BỘ GIÁO DỤC VÀ ĐÀO TẠO
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN THÀNH PHỐ HỒ CHÍ MINH
KHOA CÔNG NGHỆ THÔNG TIN

BÀI BÁO CÁO

Lab 3: Sorting

Môn: Cấu trúc Dữ liệu và Giải thuật

Giảng viên : Nguyễn Thanh Phương

Sinh viên thực hiện : Trần Kiều Minh Lâm
Lớp : 20CTT1TN

MSSV : 20120018

THÀNH PHỐ HỒ CHÍ MINH, 11/2021

MỤC LỤC

1. Giới thiệu

- Mức độ hoàn thành: 100%
- Cấu hình máy: Intel(R) Core(TM) i5-10310U CPU @ 1.70GHz 2.21 GHz - 8GB RAM.
- Hệ điều hành Window 10 Pro
- Compiler: GNU trên Window Subsystem for Linux – Ubuntu.

Mục lục…

2. Trình bày thuật toán

Nội dung tiếp theo gồm trình bày ý tưởng, mã giả và độ phức tạp của từng thuật toán sắp xếp. Các thuật toán dưới đây chỉ sắp xếp thành mảng tăng dần. Cách sắp xếp giảm dần sẽ tương tự như vậy.

2.1. Selection sort

a) Ý tưởng:

Tổng quát, lần lượt chọn (select) phần tử nhỏ nhất, nhỏ nhì,…. và rồi lần lượt đưa chúng về đầu mảng.

Cụ thể, chia mảng ra hai phần: phần đầu đã được sắp xếp, phần còn lại là chưa được sắp xếp. Ban đầu, chưa có phần tử nào trong phần được sắp xếp. Lần lượt tìm phần tử nhỏ nhất trong phần chưa sắp xếp và thêm nó vào cuối của phần được sắp xếp.

b) Mã giải:

Bước 1: Duyệt qua từng phần tử của mảng theo thứ tự.

Bước 2: Tìm vị trí phần tử nhỏ nhất của phần chưa được sắp xếp bằng cách, duyệt qua phần còn lại của mảng (tính từ phần tử đang xét ở bước 1 về cuối mảng).

Bước 3: Hoán đổi giá trị tại phần tử đang xét ở bước 1 và phần tử nhỏ nhất tìm được ở bước 2.

c) *Độ phức tạp thời gian:* do có 2 vòng lặp lồng nhau nên độ phức tạp:

i. Trung bình: $O(n^2)$
ii. Tốt nhất: $O(n^2)$
iii. Xấu nhất $O(n^2)$

d) *Độ phức tạp không gian:* $O(1)$ do sắp xếp trực tiếp trên mảng

2.2. Insertion sort

a) Ý tưởng:

Nhìn chung, thuật toán khá giống với việc xếp bài. Mỗi lần bốc được một lá bài mới, chèn (insert) nó vào trong đống bài đã được sắp xếp.

Cụ thể, chia mảng ra hai phần: phần đầu đã được sắp xếp, phần còn lại là chưa được sắp xếp. Ban đầu, chưa có phần tử nào trong phần được sắp xếp. Lần lượt lấy từng phần tử của phần chưa được sắp xếp, thêm nó vào phần đã được sắp xếp. Do phần đã được sắp xếp, nên thêm một phần tử mới vào, bằng cách, duyệt qua từng phần tử và tìm vị trí mà phần tử mới có thể thêm vào.

b) Mã giả:

Bước 1: Duyệt qua từng phần tử của mảng.

Bước 2: Nếu phần tử tại bước 1 nhỏ hơn phần tử liền trước thì đưa phần tử lớn hơn về sau. Lặp lại bước 2 nếu vị trí mới thỏa điều kiện. Nếu không sang bước 3.

Bước 3: Gán vị trí trống bằng với giá trị phần tử tại bước 1.

c) *Độ phức tạp:* do có 2 vòng lặp lồng nhau nên độ phức tạp:

ii. Trung bình: $O(n^2)$

iii. Tốt nhất: $O(n)$ trong trường hợp mảng đầu vào đã được sắp xếp sẵn. Ta không cần tìm vị trí phù hợp để chèn.

iv. Xấu nhất $O(n^2)$ trong trường hợp mảng đầu vào đã được sắp xếp theo thứ tự ngược lại.

d) *Độ phức tạp không gian:* $O(1)$ do sắp xếp trực tiếp trên mảng.

2.3. Bubble sort

a) Ý tưởng:

Nhìn chung, thuật toán khá giống với việc những quả bong bóng (bubble) ở trong nước nổi dần dần lên. Quả bóng nào nhẹ hơn sẽ nổi lên trước hay phần tử nào nhỏ hơn sẽ được đưa dần về đầu mảng.

Cụ thể, lần lượt duyệt các cặp phần tử kề nhau, thay đổi vị trí của chúng nếu sai thứ tự.

b) Mã giả:

Bước 1: Duyệt các cặp phần tử liền kề từ cuối mảng về đầu. Nếu 2 phần tử kề nhau mà sai thứ tự, hay phần tử ở sau nhỏ hơn thì thay đổi giá trị của 2 phần tử đó (phần tử giá trị nhỏ nổi lên đầu).

Bước 2: Tiếp tục lặp lại bước 1, đưa dần phần tử nhỏ về đầu cho đến khi mảng được sắp xếp.

c) *Độ phức tạp:* do có 2 vòng lặp lồng nhau nên độ phức tạp:

i. Trung bình: $O(n^2)$

ii. Tốt nhất: $O(n^2)$: trong trường hợp mảng đầu vào đã được sắp xếp sẵn. Có thể cái tiến lên $O(n)$ bằng cách, sau khi duyệt vòng lặp lần 2, nếu không có 2 phần tử nào sai vị trí thì không cần duyệt các lần sau nữa.

iii. Xấu nhất $O(n^2)$: trong trường hợp mảng đầu vào đã được sắp xếp theo thứ tự ngược lại.

d) *Độ phức tạp không gian:* $O(1)$ do sắp xếp trực tiếp trên mảng.

2.4. Shaker sort

a) Ý tưởng:

Shaker sort hay có tên gọi khác là cocktail sort. Thuật toán giống như một phiên bản biến thể của bubble sort.

Thuật toán sẽ lần lượt đưa các phần tử nhỏ nhất về đầu và các phần tử lớn nhất về cuối cùng một lúc thay vì chỉ đưa phần tử nhỏ nhất về đầu giống bubble sort. Thao tác đó giống như đang lắc (shake) ly *cocktail* để đưa phần tử bé nhất với lớn nhất ra xa nhau hay về 2 đầu của mảng.

b) Mã giả:

Tương tự bubble sort.

Bước 1: Duyệt từ đầu về cuối, nếu 2 phần tử liền kề nhau sai thứ tự thì đổi chỗ.

Bước 2: Duyệt từ cuối về đầu, nếu 2 phần tử liền kề nhau sai thứ tự thì đổi chỗ.

Bước 3: Thực hiện lại bước 1, 2 nếu mảng vẫn chưa sắp xếp xong.

c) *Độ phức tạp:* tương tự với bubble sort trong trường hợp mảng đầu vào ngẫu nhiên. Trong một số trường hợp nhanh hơn bubble sort 2 lần.

i. Trung bình: $O(n^2)$

ii. Tốt nhất: $O(n^2)$

iii. Xấu nhất: $O(n^2)$

d) *Độ phức tạp không gian:* $O(1)$ do sắp xếp trực tiếp trên mảng.

2.5. Shell sort

a) Ý tưởng:

Thuật toán giống như là một biến thể khác của insertion sort. Trong khi insertion sort sẽ đưa phần tử từng bước về phía trước, Shell sort có thể di chuyển các phần tử về đúng vị trí trong ít bước hơn. Với mỗi giá trị gap giảm dần, Shell sort sẽ dần dần biến tất cả các mảng con độ dài gap được sắp xếp.

Thông tin thêm: Shell sort được đặt tên theo nhà toán học Donald Shell.

b) Mã giả:

Bước 1 : Khởi tạo gap là một giá trị lớn và lần lượt giảm dần về 1. Có nhiều cách giảm khác nhau, kham khảo tại link: https://en.wikipedia.org/wiki/Shellsort#Gap_sequences

Bước 2: Làm tương tự như insertion sort. Mỗi lần sẽ thêm phần tử mới vào mảng đã sắp xếp sẵn cách phần tử hiện tại là gap.

c) *Độ phức tạp:* do có 2 vòng lặp lồng nhau nên

i. Trung bình: $O(n^2)$

ii. Tốt nhất: $O(n^2)$

iii. Xấu nhất: $O(n^2)$

d) *Độ phức tạp không gian:* $O(1)$ do sắp xếp trực tiếp trên mảng.

2.6. Heap sort

a) Ý tưởng:

Heap sort dựa trên cấu trúc dữ liệu Binary Heap (hàng đợi có ưu tiên). Heap sort khá tương tự với selection sort bởi vì heap sort cũng sẽ tìm các giá trị nhỏ nhất của mảng và đưa về đầu. Nhưng thay vì duyệt tất cả mảng để tìm phần tử nhỏ nhất, heap sort sẽ dùng Binary Heap để cải tiến việc tìm đó lên rất nhanh

Tại đây, bài viết sẽ không đi sâu vào cấu trúc dữ liệu Binary Heap. Đọc giả có thể tìm hiểu thêm tại đây: https://www.geeksforgeeks.org/binary-heap/

Về cơ bản Binary Heap sẽ tái cấu trúc mảng thành dạng heap (heapify) và tìm phần tử lớn nhất trong O(logN)

b) Mã giả:

Bước 1: tái cấu trúc mảng (build max-heap)

Bước 2: lần lượt hoán đổi giá phần tử lớn nhất (nằm ở đầu mảng) và phần tử cuối cùng của mảng. Sau đó heapify lại mảng (trừ phần tử cuối).

Bước 3: lặp lại bước 2 cho tới khi tất cả được sắp xếp.

c) *Độ phức tạp:* Thao tác heapify sẽ tốn O(logn) nên độ phức tạp:

i. Trung bình: O(nlogn)

ii. Tốt nhất: O(nlogn)

iii. Xấu nhất: O(nlogn)

d) *Độ phức tạp không gian:* $O(1)$ do sắp xếp trực tiếp trên mảng.

2.7. Merge sort

a) Ý tưởng:

Thuật toán ap dụng tư tưởng chia để trị, nghĩa là, để sắp xếp được toàn bộ mảng, ta cần chia đôi mảng cần sắp xếp. Gọi đệ quy việc sắp xếp cho 2 mảng con. Lại tiếp tục chia đôi mảng con đó và gọi đệ quy. Ta chia cho đến khi nào mảng còn 1 phần tử (đã được sắp xếp). Khi có được 2 mảng còn đã được sắp xếp tăng dần, tìm cách kết hợp (merge) 2 mảng con đó lại thành một mảng được sắp xếp.

Còn về chia để trị là gì có thể tìm hiểu thêm tại link:
https://www.geeksforgeeks.org/divide-and-conquer-algorithm-introduction/

b) Mã giả:

Bước 1: Tạo hàm mergeSort(arr, l, r). Bên trong hàm ta chia mảng arr làm 2 nửa ngăn cách bởi vị trí m = (l + r)/2. Gọi đệ quy mergeSort(arr, l, mid) và mergeSort(arr, mid + 1, r) và gọi hàm merge(arr, l, r) để kết hợp 2 nửa đã sắp xếp lại.

Bước 2: Tạo hàm merge(arr, l, r).

- Tạo 2 mảng L[], R[] trích từ nửa đã sắp xếp của arr. Tạo 2 con trỏ, trỏ đến 2 vị trí đầu tiên của mảng. Khi đó 2 con trỏ này nhỏ nhất.
- Nếu giá trị tại con trỏ nào nhỏ hơn thì thêm phần tử đó vào mảng arr. Lặp lại thao tác này cho đến khi có một con trỏ đi hết mảng của nó.
- Thêm tất cả những phần tử trong mảng nào chưa được thêm vào arr.

c) *Độ phức tạp:* mỗi thao tác merge sẽ tốn độ phức tạp O(n). Nhưng có thực hiện tối đa O(logn) thao tác merge (do mỗi lần chia đôi mảng). Nên độ phức tạp:

i. Trung bình: O(nlogn)

ii. Tốt nhất: O(nlogn)

iii. Xấu nhất: O(nlogn)

d) *Độ phức tạp không gian:* $O(nlogn)$ do mỗi lần gọi merge() đều cần tạo 2 mảng L[], R[].

2.8. Quick sort

a) Ý tưởng:

Tương tự như merge sort, ta cũng sẽ áp dụng tư tưởng chia để trị. Đầu tiên, ta chọn một vị trí chốt (pivot) là giá trị chia cắt mảng. Đưa những phần tử nào nhỏ hơn giá trị tại pivot về nửa trước, những giá trị lớn hơn giá trị tại pivot về nửa sau. Gọi đệ quy, tìm cách sắp xếp cho 2 nửa của mảng. Khi đó ta sẽ được mảng tổng đã được sắp xếp.

Cái tên "Quick sort" đến từ việc thuật toán này có thể sắp xếp một cách hiệu quả và nhanh hơn các thuật toán thông thường khác.

b) Mã giả:

Tạo hàm quickSort(arr, l, r). Trong hàm gồm:

Bước 1: khởi tạo pivot bằng giá trị tại vị trí ở giữa l, r. Tạo 2 con trỏ tại vị trí l, r.

Bước 2: lần lượt cho các con trỏ chạy vào giữa.

Bước 3: nếu tồn tại cặp giá trị tại con trỏ nằm sai vị trí so với pivot. Thực hiện hoán đổi giá trị tại 2 con trỏ đó.

Bước 4: thực hiện lại bước 2 cho đến khi 2 con trỏ vượt qua nhau.

Bước 5: Gọi đệ quy hàm quickSort() cho 2 nửa của mảng.

c) *Độ phức tạp:* trong trường hợp tổng quát, thuật khá giống với merge sort, cũng chia đôi mảng và cũng chạy con trỏ để duyệt qua mảng nên độ phức tạp

i. Trung bình: O(nlogn)
ii. Tốt nhất: O(nlogn)
iii. Xấu nhất: O(nlogn)

d) *Độ phức tạp không gian*: $O(1)$ do sắp xếp trực tiếp trên mảng.

2.9. Counting sort

a) Ý tưởng:

Counting sort sẽ đếm (counting) và lưu lại số lần xuất hiện của tất cả các phần tử trong mảng. Sau đó, sẽ duyệt qua mảng đánh dấu đó từ nhỏ tới lớn để sắp xếp lại mảng.

b) Mã giả:

Bước 1: Tìm số lớn nhất max trong mảng.

Bước 2: Khởi tạo mảng counter[] có max phần tử và các giá trị bằng 0

Bước 3: Duyệt qua từng phần tử a_i trong mảng đã cho. Tăng counter[a_i] lên 1 đơn vị.

Bước 4: Sau khi duyệt xong bước 3. Xóa tất cả phần tử a[] .Duyệt qua từng phần tử trong mảng counter[i]. Nếu counter[i] > 0 thì thêm counter[i] lần các giá trị i vào mảng a[].

c) *Độ phức tạp:* bằng tổng số lần duyệt của 2 vòng duyệt

i. Trung bình: O(n + k) với k là giá trị lớn nhất của mảng

ii. Tốt nhất: O(n) khi k nhỏ hơn n

iii. Xấu nhất: O(k) với k lớn có thể lên đến $n^2, n^3, ...$

d) *Độ phức tạp không gian:* $O(k)$ do cần tạo thêm mảng counter[] để đếm.

2.10. Radix sort

a) Ý tưởng:

Radix sort là thuật toán nhanh hơn tất cả những thuật toán ở trên. Độ phức tạp của nó chỉ là tuyến tính. Còn các thuật toán nhanh ở trên (merge sort, quick sort,..) chỉ là $O(nlogn)$, ngoài ra, counting sort có độ phức tạp là $O(n + k)$ nhưng trong trường hợp các giá trị là từ 1 tới n^2 thì counting sort sẽ tốn $O(n^2)$.

Quay trở lại với ý tưởng radix sort. Bản chất radix sort vẫn sẽ dựa trên counting sort hoặc các thuật toán sort "ổn định" khác để sắp xếp mảng theo từng chữ số (radix: cơ số), bắt đầu chữ số ít ảnh hưởng nhất.

Thuật toán sort "ổn định" nghĩa là khi tồn tại 2 phần tử bằng nhau thì thuật toán sort vẫn giữ tính thứ tự xuất hiện của nó. Ví dụ như bubble sort, insertion sort, merge sort, couting sort,…

b) Mã giả:

Bước 1: Duyệt qua từng cơ số của mảng, từ hàng đơn vị, tới các hàng lớn hơn.

Bước 2: Gọi thuật toán counting sort cho mảng, nhưng chỉ xét cho các chữ số cần xét.

c) *Độ phức tạp:* do dựa trên thuật counting sort và đã giới hạn lại giới hạn cho couting sort. Nên độ phức tạp:

i. Trung bình: O(n)

ii. Tốt nhất: O(n)

iii. Xấu nhất: O(n)

d) *Độ phức tạp không gian:* $O(n)$ do cần tạo mảng tạm cho counting sort.

2.11. Flash sort:

a) Ý tưởng:

Flash sort sẽ gồm 3 phần: phân lớp dữ liệu, hoán vị toàn cục, sắp xếp cục bộ. Tóm gọm lại, flash sort sẽ chia mảng ban đầu ra thành nhiều lớp, hoán vị các vị trí trong mảng để các phần tử về đúng lớp, sau đó, sắp xếp trên từng lớp.

b) Mã giả:

Bước 1: Phân lớp dữ liệu:

- Tìm giá trị nhỏ nhất minVal và giá trị lớn nhất maxVal trong mảng.
- Tạo mảng L[] có m phần tử tương ứng với m lớp. (m = 0.43n). Ban đầu các phần tử mảng L[] bằng 0. L[i] sẽ lưu số phần tử có trong lớp i.
- Duyệt qua các phần tử, nếu a[k] phần tử thuộc lớp k thì tăng L[k] lên 1 đơn vị. Với $k = int((m - 1) * (a[i] - min) / (max - min))$
- Tính vị trí kết thúc của mỗi lớp là $L[i] += L[i - 1]$.

Bước 2: Hoán vị toàn cục: Mỗi khi thấy một phần tử nằm sai lớp, ta lấy phần tử đó ra di chuyển về đúng lớp của nó và đồng thời lấy ra phần tử đang chiếm chỗ trong lớp của nó, tiếp tục đưa nó về vị trí đúng. Lặp lại liên tục cho đến khi nó không cần chuyển đi đâu nữa.

Bước 3: Sắp xếp cục bộ: Có thể dùng thuật insertion sort (do các phần tử không quá lớn) để sắp xếp lại cho từng phần lớp.

c) Độ phức tạp:

i. Trung bình: O(n)

ii. Tốt nhất:O(n)

iii. Xấu nhất: O(n)

d) *Độ phức tạp không gian*: $O(n)$ do cần tạo mảng tạm cho việc lưu các lớp.

3. Kết quả thực nghiệm và nhận xét

Dưới đây là 4 bảng kết quả thực nghiệm tương ứng với 4 kiểu dữ liệu đầu vào:

Bảng Data order: Randomize

DATA ORDER: RANDOMIZE												
Data size	10000		30000		50000		100000		300000		500000	
Result statics	Running Time(s)	Comparision	Running Time(s)	Comparision	Running Time(s)	Comparision	Running Time(s)	Comparision	Running Time(s)	Comparision	Running Time(s)	Comparision
Selection sort	0.165	100009999	1.246	900029999	3.224	2500049999	13.543	10000099999	123.55	90000299999	525.45	250000499999
Insertion sort	0.065	50493350	1.301	452649258	1.675	1244250134	7.133	4980194942	69.16	45068314958	268.95	125084179754
Bubble sort	0.302	100009999	5.343	900029999	8.954	2500049999	41.129	10000099999	396.13	90000299999	1766.61	250000499999
Shaker sort	0.266	75940975	2.987	675899100	7.572	1877846751	29.870	7507594224	597.95	67498799984	1505.14	187627934464
Shell sort	0.002	680982	0.006	2212168	0.013	4490319	0.030	9997781	0.23	35907489	0.36	67282186
Heap sort	0.002	660566	0.007	2219676	0.015	3886786	0.032	8275536	0.12	27177251	0.52	47123336
Merge sort	0.002	588019	0.006	1951857	0.011	3405429	0.024	7210727	0.09	23520303	0.39	40623963
Quick sort	0.001	295463	0.004	989785	0.007	1677216	0.016	3678835	0.06	11757019	0.19	20242090
Counting sort	0.000	69990	0.000	209999	0.001	349999	0.002	699999	0.01	2100002	0.03	3500002
Radix sort	0.001	140056	0.003	510070	0.005	850070	0.011	1700070	0.07	6000084	0.15	10000084
Flash sort	0.000	98846	0.001	288428	0.003	485857	0.007	995907	0.07	2884066	0.24	4657971

Bảng Data order: Nearly sorted

DATA ORDER: NEARLY SORTED												
Data size	10000		30000		50000		100000		300000		500000	
Result statics	Running Time(s)	Comparision	Running Time(s)	Comparision	Running Time(s)	Comparision	Running Time(s)	Comparision	Running Time(s)	Comparision	Running Time(s)	Comparision
Selection sort	0.294	100009999	2.372	900029999	4.958	2500049999	29.901	10000099999	257.91	90000299999	463.87	250000499999
Insertion sort	0.000	179870	0.001	356566	0.001	721966	0.004	1488790	0.01	4346866	0.01	8468882
Bubble sort	0.370	100009999	2.202	900029999	6.978	2500049999	35.033	10000099999	215.30	90000299999	521.61	250000499999
Shaker sort	0.001	299775	0.003	899775	0.003	1499775	0.006	2599831	0.02	8999775	0.03	14999775
Shell sort	0.001	409286	0.003	1305670	0.006	2391953	0.011	5128915	0.04	16377829	0.06	28112338
Heap sort	0.004	699101	0.009	2320491	0.018	4066591	0.040	8653821	0.11	28261651	0.19	48811431
Merge sort	0.002	514269	0.005	1664317	0.009	2937829	0.027	6176189	0.07	19649773	0.16	34372177
Quick sort	0.000	160903	0.001	518360	0.002	946653	0.006	1993254	0.02	6227192	0.03	10572916
Counting sort	0.000	70002	0.000	210002	0.000	350002	0.001	700002	0.00	2100002	0.01	3500002
Radix sort	0.001	140056	0.004	510070	0.009	850070	0.018	1700070	0.05	6000084	0.10	10000084
Flash sort	0.001	125761	0.003	377373	0.005	628969	0.010	1257967	0.02	3773967	0.04	6289969

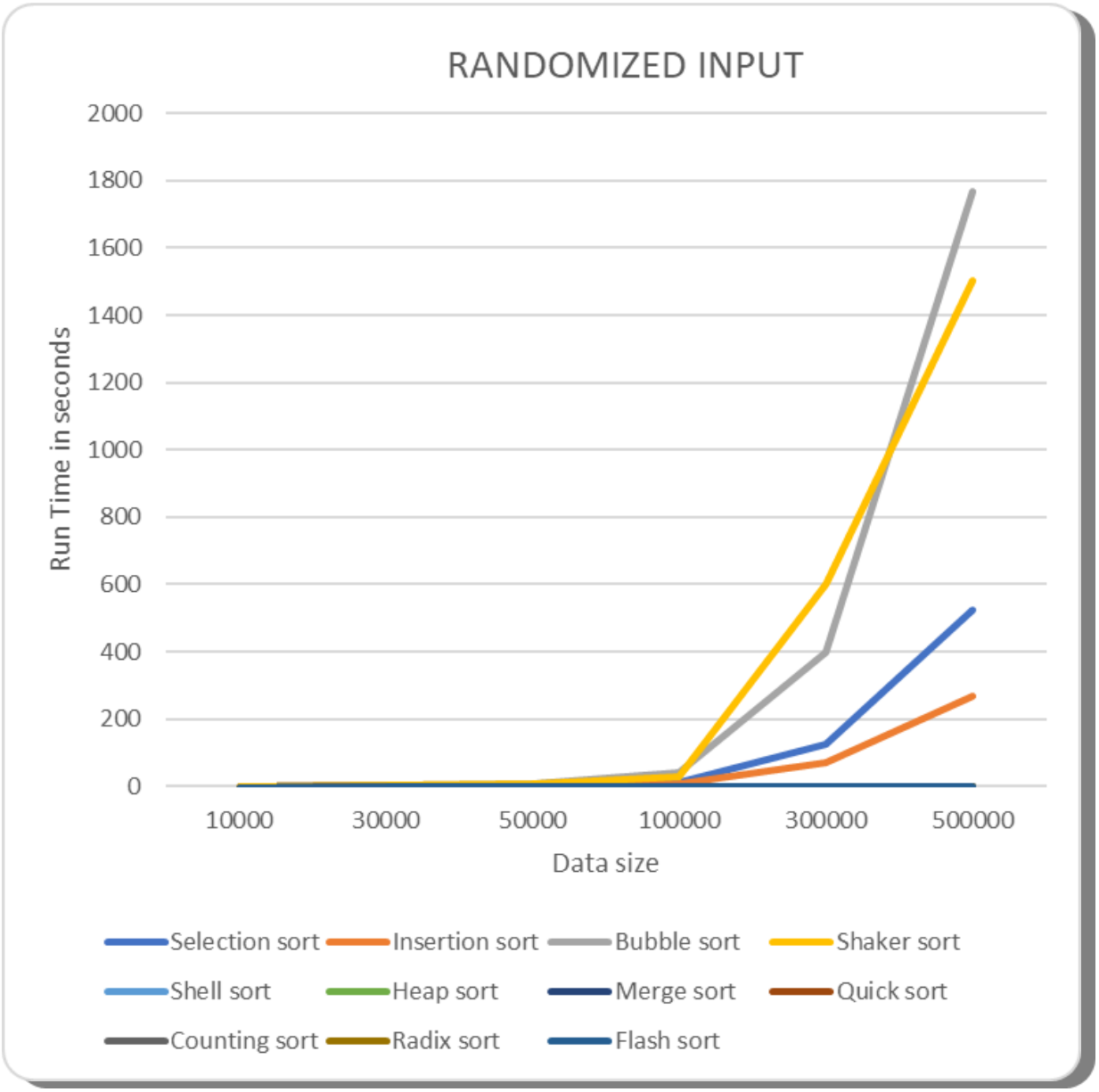
Bảng Data order: Sorted

DATA ORDER: SORTED												
Data size	10000		30000		50000		100000		300000		500000	
Result statics	Running Time(s)	Comparision	Running Time(s)	Comparision	Running Time(s)	Comparision	Running Time(s)	Comparision	Running Time(s)	Comparision	Running Time(s)	Comparision
Selection sort	0.182	100009999	1.609	900029999	4.275	2500049999	17.590	10000099999	117.91	90000299999	337.25	250000499999
Insertion sort	0.000	29998	0.000	89998	0.000	149998	0.001	299998	0.00	899998	0.00	1499998
Bubble sort	0.189	100009999	1.758	900029999	4.993	2500049999	18.902	10000099999	146.42	90000299999	347.89	250000499999
Shaker sort	0.000	19999	0.000	59999	0.000	99999	0.000	199999	0.00	599999	0.00	999999
Shell sort	0.000	360042	0.001	1170050	0.003	2100049	0.006	4500051	0.02	15300061	0.04	25500058
Heap sort	0.002	699781	0.011	2320501	0.014	4066521	0.024	8654271	0.08	28255801	0.15	48778501
Merge sort	0.001	485241	0.004	1589913	0.008	2772825	0.017	5845657	0.04	18945945	0.14	32517849
Quick sort	0.000	160863	0.001	518312	0.002	946617	0.003	1993226	0.01	6227156	0.02	10572876
Counting sort	0.000	70002	0.000	210002	0.000	350002	0.001	700002	0.00	2100002	0.01	3500002
Radix sort	0.001	140056	0.004	510070	0.007	850070	0.011	1700070	0.04	6000084	0.08	10000084
Flash sort	0.000	125797	0.002	377397	0.004	628997	0.005	1257997	0.02	3773997	0.04	6289997

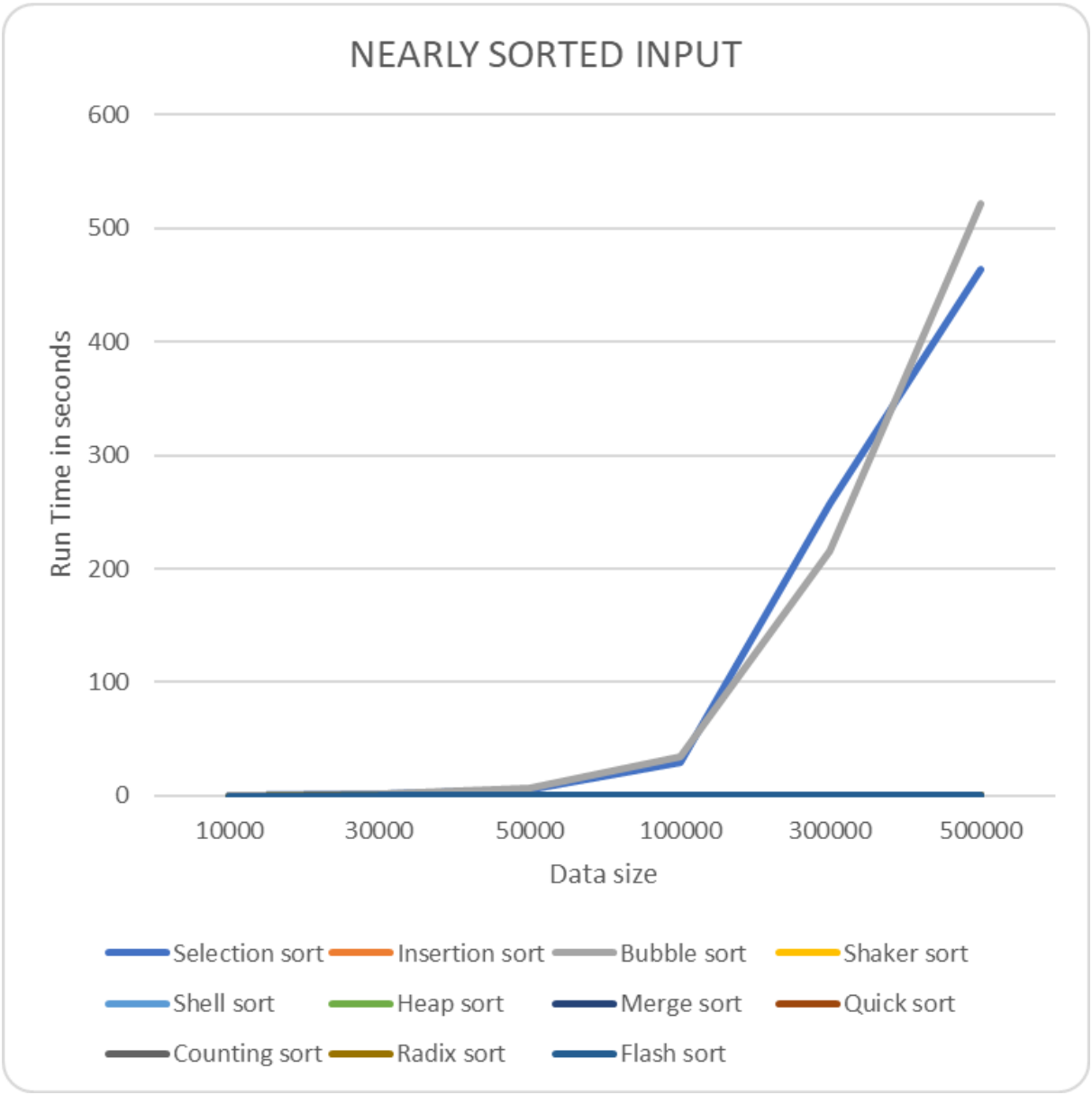
Bảng Data order: Reverse

DATA ORDER: REVERSE												
Data size	10000		30000		50000		100000		300000		500000	
Result statics	Running Time(s)	Comparision	Running Time(s)	Comparision	Running Time(s)	Comparision	Running Time(s)	Comparision	Running Time(s)	Comparision	Running Time(s)	Comparision
Selection sort	0.142	100009999	1.410	900029999	3.529	2500049999	14.184	10000099999	129.96	90000299999	323.43	250000499999
Insertion sort	0.160	100019998	1.464	900059998	4.694	2500099998	16.267	10000199998	171.26	90000599998	438.18	250000999998
Bubble sort	0.339	100009999	3.141	900029999	10.292	2500049999	35.418	10000099999	310.06	90000299999	1122.75	250000499999
Shaker sort	0.331	100000000	3.321	900000000	8.692	2500000000	34.121	10000000000	277.09	90000000000	1110.99	250000000000
Shell sort	0.000	485162	0.002	1584034	0.004	2894609	0.009	6189171	0.03	20301821	0.09	34357562
Heap sort	0.002	623481	0.007	2116061	0.013	3694461	0.025	7887171	0.08	26081231	0.25	45342251
Merge sort	0.001	476441	0.004	1573465	0.007	2733945	0.015	5767897	0.06	18708313	0.29	32336409
Quick sort	0.000	170880	0.001	548322	0.002	996628	0.004	2093238	0.01	6527178	0.04	11072890
Counting sort	0.000	70002	0.000	210002	0.000	350002	0.001	700002	0.00	2100002	0.01	3500002
Radix sort	0.000	140056	0.003	510070	0.006	850070	0.012	1700070	0.04	6000084	0.15	10000084
Flash sort	0.000	108600	0.001	325800	0.003	543000	0.006	1086000	0.02	3258000	0.06	5430000

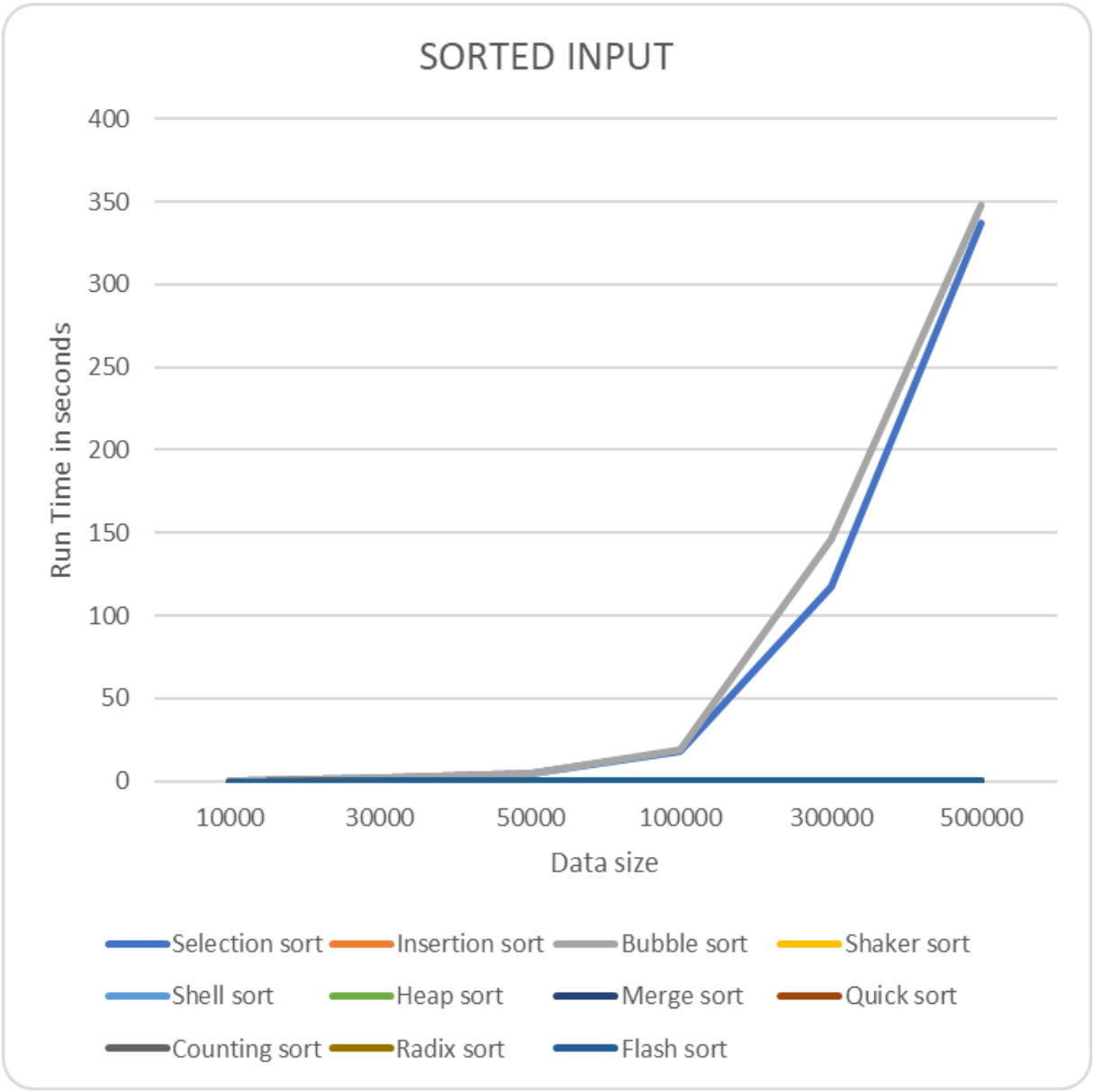
Để có cái nhìn trực quan, ta biểu diễn 4 bảng trên thành các đồ thị về thời gian và số phép so sánh dưới đây:



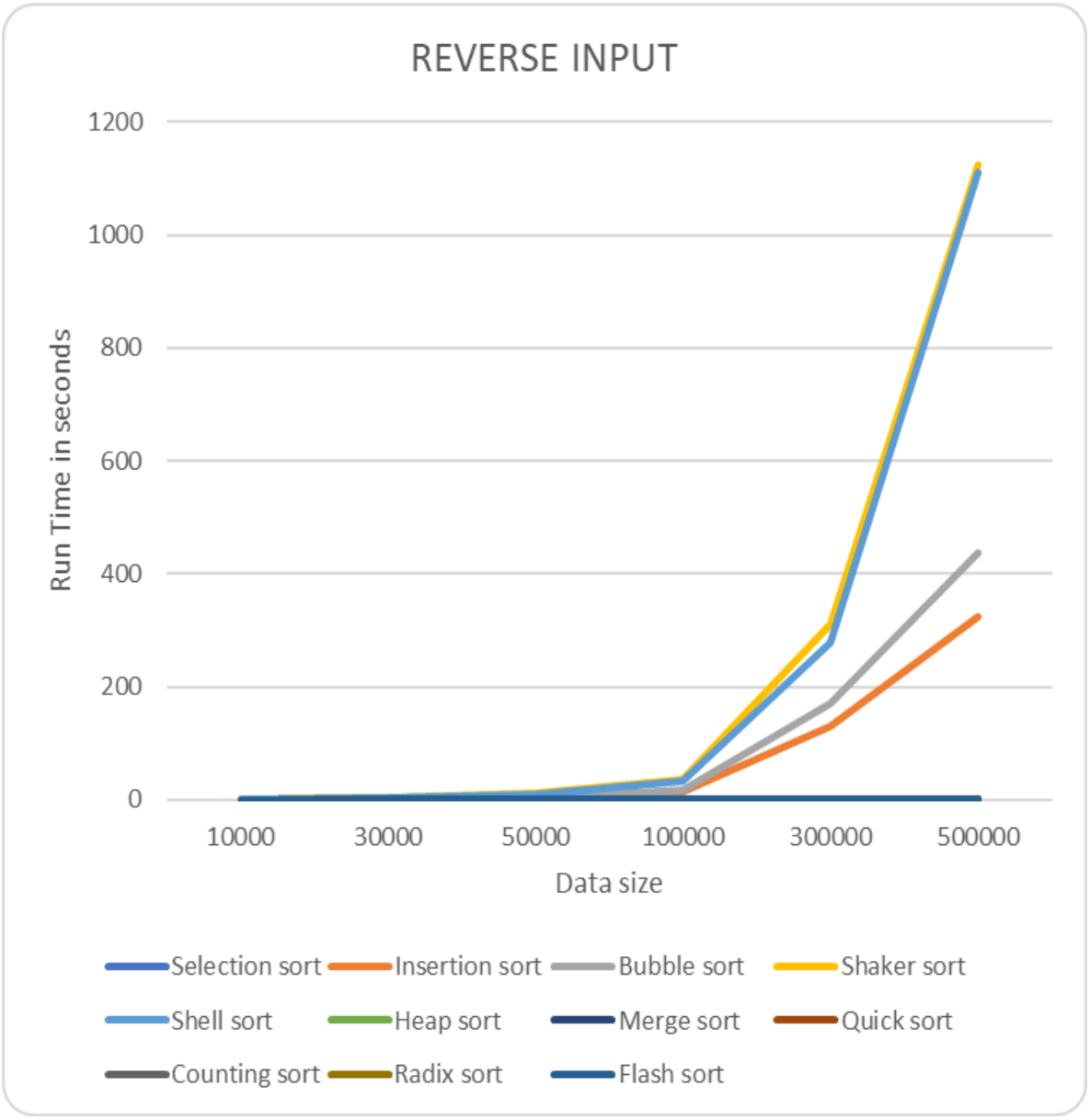
Nhận xét: Có thể thấy với một bộ dữ liệu bất kì, thì khi n rất lớn, lớn hơn 300000. Sự khác biệt của các nhóm thuật toán O(n^2) với O(nlogn) bắt đầu rõ rệt. Khi n bằng 500000, bubble sort chạy tốn thời gian nhất, khoảng 1800 giây. Shaker sort là biến thể của bubble sort nên nhanh hơn khoảng 1500 giây. Nhưng trong một số trường hợp shaker sort chậm hơn (tại 300000). Ngoài ra, 2 thuật toán selection sort và insertion sort tuy cùng nhóm thuật toán O(n^2) với bubble sort nhưng chạy nhanh hơn khá nhiều. Cụ thể, tại n bằng 500000, selection sort chạy trong khoảng 500 giây, insertion sort chạy trong khoảng 200 giây. Nhanh hơn 6 đến 9 lần so với bubble sort.



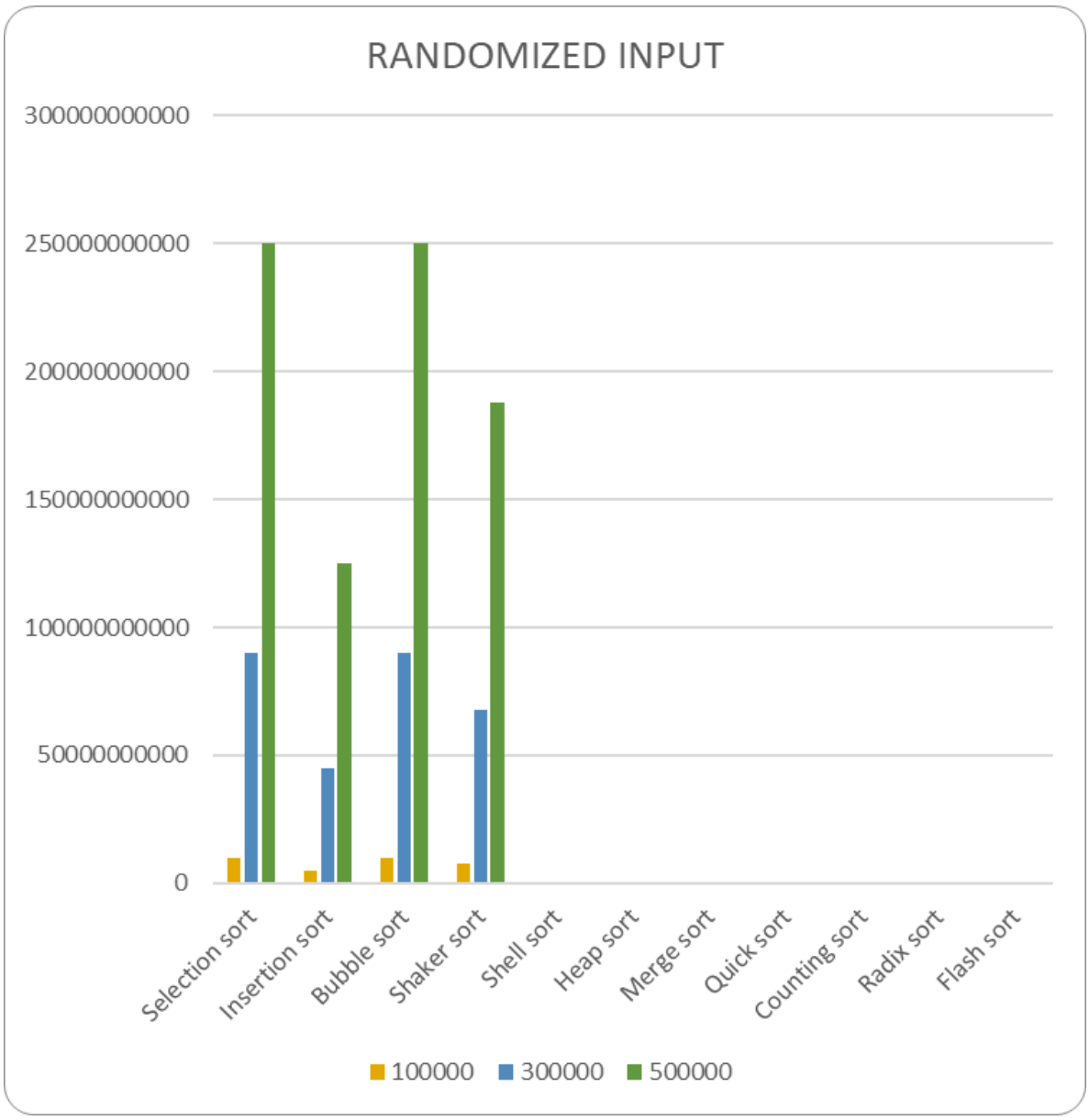
Nhận xét: Có thể thấy selection sort và bubble sort đều vươn lên dẫn đầu so với các thuật còn lại, khoảng 500 giây khi n bằng 500000. Trong khi đó, các thuật toán O(n^2) khác lại không tốn thời gian nhiều. Bởi vì cách cài đặt selection sort và bubble sort trong mã nguồn chưa có cải tiến. Cải tiến bằng cách, khi mảng đầu vào đã sắp xếp xong, hoặc không có gì thay đổi sau 1 lần lặp thì ta sẽ ngừng lại và không tiếp tục sort nữa. Sở dĩ, bubble sort và selection sort có thời gian tương đương nhau là do ý tưởng và cách cài đặt khá là giống nhau. Nhìn chung các thuật rất tốt khi chạy với Nearly sorted input khi đã cải tiến.



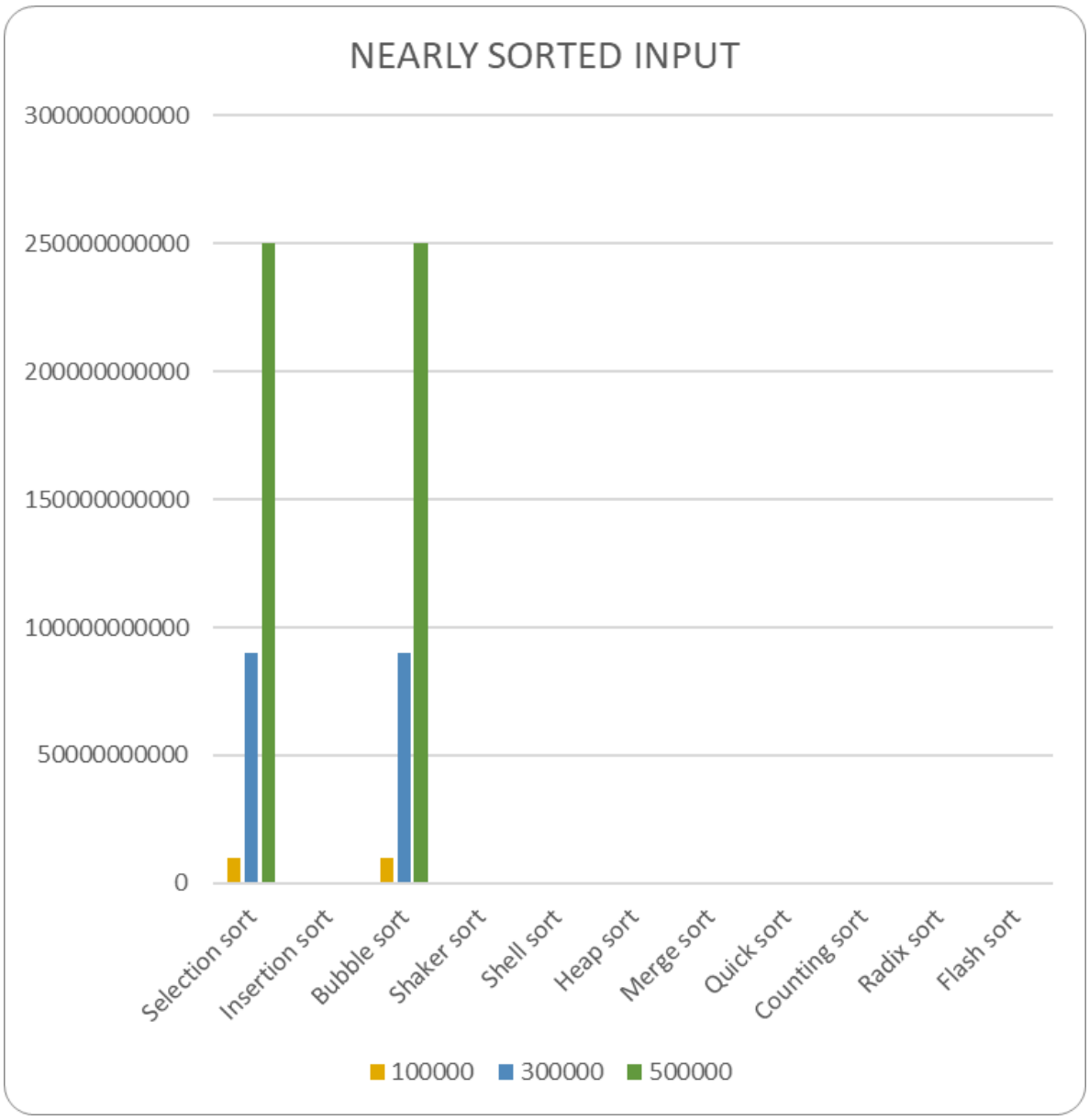
Tương tự như với Nearly sorted input, thuật toán bubble sort và selection sort chưa được cải tiến nên dẫn đến việc chạy lâu hơn so với các thuật O(n^2) khác. Các thuật toán không phải O(n^2) vẫn chạy ổn định.



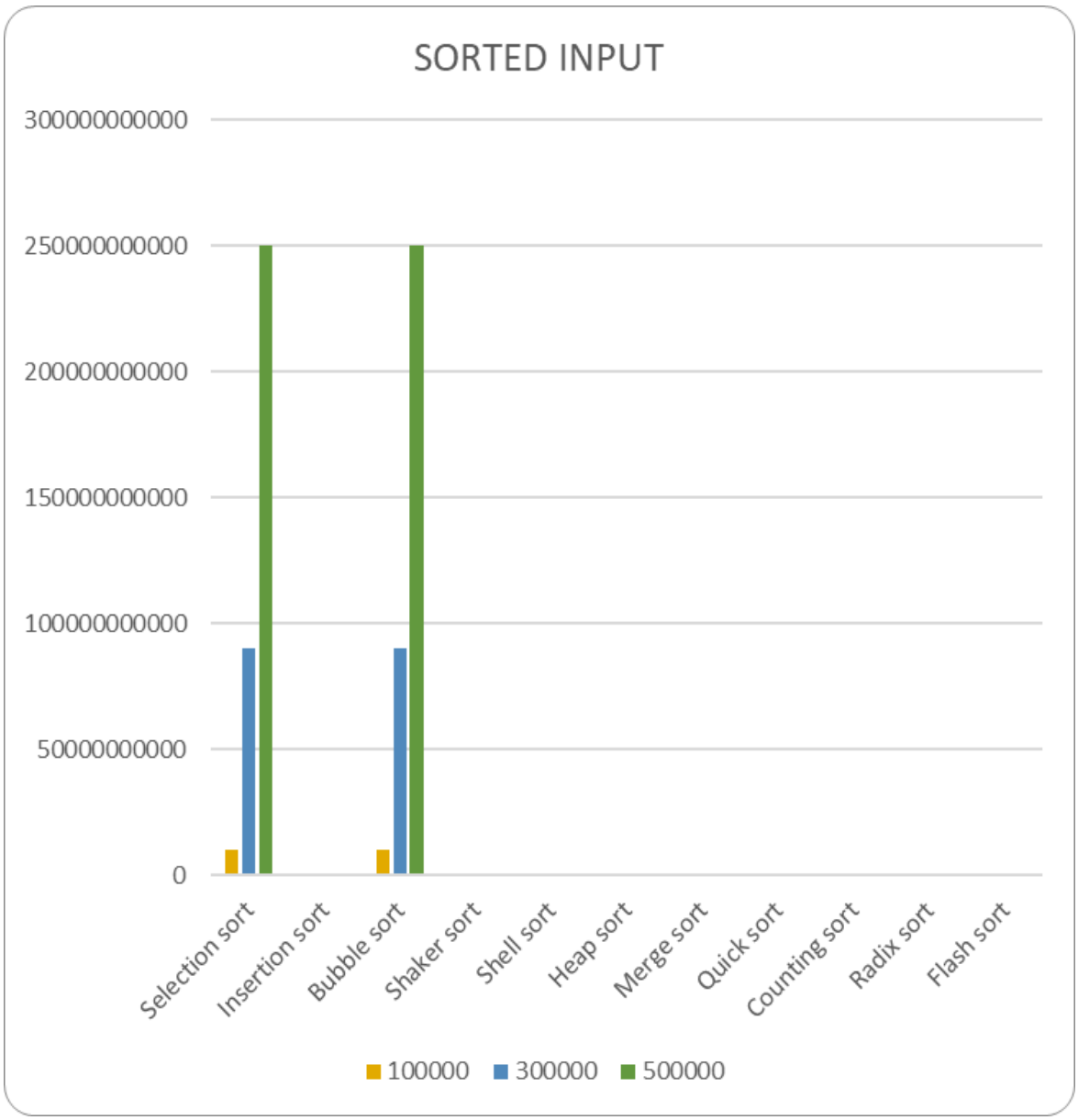
Nhận thấy rằng, khi thuật toán đã được sắp xếp sẵn nhưng ngược lại, 2 thuật toán shaker sort và insertion sort, tuy chạy tối ưu trong trường hợp được sắp xếp đúng thứ tự, nhưng vẫn chưa tối ưu khi thứ tự ngược lại. Ngoài ra, thuật shaker sort và selection sort chậm nhất, tốn khoảng 1100 giây. Trong khi 2 thuật toán O(n^2) là bubble sort và intertion sort chạy nhanh hơn gấp đôi, khoảng 400 giây. Các thuật toán không phải O(n^2) vẫn chạy ổn định.



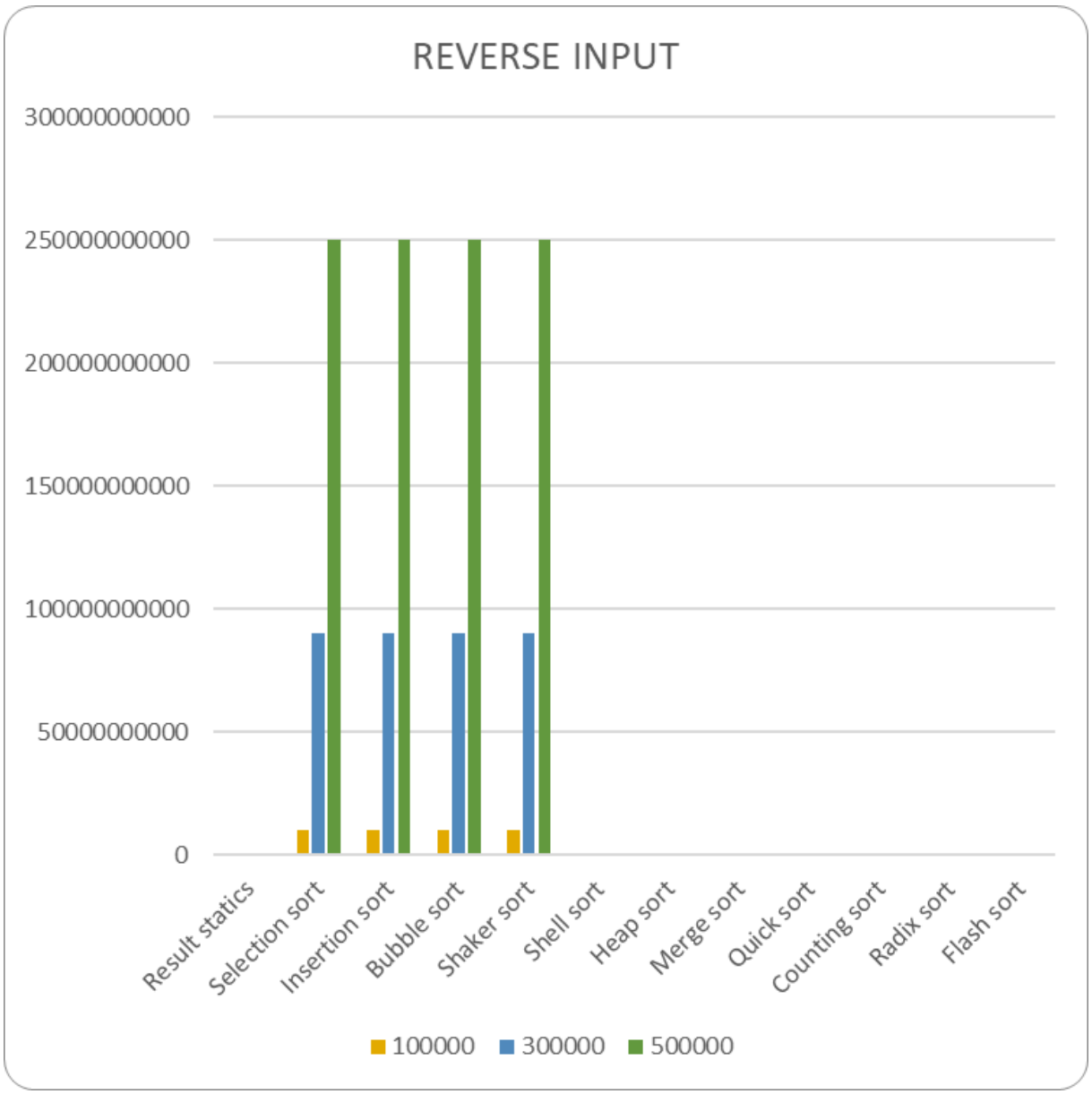
Nhận xét: Với dữ liệu đầu vào là 1 mảng bất kì, nhìn vào biểu đồ có thể thấy 4 thuật toán chạy tốn nhiều lần so sánh nhất chính là 4 thuật toán trong nhóm O(n^2). Hiển nhiên, vì thuật toán đơn giản dẫn đến việc chạy chậm hơn và so sánh nhiều hơn. Các thuật toán ngoài nhóm O(n^2) tối ưu hơn, nhanh hơn và so sánh ít hơn. Ít đến nỗi gần như không xuất hiện trên biểu đồ khi so sánh với các thuật toán O(n^2).



Với dữ liệu Nearly sorted input, có thể nhìn 2 thuật toán selection sort và bubble sort là có nhiều phép so sánh nhất. Điều trên là phù hợp với sự thật là 2 thuật toán trên cũng là 2 thuật toán chạy lâu nhất trong biểu đồ thời gian tương ứng. Và đương nhiên, khi 2 thuật toán trên được cải tiến, thì cũng sẽ ít phép so sánh hơn. Sở dĩ 2 thuật này có số phép so sánh khá giống nhau, bởi vì, giống như việc giống nhau về thời gian đã nêu ở trên.



Tương tự như với Nearly sorted input, thuật toán bubble sort và selection sort cần rất nhiều sự so sánh. Nếu cải tiến, sẽ ít hơn và gần như nhanh hơn các thuật toán khác.



Một điều bất ngờ, là số phép so sánh của 4 thuật toán nhiều nhất gần như là giống nhau và khi n bằng 500000, số phép so sánh rất cao lên đến khoản 250 tỷ lần so sánh. Do cả 4 thuật toán chạy trong 2 vòng lặp và mỗi vòng lặp chạy n lần nên có tổng cộng khoảng n^2 là 250 tỷ lần so sánh.

Nhận xét chung:

Nhìn chung, với số lượng phần tử nhỏ, khoảng 50 000, sự chênh lệch tốc độ cũng như là số phép so sánh giữa các thuật toán là không đáng kể, rất nhỏ.

Nhưng khi độ lớn lên đên 100 000, 500 000 hoặc hơn. Sự chênh lệch đó bắt đầu lộ ra rất rõ khi chạy bộ dữ liệu Randomize input. Qua đó, cho ta thấy độ khác biệt giữa các thuật toán O(n^2) và các thuật toán O(n log n).

Khi mảng đầu vào được sắp xếp sẵn, hoặc gần như sắp xếp thì đa số các thuật toán sẽ chạy tốt. Riêng với một số thuật toán O(n^2), cần phải cải tiến lên thì độ phức tạp đối với loại dữ liệu này gần như là O(n).

Riêng đối với mảng đầu vào là sắp xếp nhưng thứ tự ngược lại, thì các thuật O(n) vẫn chạy ổn định. Các thuật toán O(n^2) sẽ chạy chậm và tốn nhiều thời gian.

Trong các bài toán thực tế, nếu yêu cầu không chặt chẽ về thời gian hoặc thì có thể dùng thuật toán với độ phức tạp O(n^2) như selection sort, bubble sort,… Vì các thuật này cho ta sự ổn định.

Nếu cần sự yêu cầu chặt về thời gian thì có thể dùng các thuật toán với độ phức tạp nhanh hơn như quick sort, merge sort, heap sort,…

Nếu cần sự nhanh hơn nữa có thể sử dụng counting sort hoặc radix sort với độ phức tạp gần như O(n) nhưng phải đánh đổi vì sử dụng bộ nhớ nhiều và cẩn thận trường hợp giá trị rất lớn khi dùng counting sort.

Thuật toán flash sort có vẻ như đáp ứng được tất cả những điều trên với độ phức tạp nhanh, gần như O(n) và bộ nhớ ít nhưng vẫn có điểm yếu là cài đặt rất phức tạp. Có thể sử dụng trong các dự án khổng lồ, các dự án Big Data,….

4. Tổ chức file:

Do đề bài gồm các yêu cầu: thực hiện experimental, đếm thời gian các thuật toán, đếm số phép so sánh của các thuật toán, và đọc chương trình tư command line. Nên từ đó ta chia thành 4 file C++ header để xử lý các công việc. Ngoài ra, còn 1 file header có sẵn là DataGenerator.h mục đích để sinh ra dữ liệu cho mảng với 4 kiểu là sorted, nearly sorted, randomize và sorted reversed.

Experimental.h: để thực hiện yêu cầu chạy tất cả thuật toán và in những dữ liệu về thời gian chạy, số phép so sánh ra 1 file csv để dễ quản lý.

SortAlgorithm.h: để lưu các hàm để sắp xếp mảng sinh ra từ file DataGenerator.h với các loại thuật toán sắp xếp khác nhau. Các hàm này được dùng để đếm thời gian chạy của các thuật toán.

SortAlgorithmWithCount.h: để lưu tất cả các hàm sắp xếp nhưng mà có thêm biến đếm để đếm số lượng các phép so sánh mà mỗi thuật toán cần thực hiện. Còn về phần logic của code vẫn giống y chang trong file SortAlgorithm.h. Sở dĩ phải như thế, bởi vì, khi cho chạy thử với mảng nhỏ và có thêm biến đếm vào trong hàm, thời gian sẽ gần như tăng gấp đôi. Do đó, để tránh việc sai số về thời gian, ta cần 2 hàm sort, 1 để đo thời gian và 1 để đếm số phép so sánh.

CommandLine.h: để điều hướng các command nhập từ người dùng. Phân chia ra thành các “route” nhỏ hơn cho các command. Đồng thời, cũng thực hiện các logic của command đó.

Tài liệu tham khảo

https://www.geeksforgeeks.org/selection-sort/
https://www.geeksforgeeks.org/bubble-sort/
https://www.geeksforgeeks.org/insertion-sort/
https://www.geeksforgeeks.org/iterative-merge-sort/
https://www.geeksforgeeks.org/iterative-quick-sort/
https://www.geeksforgeeks.org/heap-sort/
https://www.geeksforgeeks.org/radix-sort/
https://www.geeksforgeeks.org/shellsort/
https://www.geeksforgeeks.org/cocktail-sort/
https://codelearn.io/sharing/flash-sort-thuat-toan-sap-xep-than-thanh