1, 3, 2|4}	4 đã ở đúng vị trí cần được sắp xếp. Bắt đầu duyệt theo thứ tự ngược lại.
1	{|0, 1, 3, 2|4}	3 và 2 đang ở sai vị trí, ta sẽ hoán đổi vị trí chúng.
1	{|0, 1, 2, 3|4}	1 và 2 đang ở đúng vị trí, bỏ qua.
1	{|0, 1, 2, 3|4}	0 và 1 đang ở đúng vị trí, bỏ qua.
1	{0|1, 2, 3|4}	0 đã ở đúng vị trí cần được sắp xếp. Kết thúc lần lặp thứ 1.
2	{0|1, 2, 3|4}	1 và 2 đang ở đúng vị trí, bỏ qua.
2	{0|1, 2, 3|4}	2 và 3 đang ở đúng vị trí, bỏ qua.
2	{0|1, 2|3, 4}	3 đã ở đúng vị trí cần được sắp xếp. Bắt đầu duyệt theo thứ tự ngược lại.
2	{0|1, 2|3, 4}	1 và 2 đang ở đúng vị trí, bỏ qua.
2	{0, 1|2|3, 4}	1 đã ở đúng vị trí cần được sắp xếp. Kết thúc lần lặp thứ 2.
3	{0, 1|2|3, 4}	2 đã ở đúng vị trí cần được sắp xếp.
3	{0, 1||2, 3, 4}	Mảng đã được sắp xếp, thuật toán dừng lại.

3.3 Phân tích thuật toán

Độ phức tạp thuật toán

- Trường hợp tốt nhất: $O(n)$
- Trường hợp xấu nhất: $O(n^2)$
- Trường hợp trung bình: $O(n^2)$

Độ phức tạp không gian

- $O(1)$

4 Counting-sort

4.1 Ý tưởng thuật toán

Thuật toán duyệt qua tất cả các phần tử và đếm số lần xuất hiện của chúng. Từ mảng đếm này ta có thể đưa ra mảng đã sắp xếp. Vị trí phần tử của mảng

đếm là phần tử trong mảng ban đầu, giá trị tại đó là số lần xuất hiện của chúng. Bằng các gán lại cho mảng ban đầu lần lượt theo tứ tự tăng dần của mảng đếm với số lần xuất hiện bằng giá trị tại đó (lớn hơn 0), ta được mảng đã sắp xếp.

4.2 Các bước hoạt động

Ví dụ, ban đầu ta có mảng như sau:

$$A = \{1,0,4,3,1\}$$

Ta có mảng đếm count[] như sau:
count[0] = 1; count[1] = 2; count [2] = 0; count[3] = 1; count [4] = 1;
Từ đây ta đưa ra được mảng được sắp xếp là:

$$A = \{0,1,1,3,4\}$$

4.3 Phân tích thuật toán

Độ phức tạp thuật toán

- Với n là số lượng phần tử, max phần tử lớn nhất và k là độ chênh lệch giữa phần tử lớn nhất và nhỏ nhất ta có độ phức tạp như sau:

- Best case: $O(n+k)$
- Wort-case: $O(n+k)$
- Average-case: $O(n+k)$

Độ phức tạp không gian

- $O(max)$

Cải tiến thuật toán

- Thuật toán ở trẻn không thể sắp xếp số âm và để tối ưu bộ nhớ, ta có thể khởi tạo mảng đếm count bắt đầu từ giá trị nhỏ nhất đến giá trị lớn nhất của mảng cần sắp xếp [0; max-min].

5 Insertion-sort

5.1 Ý tưởng tuật toán

Thuật toán Insertion sort thực hiện sắp xếp các phần tử theo cách duyệt từng phần tử. Và chèn từng phần tử đó vào đúng vị trí trong mảng con. Phần tử được chuyền vào vị trí thích hợp sao cho mảng con vẫn đảm bảo sắp xếp theo đúng thứ tự. Vị trí thích hợp chèn phần tử A[i] vào thỏa mãn a[k – 1]< a[i] <a[k] với $1 \leq k \leq i$

5.2 Các bước hoạt động

Ví dụ, ban đầu ta có mảng như sau

$$A = \{3, 0, 2, 8, 7\}$$

Giai đoạn	Mô tả	Giải thích
1	$*\|3, 0, 2, 8, 7$	Với mảng có 1 phần tử thì sẽ luôn được sắp xếp nên ta bắt đầu với vị trí thứ 2 là phần tử 0, ta duyệt từ đầu mảng tới vị trí trí thứ 1 ta thấy vị trí * là vị trí thích hợp => chèn 0 vào *
2	$0, *\|3, 2, 8, 7$	Xét tiếp vị trí tiếp theo là phần tử 2, duyệt từ đầu mảng tới vị trí đã được sắp xếp thì thấy * là vị trí thích hợp, chèn 2 vào *
3	$0, 2, *\|3, 8, 7$	Xét tiếp vị trí 3 thì thấy 3 đã ở vị trí thích hợp => không cần thay đổi vị trí
4	$0, 2, 3, *\|8, 7$	Xét tiếp vị trí thứ 4 thì thấy 8 đã ở vị trí thích hợp => không cần thay đổi vị trí
5	$0, 2, 3, *, 8\|7$	Xét tiếp vị trí thứ 5 thấy phần tử 7 chưa ở vị trí phù hợp, duyệt từ đầu đến vị trí thứ 4 ta thấy vị trí * là vị trí thích hợp, chèn 7 vào vị trí * => đã duyệt hết tất cả các phần tử trong mảng => thuật toán kết thúc

5.3 Phân tích thuật toán

Độ phức tạp thuật toán Trong mọi trường hợp ta đều phải duyệt qua n phần tử và tìm kiếm vị trí thích hợp cho mỗi phần tử. Ta thấy trong mỗi giai đoạn, ta tìm kiếm vị trí thích hợp cho một phần tử chưa được sắp xếp.

- Trong trường hợp tốt nhất vị trí thích hợp ở ngay vị trí đã sắp xếp thì có độ phức tạp O(n)
- Trong trường hợp xấu nhất vị trí thích hợp ở vị trí đầu của dãy đã được sắp xếp thì có độ phức tạp $O(n^2)$

- Trung bình: $O(n^2)$

Độ phức tạp không gian: $O(1)$

6 Merge-sort

6.1 Ý tưởng thuật toán

- Thuật toán merge sort là thuật toán áp dụng tư tưởng chia để trị vào giải bài toán, thuật toán được thực hiện với 2 bước đệ quy như sau:

+ Nếu mảng có ít hơn 2 phần tử => mảng đã được sắp xếp

+ Ngược lại ta tách mảng thành 2 phần với số lượng phần tử chênh lệch trong mảng không quá 1 phần tử, sắp xếp 2 phần đó và trộn lại với nhau

6.2 Các bước hoạt động

Ví dụ ban đầu ta có mảng như sau

$$A = \{3, 0, 2, 8, 7\}$$

Mảng A	Giải thích
[3, 0, 2, 8, 7]	Chia mảng thành 2 phần là [3, 0, 2] và [8, 7]
[3, 0, 2], [8, 7]	Chia mảng thành [3, 0, 2] thành 2 phần [3,0] và [2]
[[3, 0], [2]], [8, 7]	Chia mảng [3, 0] thành 2 phần [3], [0] và trộn lại thành [0, 3]
[[0, 3],[2]], [8,7]	Trộn [0, 3] và [2] thành [0, 2, 3]
[0, 2, 3], [8. 7]	Tách [8, 7] thành 2 phần [8], [7] và trộn lại thành [7, 8]
[0, 2, 3], [7, 8]	Trộn [0, 2, 3] và [7, 8] thành [0, 2, 3, 7, 8]
[0, 2, 3, 7, 8]	thuật toán dừng

6.3 Phân tích thuật toán

Độ phức tạp thuật toán

Gọi T(n) là thời gian thực hiện thi thuật toán merger sort

Ta có biểu thức đệ quy như sau: $T(n) = 2 * T(\frac{n}{2}) + n$. VÌ ta cần giải bài toán cho 2 mảng con có số lượng phần tử là $\frac{n}{2}$ và tốn n phép trộn 2 mảng con. Áp dụng kết quả của định lý master theorem thì ta có nghiệm là $O(nlogn)$.

Do đó độ phức tạp của thuật toán trong mọi trường hợp là $O(nlogn)$

Độ phức tạp không gian

Do tất cả phần tử trong mảng được sao chép vào một mảng phụ nên độ phức tạp không gian của thuật toán này là $O(n)$

7 Heap-sort

7.1 Ý tưởng thuật toán

Coi dãy cần sắp xếp là một cây nhị phân hoàn chỉnh, sau đó hiệu chỉnh cây thành cấu trúc max heap (node cha > 2 node con)

Dựa vào tính chất của cây heap ta có thể lấy được phần tử lớn nhất (gốc của cây heap)của dãy và đưa về cuối mảng, giảm phần tử của cây nhị phân và tái cấu trúc lại cây heap.

Thuật toán với 2 giai đoạn chính:

+ xây dựng max heap

+ lặp lại việc đem phần tử max heap ra sau mảng, sau đó điều chỉnh lại heap

7.2 Các bước hoạt động

Ví dụ ban đầu ta có mảng như sau:

$$A = \{3, 0, 2, 8, 7\}$$

Giai đoạn 1: Xây dựng max heap

Vị trí xét	Mô tả	Giải thích
1	$3, 8, 2, 0, 7$	Do các phần tử thuộc đoạn [n/2; n - 1] sẽ không có con nên ta có thể bỏ qua không cần xét, ta xét từ phần tử ở vị trí n/2 - 1, phần tử 0 có 2 con là 2 và 8, do ta đang xây dựng max heap nên ta hoán vị 0 và 8
0	$8, 3, 2, 0, 7$	Ta xét phần tử ở vị trí 0, phần tử này có 2 con là 8 và 2, ta hoán vị phần tử 3 và 8 để thỏa mãn tính chất của max heap
1	$8, 7, 2, 0, 3$	Do hiệu ứng lan truyền nên ta cần hiệu chỉnh lại phần tử thứ 1, ta hoán vị phần tử 3 và 7 để thỏa mãn tính chất của max heap

Giai đoạn 2:

Lần lặp	Mô tả	Giải thích
1	$3,7,2,0\|8$	Đầu tiên ta đưa phần tử 8 về cuối mảng và đem phần tử 3 lên thay thế phần tử 8
1	$7,3,2,0\|8$	Đẩy phần tử 3 xuống vị trí thích hợp để đảm bảo tính chất của max heap, lúc này phần cây heap ở đầu đã thỏa mãn tính chất của max heap.
2	$3,0,2\|7,8$	Đưa phần tử 7 về cuối mảng và đem phần tử 0 lên thay thế cho phần tử 7 lúc này phần cây heap chưa thỏa tính chất của max heap, điều chỉnh lại cây heap
3	$2,0\|3,7,8$	Đưa 3 về cuối mảng và đem phần tử 2 lên thay thế cho 3, lúc này phần cây heap đã thỏa tính chất của max heap
4	$0\|2,3,7,8$	Đưa 2 về cuối mảng, đưa phần tử 0 lên thay, phần tử 0 đã đúng vị trí => thuật toán kết thúc

7.3 Phân tích thuật toán

Độ phức tạp thuật toán Ở giai đoạn 1 độ phức tạp của thuật toán là $O(n)$ vì phải duyệt qua n phần tử của mảng

Ở giai đoạn 2 độ phức tạp của thuật toán là $O(nlogn)$ vì ta thực hiện n lần điều chỉnh cây heap, mỗi lần điều chỉnh sẽ tốn chi phí bằng chiều cao của cây nhị phân là $log(n)$

Do đó độ phức tạp của thuật toán là $O(n)+O(nlogn)=O(nlogn)$

+ Trong trường hợp trung bình $O(nlogn)$

+ Trong trường hợp tốt nhất $O(nlogn)$

+ Trong trường hợp xấu nhất $O(nlogn)$

Độ phức tạp không gian $O(1)$

8 Flash-sort

8.1 Ý tưởng thuật toán

Flash-sort là thuật toán sắp xếp dựa trên sự phân bố dữ liệu và các phép so sánh.

Flash-sort có 3 giai đoạn.

Giai đoạn thứ nhất: Classification. Tính toán số phân đoạn được sử dụng.

Giai đoạn thứ hai: Permutation. Phân hoạch phân tử thành m phân đoạn, phần tử a[i] sẽ nằm ở phân đoạn thứ: $1 + |(m-1) * \frac{a[i]-a_{min}}{a_{max}-a_{min}}$.

Giai đoạn thứ ba: Ordering. Sắp xếp các phần tử trong mỗi phân đoạn bằng sắp xếp chèn.

8.2 Các bước hoạt động

Ví dụ, ban đầu ta có mảng như sau:

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

Và $m = 3$ là số phân đoạn.

Phân hoạch các phần tử vào đúng phân đoạn của mình.

$$A = \{1,0|2,3|4\}$$

Dùng sắp xếp chèn sắp xếp lại các phần tử trong mỗi phân đoạn.

$$A = \{0,1|2,3|4\}$$

Dừng thuật toán.

8.3 Phân tích thuật toán

Độ phức tạp thuật toán

Giai đoạn thứ nhất: $O(n)$

Giai đoạn thứ hai: $O(n)$

Giai đoạn thứ ba: Mỗi phân đoạn sẽ có $|\frac{n}{m}|$ phần tử, để sắp xếp mỗi phân đoạn đó sẽ có độ phức tạp là $O((\frac{n}{m})^2)$. Và ta có đúng m đoạn, như vậy độ phức tạp sẽ là $O((\frac{n}{m})^2 * m) = O(\frac{n^2}{m})$. Theo thực nghiệm, ta chọn $m = 0.43n$. Vậy trường hợp trung bình sẽ có độ phức tạp là $O(n)$.

Trường hợp xấu nhất xảy ra khi các phần tử phân bố không đều làm tốn nhiều thời gian hơn ở giai đoạn sắp xếp. Khi đó độ phức tạp là $O(n^2)$

Độ phức tạp không gian $O(m)$

9 Quick-sort

9.1 Ý tưởng thuật toán

Thuật toán Quick-sort là một thuật toán chia để trị. Ý tưởng của thuật toán được thực hiện đệ quy như sau:

- Nếu mảng có ít hơn 1 phần tử thì không làm gì
- Chọn một phần tử làm mốc (hay còn gọi là pivot) và bắt đầu chia làm hai phần. Phần thứ nhất gồm các phần tử nhỏ hơn pivot sẽ nằm bên trái pivot, Phần thứ hai gồm các phần tử lớn hơn pivot và nằm bên phải pivot.

9.2 Các bước hoạt động

Giả sử ban đầu ta có một mảng như sau:

$$A = \{3,1,2,0,5\}$$

Chọn phần tử $pivot = 2$, dùng ký hiệu để đánh dấu hai phần.

Tại vòng lặp đầu tiên, số 3 và 0 đứng không đúng vị trí của mình nên sẽ hoán đổi cho nhau.

$$A = \{\{0,1\},2,\{3,5\}\}$$

Gọi đệ quy hai phần được chia ra. Chọn pivot của phần bên trái là 0 và bên phải là 3. Ta được:

$$A = \{\{0\},\{1\},2,\{3\},\{5\}\}$$

Tới đây, thuật toán dừng.

9.3 Phân tích thuật toán

Độ phức tạp thuật toán

Độ phức tạp của thuật toán Quick-sort dựa trên cách ta chọn phần tử pivot sao cho tốt. Hiện nay chưa có nghiên cứu nào cho thấy chọn pivot như nào cho tốt và phổ biến nhất là có 2 cách chọn pivot là phần tử ở giữa hoặc phần tử ngẫu nhiên.

Trường hợp tốt nhất: Khi ta chọn được pivot tốt, tức là sau mỗi lần phân hoạch ta được hai phần có số phần tử chênh lệch nhau rất ít. Trường hợp này độ phức tạp thuật toán sẽ là $O(nlog(n))$

Trường hợp xấu nhất: Khi ta chọn pivot không tốt, mỗi lần phân hoạch sẽ chỉ cho ra được hai phần với số phần tử là 1, n - 1. Khi đó số lần phân hoạch sẽ rất nhiều. Trường hợp này độ phức tạp thuật toán sẽ là $O(n^2)$.

Độ phức tạp không gian

Vì là thuật toán cài đặt bằng đệ quy nên sẽ tốn bộ nhớ stack.

Trường hợp tốt nhất: $O(logn)$

Trường hợp xấu nhất: $O(n)$

10 Bubble-sort

10.1 Ý tưởng thuật toán

Ta liên tục hoán đổi vị trí của hai phần tử trong một cặp đang xét nếu chúng đang ở sai thứ tự mà ta đang cần sắp xếp.

10.2 Các bước hoạt động

Giả sử ta có mảng một chiều sau đây với 5 phần tử: a[5] = 3, 6, 0, 1, 2.

Ở lần lặp thứ nhất, ta xét a[0] với từng phần tử từ a[1] đến a[4].

Xét 3 và 6, không sai thứ tự

Xét 3 và 0, do 3 > 0, trong khi ta đang muốn sắp xếp tăng dần, hoán đổi vị trí 3 và 0. Ta được mảng 0 6 3 1 2.

Xét 0 và 1, không sai thứ tự.

Xét 0 và 2, không sai thứ tự.

Sau lần lặp thứ nhất, ta được mảng: 0 6 3 1 2.

Ở lần lặp thứ hai, ta xét a[1] với từng phần tử từ a[2] đến a[4].

Xét 6 và 3, sai thứ tự nên ta hoán đổi vị trí hai phần tử này, ta được: 0 3 6 1 2

Xét 3 và 1, sai thứ tự nên ta hoán đổi vị trí hai phần tử này, ta được: 0 1 6 3 2

Xét 1 và 2, không sai thứ tự.

Sau lần lặp thứ hai, ta được mảng: 0 1 6 3 2.

Ở lần lặp thứ ba, ta xét a[2] với từng phần tử từ a[3] đến a[4]

Xét 6 và 3, sai thứ tự nên ta hoán đổi vị trí hai phần tử này, ta được: 0 1 3 6 2

Xét 3 và 2, sai thứ tự nên ta hoán đổi vị trí hai phần tử này, ta được: 0 1 2 6 3

Sau lần lặp thứ hai, ta được mảng: 0 1 2 6 3.

Ở lần lặp cuối cùng, ta xét a[3] với a[4]. Xét 6 và 3, sai thứ tự nên ta hoán đổi vị trí hai phần tử này, ta được: 0 1 2 3 6.

Vậy, ta được mảng đã sắp xếp theo thứ tự tăng dần.

10.3 Phân tích thuật toán

Độ phức tạp thuật toán

Trường hợp tốt nhất: O(n), khi mảng đã được sắp xếp tăng sẵn, lúc này thuật toán không cần phải thực hiện swap.

Trường hợp trung bình: $O(n^2)$

Trường hợp xấu nhất: $O(n^2)$

Độ phức tạp về không gian: do trong quá trình thực hiện, thuật toán không yêu cầu thêm bất kỳ bộ nhớ nào (trừ các biến tạm, biến chỉ số phần tử cần thiết phải có) nên độ phức tạp về không gian là O(1).

Độ ổn định của thuật toán: ổn định.

11 Radix-sort

11.1 Ý tưởng thuật toán

Khác với tất cả các thuật toán nêu trên, RadixSort không sử dụng việc so sánh 2 phần tử.

Đầu tiên, thuật toán sẽ chia các phần tử thành các nhóm, dựa trên chữ số cuối cùng (hoặc dựa theo bit cuối cùng, hoặc vài bit cuối cùng).

Sau đó ta đưa các nhóm lại với nhau, và được danh sách sắp xếp theo chữ số cuối của các phần tử. Quá trình này lặp đi lặp lại với chữ số át cuối cho tới khi tất cả vị trí chữ số đã sắp xếp.

11.2 Các bước hoạt động

Ví dụ : Cho mảng a = 55,43,1,678,5 n=5;

Ta gọi hàm radixSort(a,n)

Ta được max = 678

Vòng for sẽ chạy với place = 1,10,100

Hàm countingSort : Mảng count[10] ứng với 10 chữ số, ta sẽ dùng mảng count để tính toán và lưu vị trí trong mảng output.

Mảng output sẽ dựa trên mảng count để đưa các phần tử vào vị trí đúng của mảng . Sau mỗi lần đưa vào ta sẽ trừ đi 1 đơn vị tại vị trí tương ứng trong mảng count.

count - index	0	1	2	3	4	5	6	7	8	9
value - init (vòng for đầu tiên)	0	0	0	0	0	0	0	0	0	0
Sau vòng for thứ 2	0	1	0	1	0	2	0	0	1	0
Sau vòng for thứ 3	0	1	1	2	2	4	4	4	5	5

Ở lần chạy đầu tiên – Hàng đơn vị:

i=4 a[4]=5 - > output[count[5]-1] -> output[3]=5, count[5]=3

i=3 a[3]=678 - > output[count[8]-1] -> output[4]=678, count[8]=4

i=2 a[2]=1 - > output[count[1]-1] -> output[0]=1, count[1] = 0

i=1 a[1]=43 - > output[count[3]-1] -> output[1]=43, count[3]=1

i=0 a[0]=55 - > output[count[5]-1] -> output[2]=5,count[5] =2

Lúc này mảng output sẽ là : 1, 43, 5, 55, 678. Sau đó ta gán các giá trị phần tử vào lại mảng a.

Ở lần chạy thứ 2 – Hàng chục:

Tương tự lần chạy thứ nhất mảng count và output lần lượt là :

	1	43	5	55	678					
count - index	0	1	2	3	4	5	6	7	8	9
value - init (vòng for đầu tiên)	0	0	0	0	0	0	0	0	0	0
Sau vòng for thứ 2	2	0	0	0	1	1	0	1	0	0
Sau vòng for thứ 3	2	2	2	2	3	4	4	5	5	5
output	0	1	2	3	4					
	1	5	43	55	678					

Ở lần chạy thứ 3 – Hàng trăm:

Tương tự 2 lần chạy trên mảng count và output lần lượt là:

	1	5	43	55	678					
count - index	0	1	2	3	4	5	6	7	8	9
value - init (vòng for đầu tiên)	0	0	0	0	0	0	0	0	0	0
Sau vòng for thứ 2	4	0	0	0	0	0	1	0	0	0
Sau vòng for thứ 3	4	4	4	4	4	4	5	5	5	5
output	0	1	2	3	4					
	1	5	43	55	678					

Vậy mảng đã sắp xếp là : 1, 5, 43, 55, 678.

11.3 Phân tích thuật toán

Ưu điểm

Có thể chạy nhanh hơn các thuật toán sắp xếp sử dụng so sánh. Ví dụ nếu ta sắp xếp các số nguyên 32 bit, và chia nhóm theo 1 bit, thì độ phức tạp là O(N).

Trong trường hợp tổng quát, độ phức tạp là O(Nlog(max(a[i])))

Độ ổn định của thuật toán: ổn định.

Nhược điểm

Không thể sắp xếp số thực.

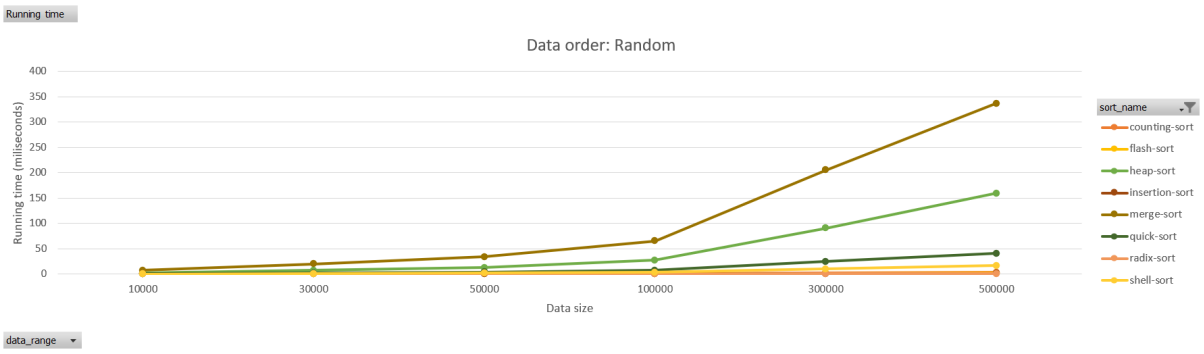
IV Kết quả thực nghiệm

1 Loại dữ liệu: Ngẫu nhiên

Bảng thời gian chạy và số phép so sánh:

Random order						
	10000		**50000**		**300000**	
	cmp	**run time**	**cmp**	**run time**	**cmp**	**run time**
bubble-sort	100009999	113.816	2500049999	2696.34	90000299999	97193.4
counting-sort	40013	0.068	200007	0.337	1200005	2.018
flash-sort	91782	0.5087	434310	0.24282	1029001	1.1775
heap-sort	668070	2.187	3928469	12.9245	27432005	91.1086
insertion-sort	29998	66.7227	149998	1600.52	899998	6640.43
merge-sort	479192	7.6846	2715006	34.2643	18108314	205.816
quick-sort	286972	0.6229	1674075	3.9003	11472573	25.0816
radix-sort	50025	0.3698	250025	0.17651	600000	0.6511
selection-sort	100010001	132.978	2500050001	2450.62	90000300001	88310.3
shaker-sort	100005001	118.02	2500025001	2791.85	90000150001	100899
shell-sort	261072	0.3163	1341072	1.8458	8091072	9.8748

1.1 Thời gian chạy



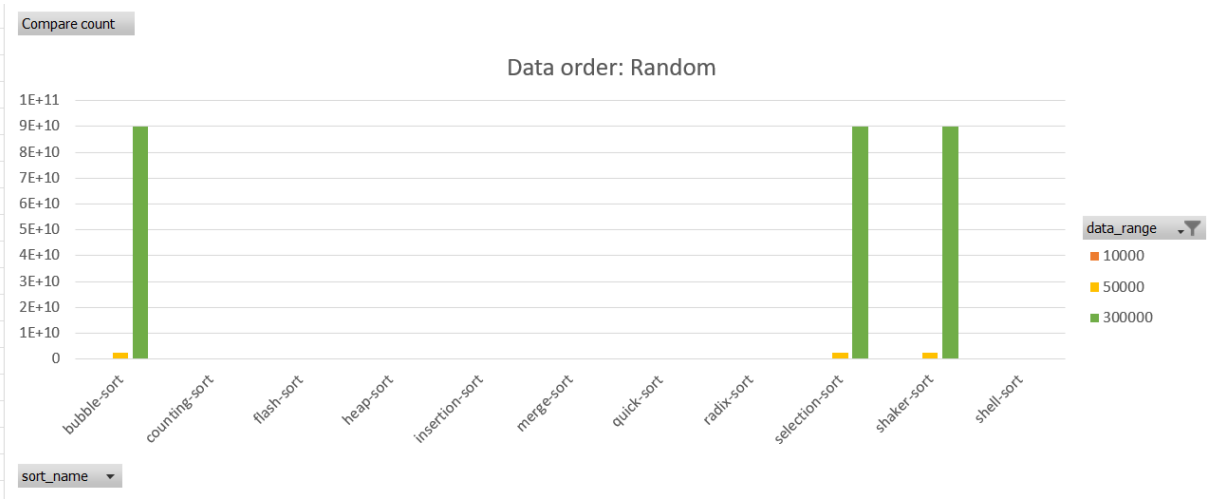
Đồ thị này đã được lược bỏ selection sort, bubble sort và shaker sort vì thời gian chạy quá lớn làm cho những thuật toán còn lại khó quan sát. Qua đây ta thấy được các thuật toán có độ phức tạp $O(n^2)$ là các thuật toán tệ nhất, có nhiều phép so sánh và hoán vị nhất dù có là cải tiến như shaker sort cũng không có nhiều hiệu quả.

Trừ 3 thuật toán trên, merge sort chạy chậm nhất do công việc phân tách và trộn mảng ngẫu nhiên, với một nửa thời gian của merge sort là heap sort.

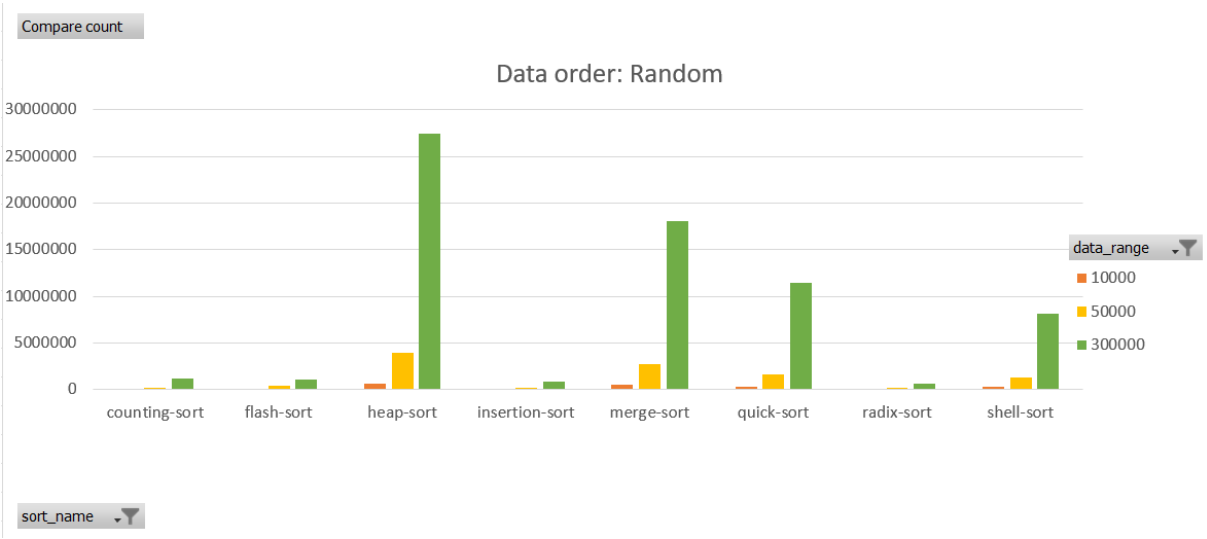
Tiếp theo ta loại bỏ 2 thuật toán trên khỏi biểu đồ. Ta thấy quick sort chạy chậm nhất do mất nhiều thời gian phân hoạch mảng ngẫu nhiên. Nhanh hơn là shell sort với một nửa thời gian của quick sort.

Đến với nhóm thuật toán còn lại. Thời gian chạy của các thuật toán còn lại khá tương đương với nhau, nhanh nhất là radix sort.

1.2 Phép so sánh



Ở mức $n >= 500000$ phần tử thì bubble sort, selection sort và shaker sort có số lần so sánh cực lớn nên tôi sẽ loại bỏ chúng ra khỏi biểu đồ.



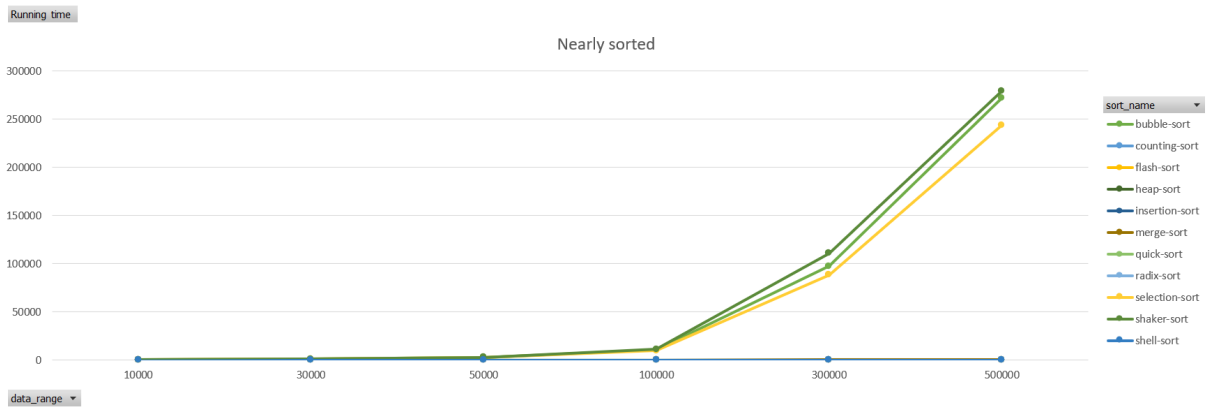
Với các thuật toán còn lại, với lượng dữ liệu $n <= 500000$ thì số lượng phép so sánh không quá nhiều. Ở mức số lượng phần tử cao hơn thì heap sort, merge sort và quick sort cho kết quả lớn do phải thực hiện tạo heap, phân tách và phân hoạch.

2 Loại dữ liệu: Gần như sắp xếp

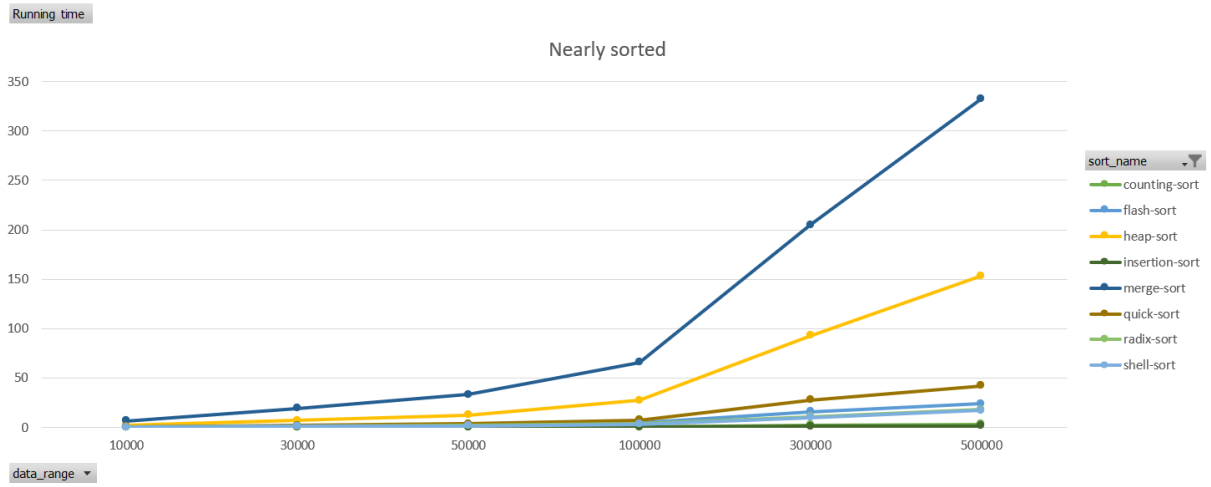
Bảng thời gian chạy và số phép so sánh:

Nearly sorted order						
	10000		**50000**		**300000**	
	cmp	**run time**	**cmp**	**run time**	**cmp**	**run time**
bubble-sort	100009999	108.357	2500049999	2688.58	90000299999	97227.6
counting-sort	40009	0.068	200007	0.3371	1200009	2.3239
flash-sort	91783	0.5316	434310	2.4525	2802439	16.1161
heap-sort	670333	2.1614	3925355	12.5142	27413234	93.0389
insertion-sort	29998	0.036	149998	0.2404	899998	1.0643
merge-sort	485559	6.6876	2715006	33.1297	19081135	204.758
quick-sort	281070	0.7205	1674075	3.5996	11487835	27.6048
radix-sort	50025	0.5114	250025	1.9495	1500025	10.8967
selection-sort	100010001	98.2894	2500050001	2455.83	90000300001	88208.8
shaker-sort	100005001	111.468	2500025001	2804.55	90000150001	110604
shell-sort	261072	0.3207	1341072	1.6357	8091072	9.847

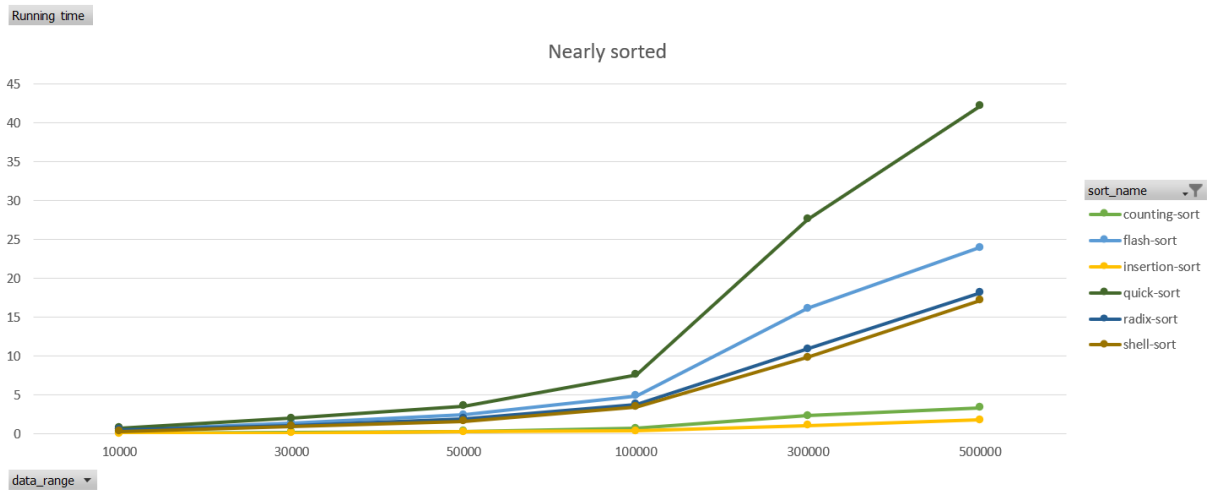
2.1 Thời gian chạy



Các thuật toán chạy chậm nhất vẫn là các thuật toán có độ phức tạp $O(n^2)$ lần lượt là shaker sort, bubble sort và selection sort. Thậm chí thuật toán cải tiến shaker sort còn chậm hơn cả bubble sort.

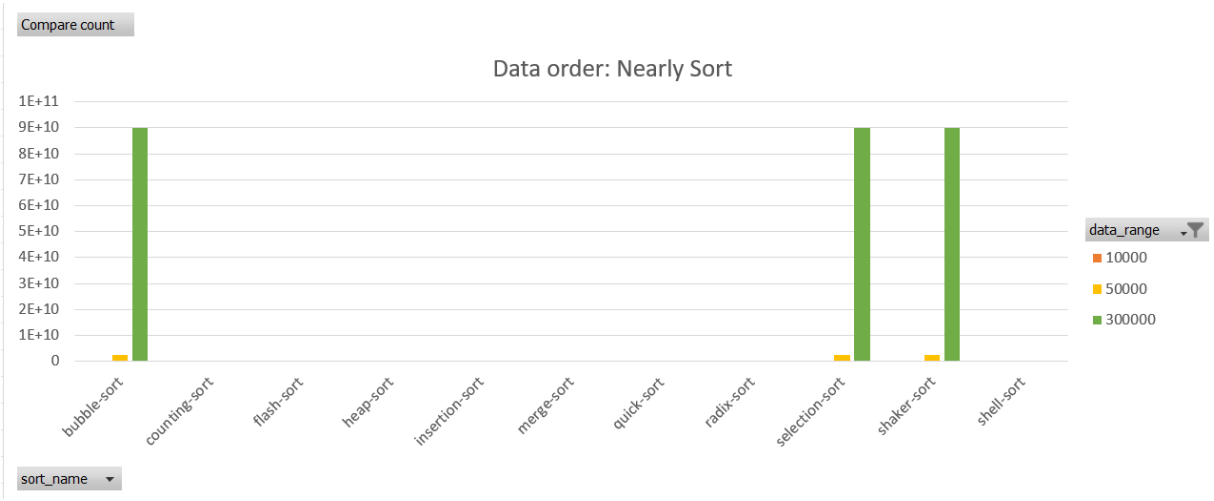


Ta loại bỏ 3 thuật toán trên ra khỏi biểu đồ. Một lần nữa, sau khi loại bỏ 3 thuật toán trên, merge sort vẫn là thuật toán chậm nhất và ít hơn một nửa thời gian vẫn là heap sort.

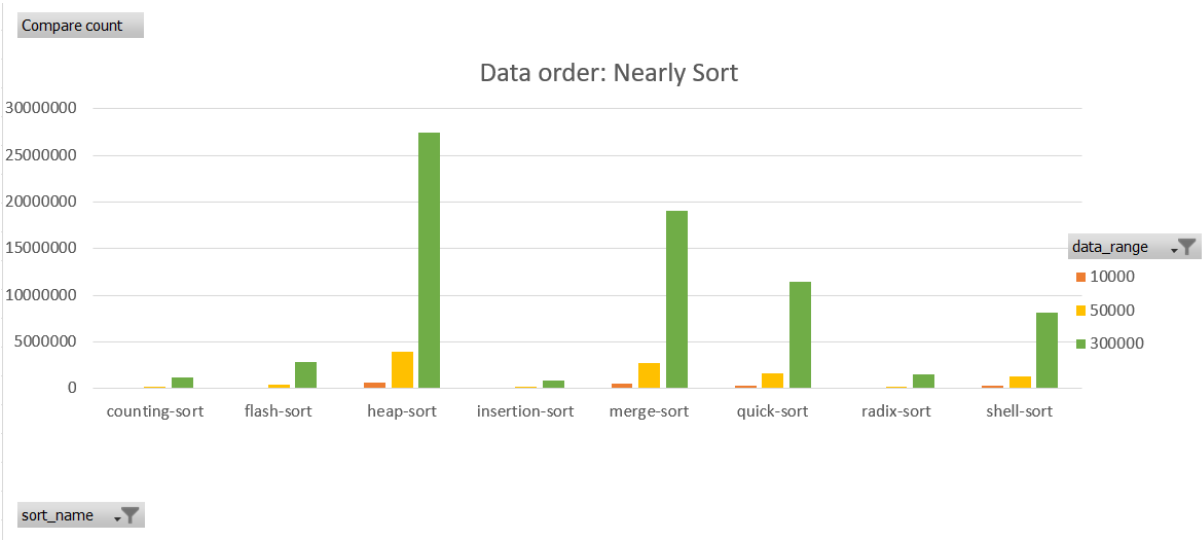


Và các thuật toán còn lại, nhanh nhất là insertion sort và counting sort với độ ổn định cao nhất. Chậm hơn là shell sort và radix sort với thời gian tương đương nhau. Cuối cùng là flash sort và chậm nhất là quick sort.

2.2 Phép so sánh



Tương tự loại dữ liệu ngẫu nhiên. Bubble sort, Selection sort và Shaker sort vẫn có số lượng phép so sánh cực cao ở mức $n >= 500000$ phần tử. Ta loại bỏ chúng ra khỏi biểu đồ để tiện quan sát các thuật toán còn lại.



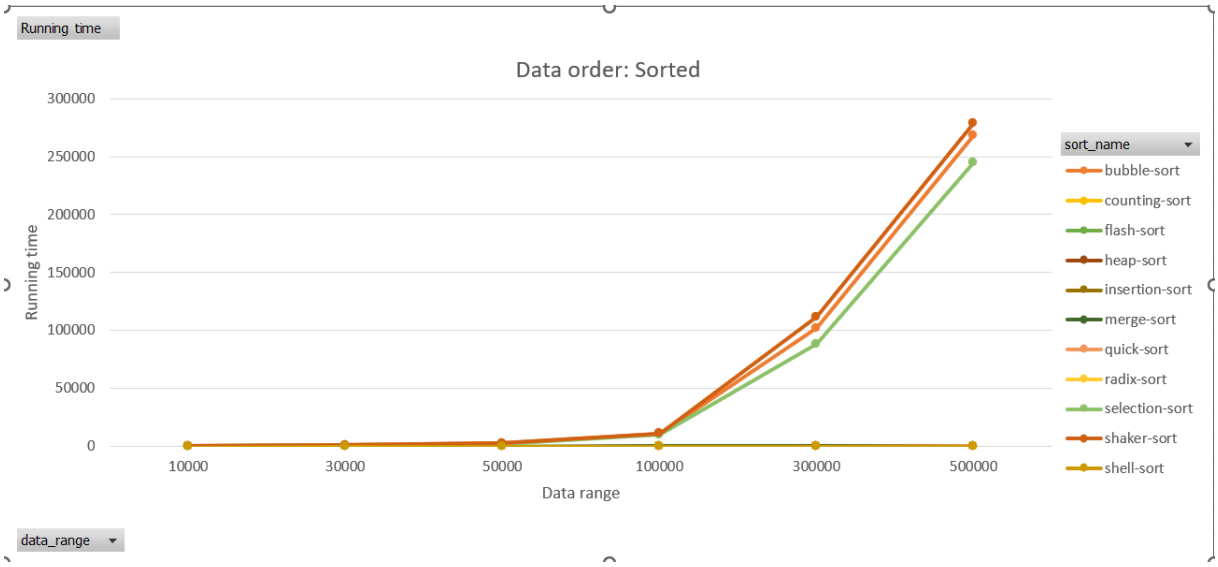
Ở mức $n >= 500000$ phần tử, một lần nữa Heap sort, Merge sort và Quick sort có số phép so sánh nhiều nhất trong các thuật toán còn lại vì tốn nhiều chi phí cho heapify, phân tách và phân hoạch. Các thuật toán còn lại tốn rất ít chi phí cho việc so sánh.

3 Loại dữ liệu: Đã sắp xếp

Bảng thời gian chạy

Sorted order						
	10000		**50000**		**300000**	
	cmp	**run time**	**cmp**	**run time**	**cmp**	**run time**
bubble-sort	100009999	107.374	2500049999	2694.04	90000299999	102088
counting-sort	40009	0.0674	200007	0.338	1200009	2.0501
flash-sort	91783	0.5163	434310	2.426	2802439	16.1009
heap-sort	670333	2.1397	3925355	12.3644	27413234	103.756
insertion-sort	29998	0.0359	149998	0.1779	899998	1.3501
merge-sort	485559	6.6434	2715006	33.6621	19081135	229.796
quick-sort	281070	0.6084	1674075	3.5877	11487835	29.2766
radix-sort	50025	0.3797	250025	2.1325	1500025	11.765
selection-sort	100010001	97.2195	2500050001	2444.8	90000300001	88263.5
shaker-sort	100005001	111.069	2500025001	2794.06	90000150001	111521
shell-sort	261072	0.3163	1341072	1.6361	8091072	9.9415

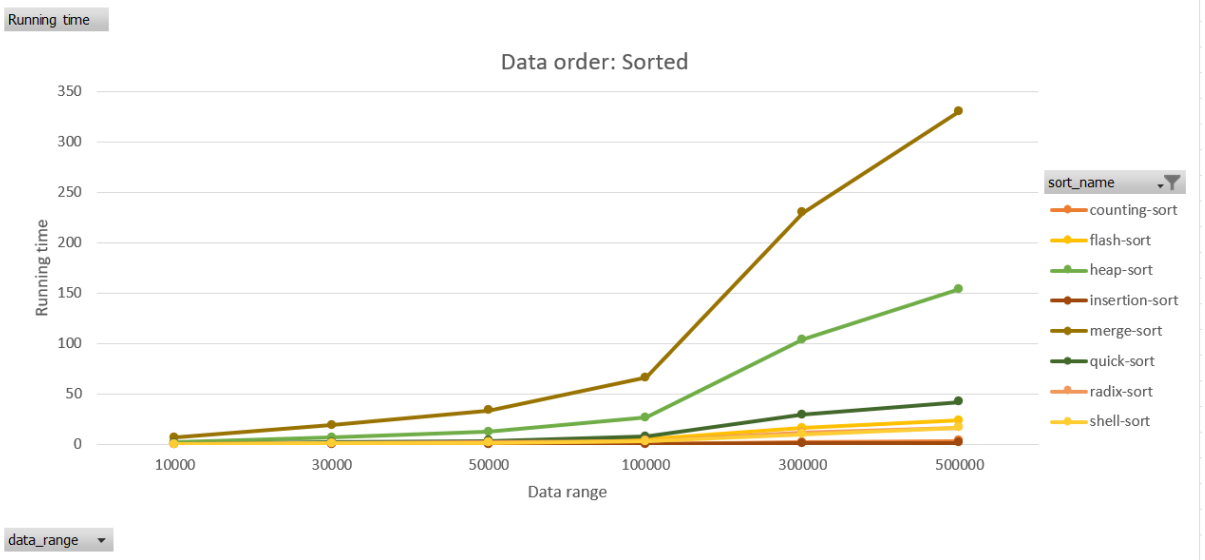
3.1 Thời gian chạy



Qua đồ thị ta thấy dù là mảng đã được sắp xếp tuy nhiên với những thuật toán có độ phức tạp $O(n^2)$ như Shaker sort, Bubble sort và Seclection sort thì thời gian chạy vẫn quá lớn. trong đó thuật toán chạy lâu nhất là bubble sort và với thuật toán cải thiện của nó là shaker sort thì thời gian thực hiện vẫn không có quá nhiều sự thay đổi.

Để thuận tiện cho việc phân tích các thuật toán còn lại, tôi sẽ loại bỏ 3 thuật toán này ra khỏi chart để dễ quan sát

3.2 Phép so sánh



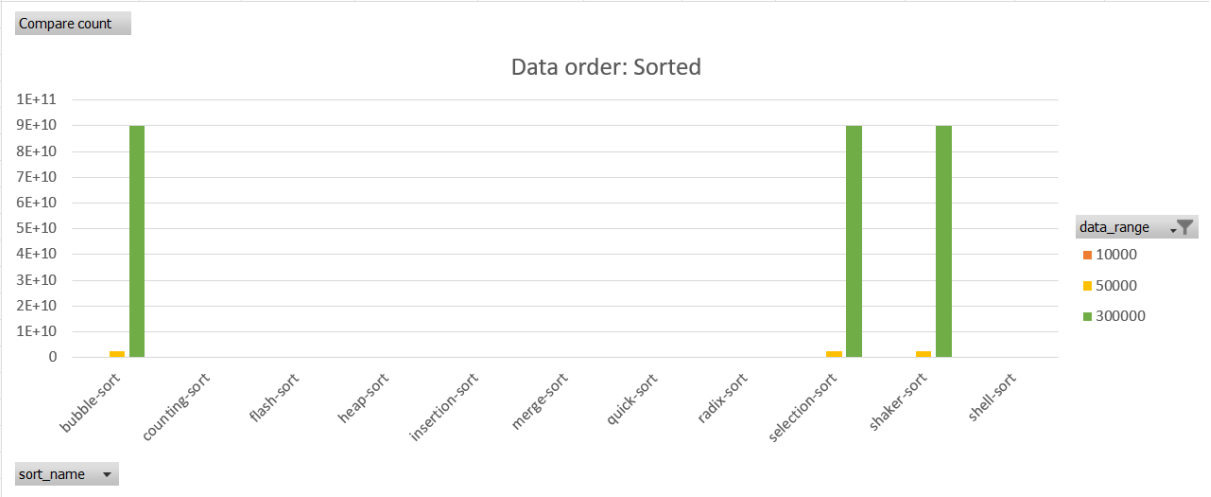
Trong số những thuật toán còn lại có độ phức tạp chủ yếu là O(nlogn) và O(n). Trong đó thuật toán chậm nhất là Merge sort với độ phức tạp O(nlogn) vì dù là với dữ liệu đã sắp xếp thì thuật toán vẫn tốn thời gian để phân tách và gộp lại. Tiếp theo đó là Heap sort, sỡ dĩ thuật toán này khá lâu vì nó tốn thời gian để xây dựng max - heap

Tiếp theo đó là Quick sort và Flash sort

Shell sort và Radix sort có thời gian chạy trong trường hợp này khá tương đương nhau.

Thuật toán Counting sort và Insertion sort có thời gian chạy nhanh nhất bởi lẽ Insertion sort có thời gian chạy nhanh nhất bởi lẽ trong trường hợp dữ liệu đã được sắp xếp thì độ phức tạp của thuật toán trong trường hợp này là O(n)

Bảng số phép so sánh



Nhìn vào biểu đồ ta có thể thấy Bubble sort, Selection sort và Shaker sort có số lượng phép so sánh lớn nhất bởi lẽ ta phải liên tục so sánh 2 phần tử để tim phần tử nhỏ hơn. Và dù với bộ dữ liệu dã sắp xếp thì 2 thuật toán này vẫn không có sự giảm thiệu việc so sánh bất kì 2 phần tử nào.

Để thuận tiện cho việc phân tích các thuật toán còn lại, tôi sẽ loại bỏ 3 thuật toán này ra khỏi chart để dễ quan sát

Có thể thấy trong số những thuật toán dựa trên việc so sánh 2 phần tử thì Shell sort và Insertion sort lần lượt là hai thuật toán có số lượng phép so sánh ít nhất bởi lẽ mỗi phần tử ta chỉ cần so sánh đúng 1 lần

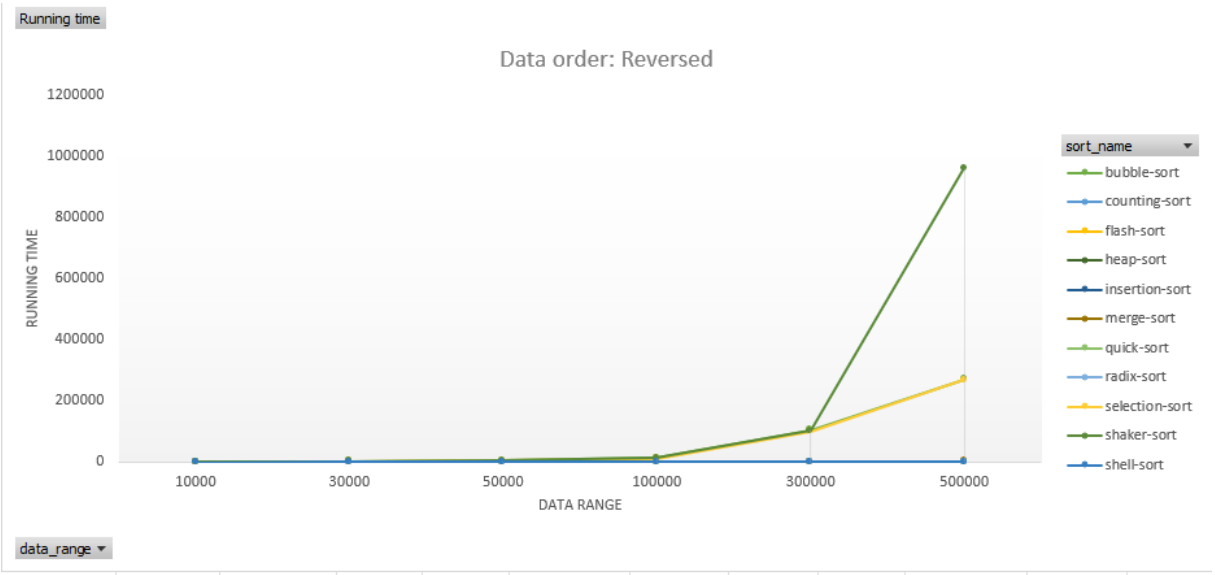
Tiếp theo là tới nhóm thuật toán được xây dựng không dựa trên việc so sánh 2 phần tử trong mảng là Radix sort, Counting sort, Flash sort và Quick sort thì Couting sort là thuật toán có số lượng phép so sánh ít nhất.

4 Loại dữ liệu: Có thứ tự đảo ngược

4.1 Thời gian chạy

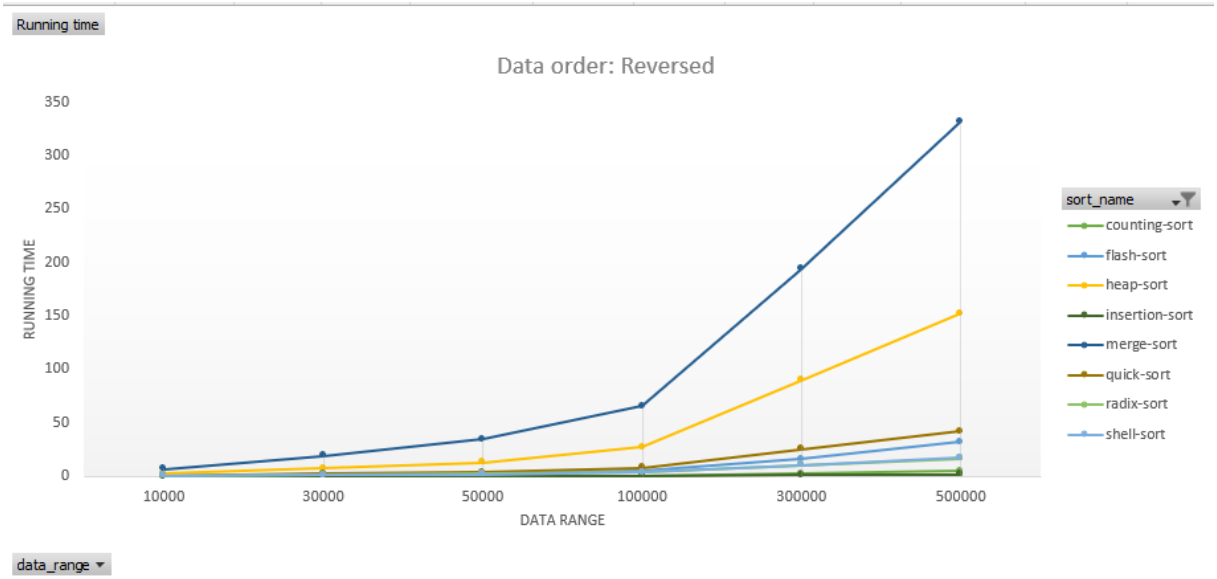
Bảng thời gian chạy:

Reversed order						
	10000		**50000**		**300000**	
	cmp	**run time**	**cmp**	**run time**	**cmp**	**run time**
bubble-sort	100009999	112.736	2500049999	2694.22	90000299999	102432
counting-sort	40009	0.0682	200007	0.3369	1200009	2.0184
flash-sort	91783	0.5141	434310	2.4286	2802439	15.9152
heap-sort	670333	2.14	3925355	12.5706	27413234	89.913
insertion-sort	29998	114.2	149998	3178.17	899998	98901.3
merge-sort	485559	6.6036	2715006	34.3653	19081135	194.12
quick-sort	281070	0.6087	1674075	3.5935	11487835	24.902
radix-sort	50025	0.3712	250025	1.9318	1500025	9.9958
selection-sort	100010001	106.757	2500050001	2693.52	90000300001	98866.4
shaker-sort	100005001	110.799	2500025001	2806.34	90000150001	100723
shell-sort	261072	0.3186	1341072	1.6336	8091072	10.0052



Tốn nhiều thời gian nhất là Bubble sort (có bước nhảy vọt từ 30000 lên 50000 phần tử, tiếp theo là Shaker sort, Selection sort. Điều này do 3 thuật toán trên sử dụng rất nhiều phép so sánh và phép hoán vị. Tôi sẽ xếp 3 thuật toán trên vào nhóm Lâu nhất.

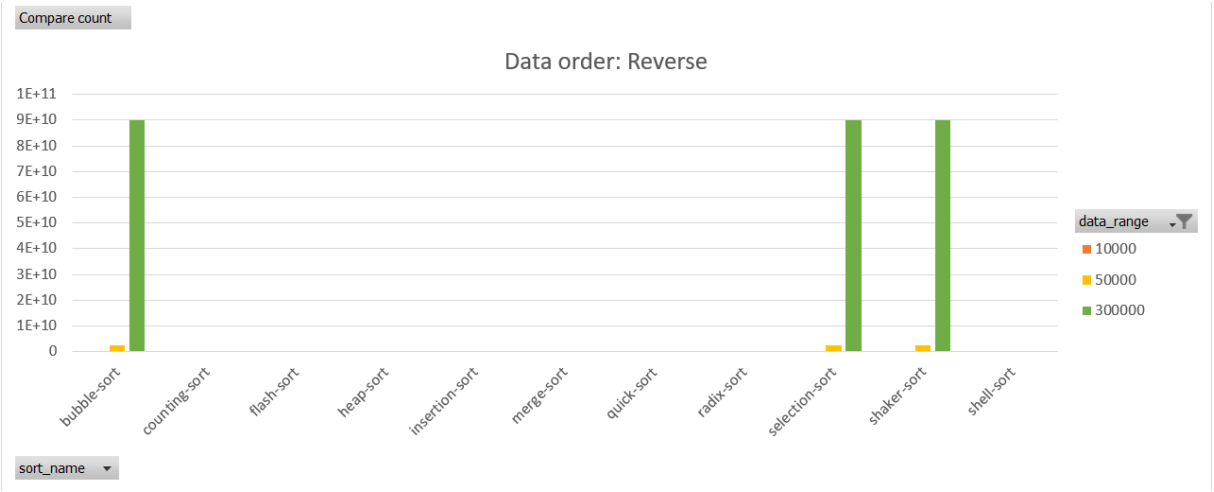
Để thuận tiện cho việc phân tích các thuật toán còn lại, tôi sẽ loại bỏ 3 thuật toán này ra khỏi chart để dễ quan sát.



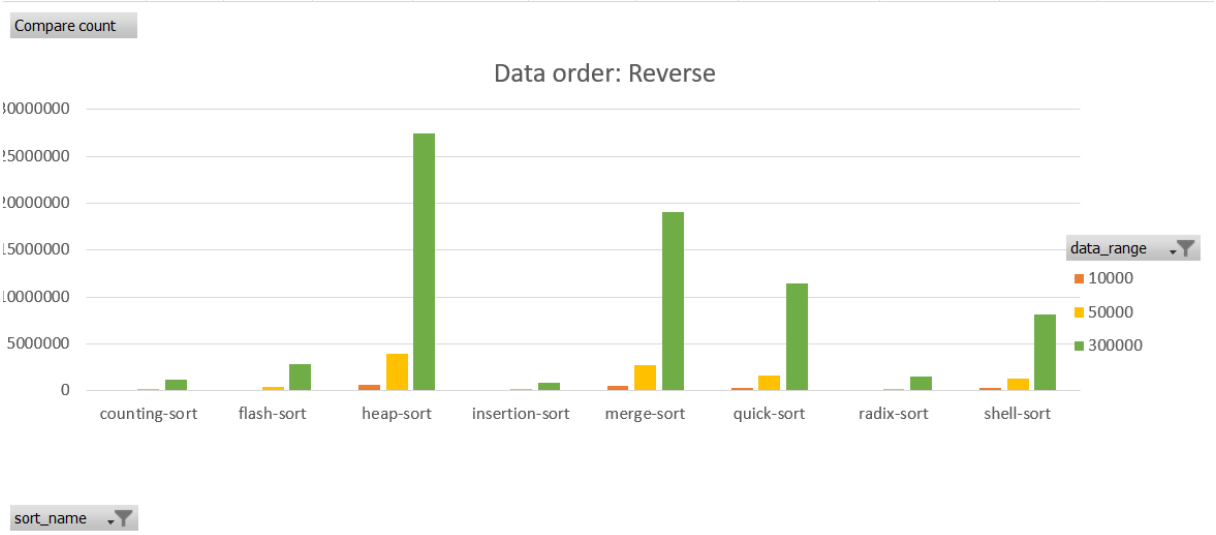
Qua biểu đồ ta thấy Merge sort sẽ tốn nhiều thời gian nhất bởi nó sẽ tốn thời gian phân tách, so sánh từng dãy con trong trường hợp dữ liệu bị đảo ngược. Một phần do cách tôi xây dựng nó. Tiếp sau đó là Heap sort, Quick sort. Sở dĩ Heap sort lâu trong kiểu dữ liệu đảo ngược là nó sẽ tốn rất nhiều thời gian trong việc xây dựng max-heap (đạt trường hợp xấu nhất). Và nhóm còn lại là Radix sort, Counting sort, Flash sort hoạt động rất ổn định.

4.2 Phép so sánh

Biểu đồ dạng cột.



Dễ dàng thấy được 3 thuật toán tốn nhiều phép so sánh nhất là Bubble sort, shell sort, selection sort. 3 thuật toán này làm đồ thị khó quan sát các thuật toán còn lại nên tôi lược bỏ 3 thuật toán này.



Nhìn chung, với dữ liệu $n <= 50000$ thì hầu hết các thuật toán đều có rất ít phép so sánh. Có thể thấy trong nhóm các thuật toán này, Heap sort và Merge sort tốn nhiều phép so sánh nhất vì tốn chi phí tạo heap và phân tách.

V Tổng kết

Thuật toán	Thời gian	Không gian	Độ ổn định
Selection Sort	$O(n^2)$	$O(1)$	Không
Insertion Sort	$O(n^2)$	$O(1)$	Có
Bubble Sort	$O(n^2)$	$O(1)$	Có
Shaker Sort	$O(n^2)$	$O(1)$	Có
Shell Sort	$O(nlog(n)^2$	$O(1+m)$	Không
Heap Sort	$O(nlogn)$	$O(1)$	Không
Merge Sort	$O(nlogn)$	$O(n)$	Có
Quick Sort	$O(n)$	$O(log(n))$	Không
Counting Sort	$O(n+k)$	$O(max)$	Có
Radix Sort	$O(nlog(max(a[i])))$	$O(1)$	Có
Flash Sort	$O(n)$	$O(m)$	Không

1 Nhóm các thuật toán có tốc độ chậm

- Bao gồm các thuật toán:Bubble Sort, Shaker Sort, Selection Sort, Insertion Sort. Trong đó Bubble sort có tóc độ chậm nhất, chậm hơn hẳn các thuật toán khác. Insertion phù hợp nhất với các loại dữ liệu có số lượng ít và hoạt động rất tốt trên loại dữ liệu đã sắp xếp và gần như được sắp xếp.

2 Nhóm các thuật toán có tốc độ trung bình

- Bao gồm các thuật toán: Merge Sort, Heap Sort, Quick Sort. Trong đó Merge sort có tốc độ chạy chậm nhất, và tốn thêm bộ nhớ ngoài. Heap sort và Quick sort có tốc độ nhanh hơn trong đa số trường hợp Quick sort có nguy cơ lên tới $O(n^2)$ (do cách chọn phần tử pivot).

- Một thuật toán khó đánh giá độ phức tạp nhưng dựa trên thực nghiệm có thể thấy Shell sort có tốc độ chạy cũng rất tốt không kém gì 3 thuật toán trên.

- Radix-sort cũng sở hữu độ phức tạp $O(nlog(max(a[i])))$, chạy rất nhanh ổn định so với 3 thuật toán trên vì nó sắp xếp dựa vào cơ số chứ không dựa vào phép so sánh. Nhưng nhược điểm là không sắp xếp được số thực.

3 Nhóm các thuật toán có tốc độ nhanh

- Bao gồm Flash sort và Counting sort. Hai thuật toán chay rất nhanh nhưng cách cài đặt rất phức tạp.

- Với lượng dữ liệu từ nhỏ đến trung bình thì Counting sort phù hợp hơn cả, với tốc độ cao và dễ cài đặt. Nhưng với lượng dữ liệu lớn thì Flash sort là phù hợp nhất. Trong một vài trường hợp thì Counting sort cho tốc độ vượt trội hơn Flash sort.

- Các thuật toán này đều sử dụng thêm bộ nhớ ngoài.

4 Nhóm các thuật toán dựa vào so sánh

- Nhóm này bao gồm các thuật toán như: Bubble Sort, Shaker Sort, Insertion Sort, Selection Sort, Shell Sort, Heap Sort, Merge Sort, Quick Sort.

- Nhìn chung ta thấy các thuật toán này có độ phức tạp nhỏ hơn $O(n^2)$. Trong hầu hết trường hợp thì Quick sort là chạy nhanh nhất trong nhóm này nhưng Merge-sort lại ổn định hơn và đảm bảo trường hợp tệ nhất sẽ không lớn hơn $O(nlogn)$.

5 Nhóm các thuật toán không dựa vào so sánh

- Bao gồm các thuật toán như: Counting sort, Radix sort và Flash sort.

- Về thuật toán Counting sort: Sử dụng phân phối dữ liệu để sắp xếp.
- Về thuật toán Radix sort: Sắp xếp dựa trên cơ số.
- Về thuật toán Flash sort: Sắp xếp dựa trên phân phối dữ liệu (chia nhóm các phần tử) và một phần phép so sánh (có dùng thuật toán insertion).

- Các thuật toán này có độ ổn định cao (hoạt động tốt trên cả 4 loại dữ liệu). Tuy nhiên chỉ thích hợp cho việc sắp xếp các loại dữ liệu là số (số nguyên, số thực(trừ Radix sort không sắp xếp được số thực)) còn các kiểu dữ liệu là Xâu, kiểu dữ liệu trừu tượng thì rất khó hoặc không thể áp dụng và cài đặt.

VI Tài liệu tham khảo

1. https://vnoi.info/wiki/algo/basic/sorting.md
2. https://www.geeksforgeeks.org/radix-sort/
3. https://en.wikipedia.org/wiki/Sorting_algorithm
4. https://github.com/leduythuccs/Sorting-Algorithms
5. https://www.programiz.com/dsa
6. https://stackoverflow.com/questions/2539545/fastest-gap-sequence-for-shell-sort
7. Slide - Cấu trúc dữ liệu và giải thuật - GV: Nguyễn Ngọc Thảo
8. Sách nhập môn cấu trúc dữ liệu và thuật toán - Đại học Khoa học Tự Nhiên, Đại học Quốc Gia TP HCM
9. Template Latex: 21CLC06