TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
KHOA CÔNG NGHỆ THÔNG TIN

BÁO CÁO ĐỒ ÁN SORTING

Môn học	**:**	**Cấu trúc dữ liệu và giải thuật**
Giảng viên hướng dẫn	**:**	**Lê Đình Ngọc**
Lớp	**:**	**21CTT5B**
Sinh viên thực hiện	**:**	**21120535 – Nguyễn Đình Hoàng Quân**
		21120537 - Trần Huỳnh Anh Quân
		21120551 – Đàm Tử Tâm
		21120559 – Nguyễn Ngọc Thiên

TP. HỒ CHÍ MINH, THÁNG 11 NĂM 2022

I. Thông tin chung

1. Thông tin nhóm

- Nhóm: 13
- Bộ thuật toán đã chọn: 7 thuật toán (Selection Sort, Insertion Sort, Bubble Sort, Heap Sort, Merge Sort, Quick Sort, Radix Sort)
- Số lượng sinh viên thực hiện: 4

Bảng thông tin sinh viên		
STT	***MSSV***	***Họ tên***
1	21120535	Nguyễn Đình Hoàng Quân
2	21120537	Trần Huỳnh Anh Quân
3	21120551	Đàm Tử Tâm
4	21120559	Nguyễn Ngọc Thiên

2. Phân chia công việc

MSSV	***Họ tên***	***Công việc***
21120535	Nguyễn Đình Hoàng Quân	– Insertion Sort, Merge Sort: viết các hàm tính thời gian chạy và số phép so sánh của hai thuật toán này; tính thời gian chạy và số phép so sánh của hai thuật toán này trên tất cả data order và data size; trình bày hai thuật toán này trong báo cáo – Điền bảng data order Nearly Sorted data; vẽ các biểu đồ thể hiện thời gian chạy và số phép so sánh của data order Nearly Sorted data, nhận xét các biểu đồ này – Viết command line arguments 4 – Tổ chức file nộp bài
21120537	Trần Huỳnh Anh Quân	– Bubble Sort, Heap Sort: viết các hàm tính thời gian chạy và số phép so sánh của hai thuật toán này; tính thời gian chạy và số phép so sánh của hai thuật toán này trên tất cả data order và data size; trình bày hai thuật toán này trong báo cáo – Điền bảng data order Sorted data; vẽ các biểu đồ thể hiện thời gian chạy và số phép so sánh của data order Sorted data, nhận xét các biểu đồ này – Viết command line arguments 3 – Tổng hợp, viết lại các ghi chú lập trình và tổ chức project

21120551	Đàm Tử Tâm	– Selection Sort, Quick Sort: viết các hàm tính thời gian chạy và số phép so sánh của hai thuật toán này; tính thời gian chạy và số phép so sánh của hai thuật toán này trên tất cả data order và data size; trình bày hai thuật toán này trong báo cáo – Điền bảng data order Randomized data; vẽ các biểu đồ thể hiện thời gian chạy và số phép so sánh của data order Randomized data, nhận xét các biểu đồ này – Viết command line arguments 5 – Viết trang giới thiệu trong báo cáo (mục I) – Tổng hợp code của các thành viên để tạo thành chương trình hoàn chỉnh – Viết template bài báo cáo – Nhận xét tổng thể cho phần thực nghiệm (mục 5 của mục III)
21120559	Nguyễn Ngọc Thiên	– Radix Sort: viết các hàm tính thời gian chạy và số phép so sánh của thuật toán này; tính thời gian chạy và số phép so sánh của thuật toán này trên tất cả data order và data size; trình bày thuật toán này trong báo cáo – Điền bảng data order Reversed data; vẽ các biểu đồ thể hiện thời gian chạy và số phép so sánh của data order Reversed data, nhận xét các biểu đồ này – Viết command line arguments 1, 2 – Viết trang bìa của bài báo cáo – Tổng hợp, viết lại các nguồn tài liệu tham khảo của các thành viên trong nhóm

3. Đánh giá mức độ hoàn thành công việc

Dựa trên sự phân chia công việc trong bảng trên và trong file Checklist.xlsx, đánh giá mức độ hoàn thành công việc của từng thành viên như sau:

MSSV	*Họ tên*	*Mức độ*	*Ghi chú*
21120535	Nguyễn Đình Hoàng Quân	100%	
21120537	Trần Huỳnh Anh Quân	100%	
21120551	Đàm Tử Tâm	100%	
21120559	Nguyễn Ngọc Thiên	100%	

II. Trình bày thuật toán

Theo yêu cầu của bài tập, tất cả các thuật toán sắp xếp phía dưới sẽ được trình bày theo hướng dùng để sắp xếp mảng tăng dần.

1. Selection Sort

a. *Ý tưởng*

- Selection Sort là một thuật toán dùng để sắp xếp một mảng bằng cách liên tục tìm phần tử có giá trị nhỏ nhất (xét theo sắp xếp mảng tăng dần) từ mảng con chưa được sắp xếp và đặt lên đầu mảng.
- Input là một mảng số A chưa được sắp xếp có n phần tử a_0, a_1, a_2,… a_{n-1}.
- Chọn phần tử a_i (i bắt đầu từ 0, i chạy từ 0 đến n – 2) rồi hoán vị a_i với phần tử nhỏ nhất trong mảng con từ a_{i+1} đến a_{n-1} nếu phần tử này nhỏ hơn a_i. Sau đó tăng giá trị của i lên 1 đơn vị. Lặp lại quá trình này đến khi i = n – 1 thì ngưng, thu được output là mảng số A đã được sắp xếp tăng dần.

b. *Mô tả từng bước (mã giả)*

- ***Bước 1:*** khởi tạo biến i = 0.
- ***Bước 2:*** cho i chạy từ 0 đến n – 2 bằng vòng lặp for. Bên trong vòng lặp của biến i, cứ mỗi lần lặp của biến i, ta có các bước sau:
  - ***Bước 2.1:*** gán biến minIndex = i (minIndex là biến lưu vị trí của các phần tử trong mảng).
  - ***Bước 2.2:*** khởi tạo biến j = i + 1 và cho biến j chạy từ i + 1 đến n – 1 bằng vòng lặp for. Cứ mỗi lần lặp của biến j, so sánh nếu phần tử A[j] < A[minIndex], gán giá trị của minIndex thành j.
  - ***Bước 2.3:*** thoát khỏi vòng lặp của biến j, hoán vị A [i] và A [minIndex].
- ***Bước 3:*** tiếp tục thực hiện bước 2 đến khi thoát khỏi vòng lặp for thì dừng chương trình, thuật toán kết thúc.

c. *Đánh giá độ phức tạp thuật toán*

- ***Độ phức tạp thời gian:*** $O(n^2)$ do Selection Sort là thuật toán dùng 2 vòng lặp lồng nhau.
- ***Độ phức tạp không gian:*** O(1) do thuật toán Selection Sort chỉ cần thêm vài vùng nhớ để lưu các biến tạm và các biến của vòng lặp, còn lại các phép tính đều thực hiện trên mảng ban đầu mà không khởi tạo thêm mảng mới nên không ảnh hưởng nhiều đến bộ nhớ.

d. *Biến thể/cải tiến của Selection Sort (nếu có)*

- Heap Sort
- Double Selection Sort (hay còn gọi là Cocktail Sort)
- Bingo Sort

2. Insertion Sort

a. Ý tưởng

- Insertion sort là thuật toán xắp xếp mảng bằng cách lần lượt chèn các phần tử vào mảng con đã được xắp xếp sao cho sau khi chèn không làm thay đổi thứ tự của mảng con.
- Bỏ qua phần tử đầu tiên, lần lượt chọn các phần tử tiếp theo làm đối tượng so sánh. Nếu phần tử đó và phần tử liền trước không theo đúng thứ tự thì hoán vị 2 phần tử này. Đổi đối tượng so sánh thành phần tử liền trước, lặp lại bước so sánh và hoán vị đến khi đối tượng và phần tử liển trước đúng thứ tự hoặc không còn phần tử liền trước đối tượng so sánh

b. Mô tả từng bước (mã giả)

- ***Bước 1:*** Khởi tạo biến key và j
- ***Bước 2:*** Vòng lặp cho i chạy từ 1 đến n – 1, i tăng dần. bên trong vòng lặp có các bước:

  + ***Bước 2.1:*** gán giá trị của phần tử thứ i trong mảng cho key.

  + ***Bước 2.2:*** gán giá trị bé hơn i một đơn vị cho j

  + ***Bước 2.3:*** lặp với điều kiện j không bé hơn 0 và phần tử thứ i của mảng lớn hơn key. Bên trong vòng lặp có các bước:

    - ***Bước 2.3.1:*** gán giá trị của phần tử thứ j trong mảng cho phần tử thứ j +1
    - ***Bước 2.3.2:*** j giảm đi một đơn vị.

  + ***Bước 2.4:*** gán giá trị của key cho phần tử thứ j + 1 trong mảng

c. Đánh giá độ phức tạp thuật toán

- ***Độ phức tạp thời gian:*** $O(n^2)$ do Insertion Sort là thuật toán dùng 2 vòng lặp lồng nhau. Tuy nhiên, nếu phần tử so sánh đã ở đúng vị trí thì vòng lặp sẽ kết thúc sớm. Khi đó, trong trường hợp tốt nhất (tất cả mọi phần tử đều ở đúng thứ tự xắp xếp) thì độ phức tạp thuật toán là O(n).
- ***Độ phức tạp không gian:*** O(1) do thuật toán Insertion Sort chỉ cần thêm vài vùng nhớ để lưu các biến tạm và các biến của vòng lặp, còn lại các phép tính đều thực

hiện trên mảng ban đầu mà không khởi tạo thêm mảng mới nên không ảnh hưởng nhiều đến bộ nhớ.

d. Biến thể/cải tiến của Insertion Sort (nếu có)

- Shell sort
- Binary Insertion Sort
- Binary Merge Sort
- Library Sort

3. Bubble Sort

a. Ý tưởng

- Bubble Sort là thuật toán sắp xếp bằng các thao tác là so sánh hai phần tử kề nhau, nếu chúng chưa đúng thứ tự (phần tử bên trái lớn hơn phần tử bên phải) thì đổi chỗ chúng. Sau mỗi lần lặp thì các phần tử lớn nhất sẽ được đưa về cuối dãy. Lặp lại cho đến khi dãy được sắp xếp xong.

b. Mô tả từng bước (mã giả)

Với dữ liệu đầu vào là mảng một chiều gồm N phần tử:

- **Bước 1:** Khởi tạo vòng lặp for với biến i chạy từ 0 đến N-2 (với N là số phần tử của mảng, vì mảng có N phần tử nên sau khi duyệt N-1 lần thì mảng sẽ được sắp xếp tăng dần xong.)
- **Bước 2:** Khởi tạo vòng lặp for với biến j chạy tử i đến 0 đến n - i –1 (Sau mỗi vòng lặp for ở **bước 1** thì phần tử lớn nhất của dãy đã được đưa về cuối dãy, nên trong vòng lặp for tiếp theo, ta sẽ phải đưa phần tử lớn nhất trong n-i phần tử còn lại về cuối dãy (cạnh phần tử lớn nhất vừa mới đưa về cuối dãy ở vòng lặp trước) nên sau mỗi dòng for ở bước 1 thì vòng for ở bước 2 sẽ giảm đi 1 lần so sánh, tức là j sẽ chỉ phải chạy đến n- i -1)
- **Bước 3:** So sánh phần tử thứ j và phần tử thứ j+1; nếu phần tử thứ j lớn hơn phần tử thứ j+1 thì ta đổi chỗ hai phần tử, rồi xét đến phần tử thứ j+1 và phần tử thứ j+2; nếu không thì xét đến phần tử thứ j+1 và phần tử thứ j+2.

c. Đánh giá độ phức tạp thuật toán

- Trong mọi trường hợp, ta đều phải duyệt qua tất cả các phần tử và phải lặp lại điều này cho đến khi tất cả các phần tử đều được sắp xếp.
- Độ phức tạp khi so sánh các cặp là O(n)
- Độ phức tạp khi lặp lại việc so sánh các cặp là O (n x n) = $O(n^2)$
  => Độ phức tạp của thuật toán là $O(n^2)$

d. Biến thể/cải tiến của Bubble Sort (nếu có)

4. Heap Sort

a. Ý tưởng

- Heap sort là thuật toán sắp xếp nhanh sử dũng kĩ thuật phân loại dựa trên cấu trúc cây nhị phân đặc biệt gọi là binary heap. Thuật toán dựa vào sự đặc biệt của cây nhị phân để lựa chọn ra phần tử lớn nhất rồi đưa phần tử này về cuối.

b. Mô tả từng bước (mã giả)

- Với dữ liệu đầu vào là mảng một chiều:
- **Bước 1: Tạo hàm hiệu chỉnh Max heap** cho Node tại vị trí i của mảng
  + **Bước 1.1:** tạo biến largest, biến này sẽ lưu vị trí của Node I (largest=I); tạo biến l (left) lưu vị trí của con bên trái Node I (l=2*I+1) và tạo biến r (Right) lưu vị trí của con bên phải Node I (r=2*I+2); Ta so sánh nếu giá trị tại Node cha I nhỏ hơn giá trị của hai Node con r và l thì largest sẽ bằng vị trí của Node con mang giá trị lớn nhất (r hoặc l)
  + **Bước 1.2:** Sau đó ta sẽ kiểm tra xem largest có bằng I hay không? Nếu bằng thì có nghĩa là không có Node con nào mang giá trị hơn Node cha I. Nếu không bằng tức là có giá trị Node con lớn hơn Node cha, lúc này ta sẽ hoán đổi chỗ 2 phần tử I và largest (largest lúc này mang vị trí của phần tử con có giá trị lớn nhất và lớn hơn phần tử Node cha) sau đó gọi lại **hàm hiệu chỉnh Max heap** cho phần tử tại vị trí largest.
- **Bước 2: Tạo hàm Heap Sort:**
  + **Bước 2.1:** Khởi tạo vòng for với biến chạy I chạy từ N/2-1 trở về 0 **(**Vì trong mảng 1 chiều, phần tử cha I sẽ có 2 phần tử con là 2*I+1 và 2*I+2 nên với mảng có N phần tử thì từ phần tử thứ N/2-1 trở về trước ([0]) mới có con, còn phần tử tử N/2 đến N-1 sẽ là con.), ta sẽ hiệu chỉnh Min heap cho phần tử cha ở Xa phần tử gốc nhất N/2-1 rồi mới quay ngược dần về phần tử gôc ([0]).
  + **Bước 2.2:** Sau khi xong **Bước 2.1** thì lúc này phần tử gốc ([0]) sẽ mang giá trị lớn nhất. Ta sẽ chuyển giá trị lớn nhất về cuối dãy, sau khi chuyển thì phần tử gốc lúc này sẽ không phải là Max heap nữa, ta sẽ gọi lại Hàm ở **Bước 1** để hiệu chỉnh phần tử gốc thành Max heap, sau khi hiệu chỉnh Max heap thì Node gốc lúc này lại mang giá trị lớn nhất, ta sẽ lại đổi chỗ phần tử gốc về cuối dãy (kế bên trái phần tử lớn nhất đã chuyển ở đầu **bước 2.2**)
  + **Bước 2.3:** vì sau mỗi bước 2.2 thì số phần tử cần xét sẽ giảm xuống 1 nên ta sẽ cần lặp lại N-1 lần (bằng cách tạo vòng for chạy từ N-1 trở về 0) cuối

cùng mảng sẽ được sắp xếp tăng dần .

c. Đánh giá độ phức tạp thuật toán

- Chiều cao của một cây nhị phân hoàn chỉnh chứa n phần tử là O(log(n)).
- Để tạo cấu trúc Heap cho một phần tử có các cây con đã là Max Heap, chúng ta cần tiếp tục so sánh phần tử với các phần tử con bên trái và bên phải của nó và đẩy nó xuống dưới cho đến kĐể tạo cấu trúc Heap cho một phần tử có các cây con đã là Max Heap, chúng ta cần tiếp tục so sánh phần tử với các phần tử con bên trái và bên phải của nó và đẩy nó xuống dưới cho đến khi nó đạt đến điểm mà cả hai cây con của nó đều nhỏ hơn nó.Để tạo cấu trúc Heap cho một phần tử có các cây con đã là Max Heap, chúng ta cần tiếp tục so sánh phần tử với các phần tử con bên trái và bên phải của nó và đẩy nó xuống dưới cho đến khi nó đạt đến điểm mà cả hai cây con của nó đều nhỏ hơn nó.
- Trong trường hợp xấu nhất, chúng ta sẽ cần phải di chuyển một phần tử từ nút gốc đến nút lá để thực hiện nhiều phép so sánh và hoán đổi O(log(n)).
- Trong giai đoạn xây dựng cấu trúc Max Heap, chúng ta đã thực hiện điều đó cho n/2 phần tử nên độ phức tạp trong trường hợp xấu nhất của bước này là $O((n/2)\times\log(n)) \approx O(n\log(n))$.
- Trong bước sắp xếp, chúng ta hoán đổi phần tử gốc với phần tử cuối cùng và tạo cấu trúc Heap cho phần tử gốc. Đối với mỗi phần tử, điều này lại làm tốn thời gian là O(log(n)) vì chúng ta có thể phải đưa phần tử đó từ nút gốc đến nút lá. Vì chúng ta lặp lại n lần này nên bước sắp xếp vun đống cũng là O(nlog(n)).
- Cũng vì các bước xây dựng cấu trúc Max Heap và sắp xếp vun đống được thực hiện lần lượt nên độ phức tạp của thuật toán không được nhân lên và nó vẫn theo thứ tự O(nlog(n)).

d. Biến thể/cải tiến của Heap Sort (nếu có)

5. Merge Sort

a. Ý tưởng:

- Merge Sort là một thuật toán sắp xếp dựa trên giải thuật chia để trị. Mảng được chia thành hai nửa liên tục, tới khi mảng con còn 1 phần tử. Sau đó lần lượt kết hợp từng cặp 2 mảng con cho đến khi chỉ còn một mảng duy nhất chính là mảng đã được sắp xếp.

b. Mô tả từng bước (mã giả)

- **Bước 1:** đệ quy xắp xếp nửa mảng bên trái

- **Bước 2:** đệ quy sắp xếp nửa mảng bên phải
- **Bước 3:** ghép 2 mảng đã xắp xếp về 1 mảng
  + **Bước 3.1:** chia mảng đã cho ra thành 2 mảng con tempL và tempR
    - **Bước 3.1.1:** tìm số lượng phần tử sizeL, sizeR của 2 mảng con
    - **Bước 3.1.2:** khởi tạo tempL, tempR. Copy các phần tử nửa bên trái của mảng cha vào tempL. Copy các phần tử nửa bên phải mảng cha vào tempR
  + **Bước 3.2:** khởi tạo các biến chỉ số:
    - Biến i: chỉ số trong mảng con bên trái.
    - Biến j: chỉ số trong mảng con bên phải.
    - Biến k: chỉ số trong mảng cha

      Biến i,j chạy từ 0; riêng k chạy từ chỉ số bên trái cùng của mảng cha.
  + **Bước 3.3:** ghép 2 mảng:
    - **Bước 3.3.1:** chạy vòng lặp điều kiện i bé hơn sizeL và j bé hơn sizeR: Phần tử của mảng con bên nào lớn hơn thì gán phần tử thứ k của a bằng phần tử đó. Đồng thời tăng biến chạy k và biến chạy của bên tương ứng thêm 1.
    - **Bước 3.3.2:** sau khi vòng lặp trên kết thúc, nếu sizeL khác sizeR thì vẫn còn sót lại một số phần tử của mảng con có kích thước lớn hơn. Chạy 2 vòng lặp riêng biệt kiểm tra nếu biến chạy của mảng con vẫn bé hơn kich thước mảng con tương ứng thì sẽ thêm các phần tử dư đó tiếp nối mảng đã xắp xếp.

c. Đánh giá độ phức tạp thuật toán

- ***Độ phức tạp thời gian:***
  + Chia đôi mảng có độ phức tạp $O(logn)$
  + Trộn mảng có độ phức tạp $O(n)$

    => $O(n * logn)$
- ***Độ phức tạp không gian:***
  + Tạo 2 mảng con có kích thước tổng bằng mảng gốc

    => $O(n)$.
- ***Đánh giá:*** độ phức tạp thuật toán luôn không đổi vì dù dữ liệu có được xếp theo cách nào thì Merge Sort luôn chia đôi mảng, so sánh và ghép mảng.

d. Biến thể/cải tiến của Merge Sort (nếu có)

- Bitonic Merge Sort
- 3-way Merge Sort

6. Quick Sort

a. Ý tưởng

- Giống như Merge Sort, thuật toán sắp xếp Quick Sort là một thuật toán áp dụng kỹ thuật chia để trị. Nó chọn một phần tử trong mảng làm pivot, sau đó thuật toán sẽ thực hiện chia mảng thành các mảng con dựa vào pivot đã chọn. Có nhiều phiên bản khác nhau của QuickSort chọn pivot theo những cách khác nhau, dưới đây là một số cách chọn pivot:
  + Luôn chọn phần tử đầu tiên của mảng (được dùng trong bài tập này).
  + Luôn chọn phần tử cuối cùng của mảng.
  + Chọn một phần tử random.
  + Chọn một phần tử có giá trị nằm giữa mảng (median element).
- Đầu tiên, ta phân đoạn dãy số: cho input là một mảng số A gồm n phần tử a_0, a_1, a_2,… a_{n-1} và một phần tử x là pivot, đặt x vào đúng vị trí của mảng đã sắp xếp. Di chuyển tất cả các phần tử của mảng mà nhỏ hơn x sang bên trái vị trí của x, và di chuyển tất cả các phần tử của mảng mà lớn hơn x sang bên phải vị trí của x.
- Sau bước trên ta sẽ có 2 mảng con: mảng bên trái của x và mảng bên phải của x. Lặp lại công việc trên với mỗi mảng con (chọn pivot, phân đoạn) cho tới khi cả mảng được sắp xếp tăng dần.

b. Mô tả từng bước (mã giả)

- ***Bước 1:*** phân đoạn và sắp xếp dãy số theo pivot: cho mảng A gồm n phần tử a_0, a_1, a_2,… a_{n-1}, vị trí đầu của mảng A, vị trí cuối của mảng A
  + ***Bước 1.1:*** xác định vị trí của pivot: chọn pivot là phần tử đầu tiên trong mảng, dùng một vòng lặp for duyệt qua các phần tử đi từ vị trí thứ (vị trí của pivot + 1) đến vị trí cuối của mảng để tìm ra số lượng phần tử bé hơn hoặc bằng pivot trong mảng. Sau đó, hoán vị pivot với phần tử thứ (vị trí của pivot + số lượng phần tử bé hơn hoặc bằng pivot). Lúc này pivot đã được đặt đúng chỗ, lưu lại vị trí của pivot.
  + ***Bước 1.2:*** khởi tạo biến i, gán biến i là vị trí đầu tiên của mảng. Khởi tạo biến j, gán biến j là vị trí cuối cùng của mảng.
  + ***Bước 1.3:*** khởi tạo vòng lặp while với điều kiện chạy là i < vị trí của pivot và j > vị trí của pivot. Bên trong vòng lặp gồm các bước sau:
    - ***Bước 1.3.1:*** khởi tạo vòng lặp while với điều kiện chạy là a[i] <= pivot, với mỗi lần lặp thì tăng giá trị của biến i lên một đơn vị
    - ***Bước 1.3.2:*** khởi tạo vòng lặp while với điều kiện chạy là a[j] > pivot, với mỗi lần lặp thì tăng giá trị của biến j lên một đơn vị
    - ***Bước 1.3.3:*** sau hai bước trên, nếu i < vị trí của pivot và j > vị trí của pivot nghĩa là ta tìm được một phần tử lớn hơn pivot ở mảng con bên

trái pivot và một phần tử bé hơn pivot ở mảng con bên phải pivot. Tiến hành hoán vị hai phần tử này để đưa chúng về đúng vị trí.

+ ***Bước 1.4:*** tiếp tục thực hiện bước 1.3 đến khi thoát khỏi vòng lặp while. Sau đó trả về vị trí của pivot

– ***Bước 2:*** đệ quy để sắp xếp mảng bằng Quick Sort: cho mảng A gồm n phần tử a_0, a_1, a_2,… a_{n-1}, vị trí đầu của mảng A, vị trí cuối của mảng A. So sánh nếu vị trí đầu của mảng < vị trí cuối của mảng thì tiến hành các bước sau:
  + ***Bước 2.1:*** thực hiện bước 1. Đồng thời, sau bước này ta có được vị trí của pivot
  + ***Bước 2.2:*** gọi đệ quy để thực hiện bước 2 đối với mảng con bên trái pivot
  + ***Bước 2.3:*** gọi đệ quy để thực hiện bước 2 đối với mảng con bên phải pivot

c. Đánh giá độ phức tạp thuật toán

– ***Độ phức tạp thời gian:***
  + ***Trường hợp tốt nhất:*** O(n*log(n)). Trường hợp này xảy ra khi ta chọn pivot là phần tử có giá trị nằm giữa mảng (median element). Khi đó, Quick Sort sẽ tốn chi phí là O(log(n)) cho việc chia đôi mảng liên tục. Trong mỗi mảng được chia đôi, Quick Sort lại tốn chi phí là O(n) cho việc duyệt qua mảng đó để sắp xếp mảng xoay quanh pivot. Do đó, độ phức tạp thời gian cho trường hợp này là O(n*log(n)).
  + ***Trường hợp xấu nhất:*** $O(n^2)$. Trường hợp này xảy ra khi ta chọn trúng pivot là phần tử có giá trị nhỏ nhất hoặc lớn nhất trong mảng. Nếu ta luôn chọn pivot là phần tử cuối cùng hoặc phần tử đầu tiên trong mảng, trường hợp xấu nhất sẽ xảy ra nếu mảng đầu vào đã được sắp tăng dần hoặc giảm dần. Khi đó, Quick Sort sẽ tốn chi phí là O(n) cho việc chia đôi mảng liên tục. Trong mỗi mảng được chia đôi, Quick Sort lại tốn chi phí là O(n) cho việc duyệt qua mảng đó để sắp xếp mảng xoay quanh pivot. Do đó, độ phức tạp thời gian cho trường hợp này là $O(n^2)$.
  + ***Trường hợp trung bình:*** O(n*log(n)). Trường hợp này xảy ra khi mảng input có các phần tử sắp xếp lộn xộn, không có thứ tự tăng dần cũng không có thứ tự giảm dần, do đó sẽ không rơi vào trường hợp xấu nhất đã trình bày bên trên. Khi đó, Quick Sort sẽ tốn chi phí là O(log(n)) cho việc chia đôi mảng liên tục. Trong mỗi mảng được chia đôi, Quick Sort lại tốn chi phí là O(n) cho việc duyệt qua mảng đó để sắp xếp mảng xoay quanh pivot. Do đó, độ phức tạp thời gian cho trường hợp này là O(n*log(n)) .

– ***Độ phức tạp không gian:*** vì thuật toán Quick Sort có gọi đệ quy nên đối với mỗi lần gọi đệ quy, thuật toán sẽ tốn thêm bộ nhớ để tạo stack. Lượng bộ nhớ cho

stack sẽ tăng theo số lần chia mảng theo pivot. Do đó, ta có thể chia độ phức tạp không gian của Quick Sort thành hai trường hợp sau:

+ ***Trường hợp trung bình:*** O(log(n)). Trường hợp này xảy ra khi độ phức tạp thời gian cho việc chia đôi mảng là O(log(n)).
+ ***Trường hợp xấu nhất:*** O(n). Trường hợp này xảy ra khi độ phức tạp thời gian cho việc chia đôi mảng là O(n).

d. Biến thể/cải tiến của Quick Sort (nếu có)

- Multi-pivot quicksort
- External quicksort
- Three-way radix quicksort
- Quick radix sort
- BlockQuicksort
- Partial and incremental quicksort

7. Radix Sort

a. Ý tưởng:

- Radix Sort là thuật toán sắp xếp không dựa vào so sánh mà dựa vào phân bố dữ liệu. Với mỗi bước, thuật toán sẽ nhóm các chữ số riêng lẻ của một giá trị có cùng một vị trí, sau đó sắp xếp các phần tử theo thứ tự tăng hoặc giảm dần. Thuật toán có 2 cách sắp xếp:
  + Sắp xếp từ hàng đơn vị lên.
  + Sắp xếp từ hàng trên cùng xuống.
- Cách hoạt động của thuật toán:
  + Tìm phần tử lớn nhất trong mảng để xác định được số chữ số của số lớn nhất, qua đó xác định được số lần phải lặp lại thuật toán.
  + Bắt đầu từ hàng đơn vị, sử dụng hàm CountSort để xếp các phần tử dựa theo vị trí giá trị của hàng đơn vị. Sau đó lặp lại với các hàng lớn hơn cho đến khi đạt được số lần phải lặp (đã được xác định ở bước trên).

b. Mô tả từng bước (mã giả)

- **Bước 1:** Tìm giá trị max trong mảng.
- **Bước 2:** Hàm Count Sort
  + **Bước 1.1:** Khởi tạo mảng đếm có 10 phần tử với tất cả giá trị bằng 0, khởi tạo mảng xuất có số phần tử bằng với kích thước của mảng.

+ **Bước 1.2:** Dùng vòng lặp For từ i = 0 tới giá trị kích thước của mảng trừ đi 1. Đếm các số có cùng giá trị ở hàng bé nhất bằng cách tăng giá trị của phần tử trong mảng đếm lên 1 đơn vị mỗi khi gặp chỉ số tương ứng.
+ **Bước 1.3:** Dùng vòng lặp For từ i = 1 tới 9. Tính tổng tích lũy count[i] += count[i - 1] để xác định được vị trí thực tế của từng phần tử trên mảng xuất.`
+ **Bước 1.4:** Sử dụng vòng lặp For từ i = n – 1 về 0. Gán giá trị từ mảng ban đầu vào mảng xuất, chỉ số vị trí trên mảng xuất đã được xác định ở trên. Sau mỗi lần sắp xếp thì giảm giá trị count ở vị trí đó đi 1.
+ **Bước 1.5:** Dùng vòng lặp For từ i = 0 tới giá trị kích thước của mảng trừ đi 1, gán giá trị từ mảng xuất về lại mảng ban đầu.

– **Bước 3:** Hàm Radix Sort
  + **Bước 3.1:** Lấy giá trị max của mảng đã được xác định ở trên.
  + **Bước 3.2:** Gọi đệ quy hàm Counting sort với số lần bằng với số chữ số của giá trị max.

c. Đánh giá độ phức tạp thuật toán

– **Độ phức tạp thời gian:**
  + **Trong trường hợp xấu nhất:** Trong Radix Sort, trường hợp xấu nhất là khi tất cả các phần tử có cùng số chữ số ngoại trừ một phần tử có số lượng chữ số lớn đáng kể. Nếu số chữ số trong phần tử lớn nhất bằng n, thời gian chạy là $O(n^2)$.
  + **Trong trường hợp trung bình:** Có p chữ số và mỗi chữ số có thể có tối đa d giá trị khác nhau (là hệ cơ số), n là số phần tử trong mảng đầu vào. Radix Sort có độ phức tạp thời gian trường hợp trung bình là O(p(n+d)).
  + **Trong trường hợp tốt nhất:** Khi tất cả các phần tử có cùng số chữ số, trường hợp tốt nhất xảy ra. O(p(n+d)) là độ phức tạp thời gian trong trường hợp tốt nhất. Nếu d bằng O(n), độ phức tạp thời gian là O(p*n).
– **Độ phưc tạp không gian:** Bởi vì Radix Sort sử dụng count sort, sử dụng các mảng phụ có kích thước n và d, trong đó n là số phần tử trong mảng đầu vào và d là số giá trị khác nhau mà mỗi chữ số có thể có. Do đó, Radix Sort có độ phức tạp không gian là O(n+d).

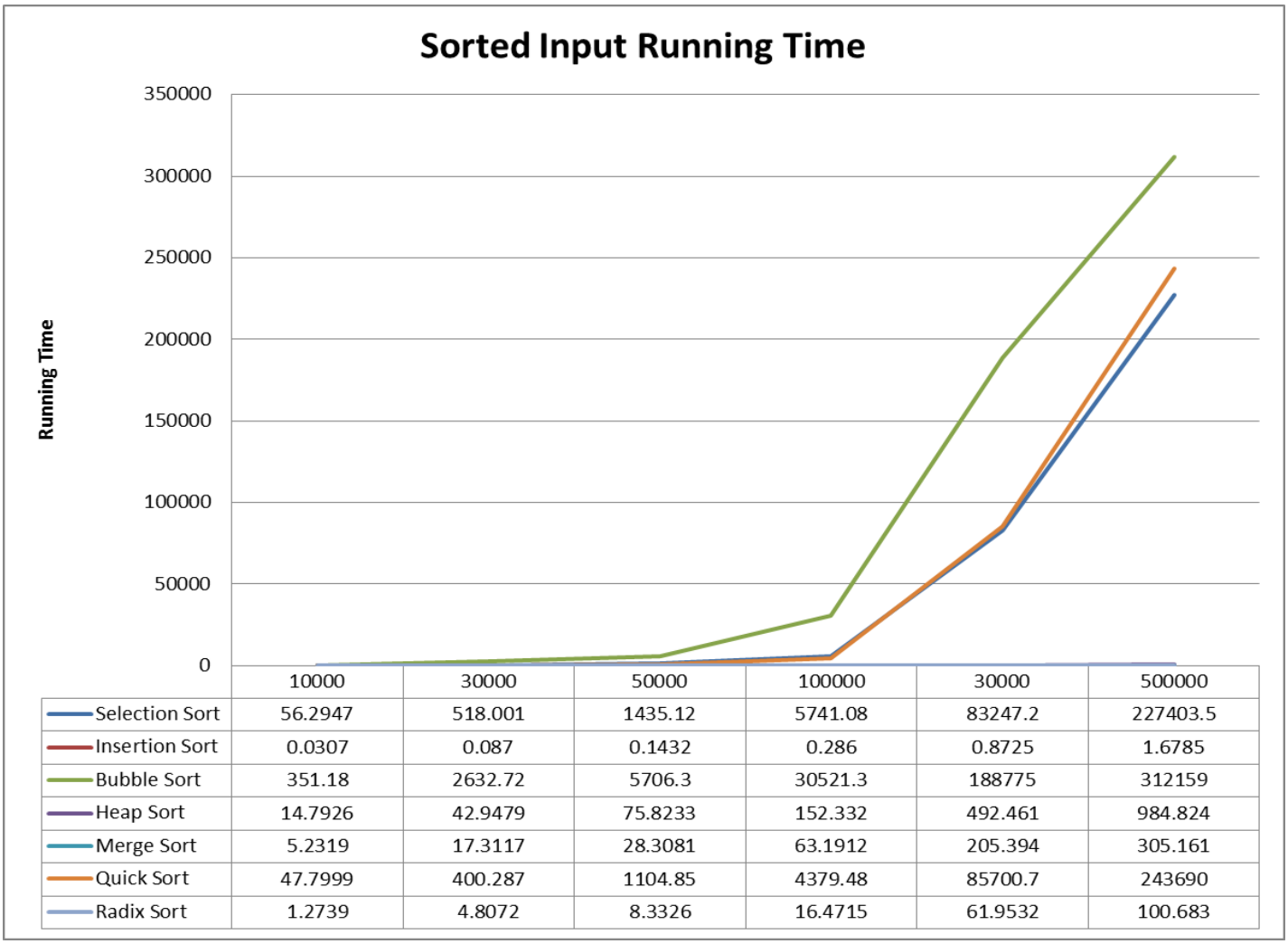
d. Biến thể/cải tiến của Radix Sort (nếu có)

III. Kết quả thực nghiệm và nhận xét

Phần "Running time" có đơn vị là milliseconds. Các kết quả thực nghiệm phía dưới được đo thông qua dùng các thuật toán để sắp xếp mảng ***tăng dần****.*

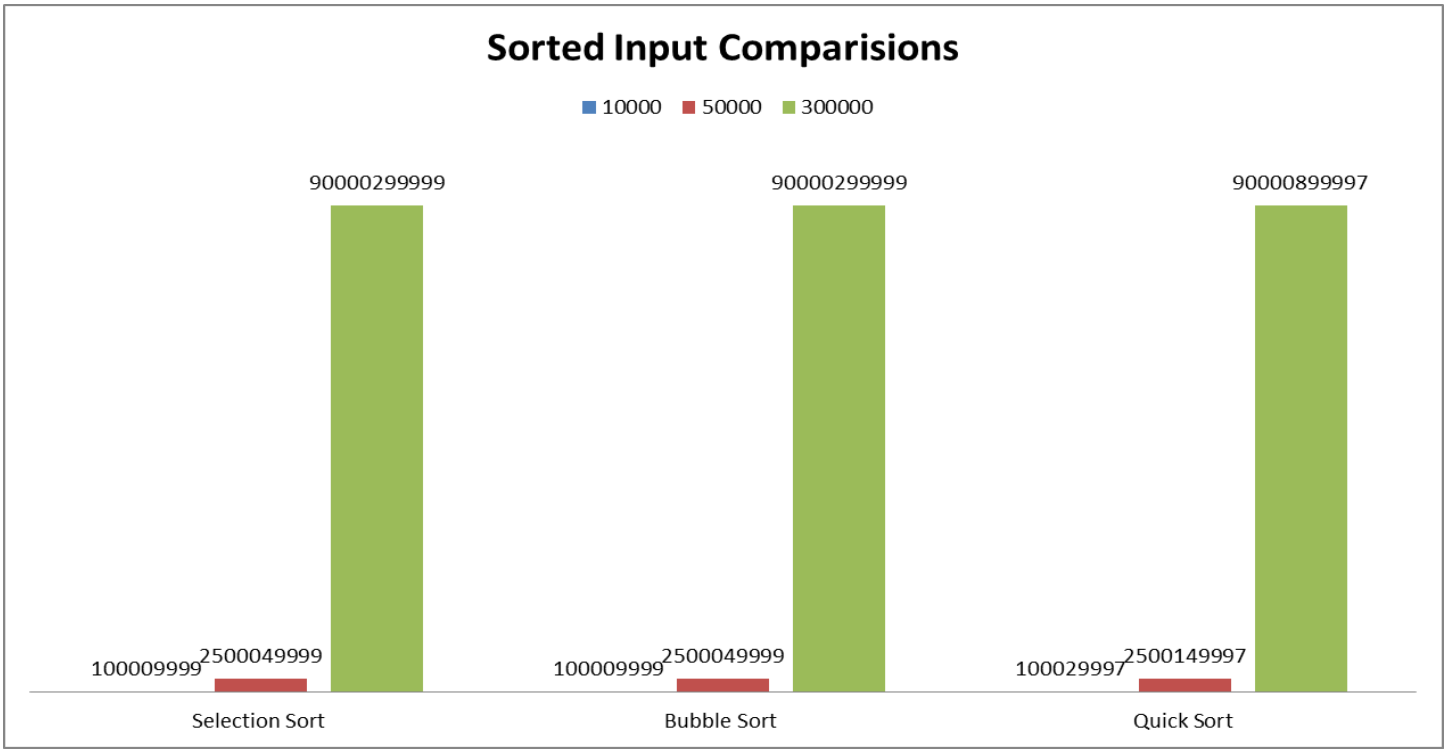
1. Sorted data

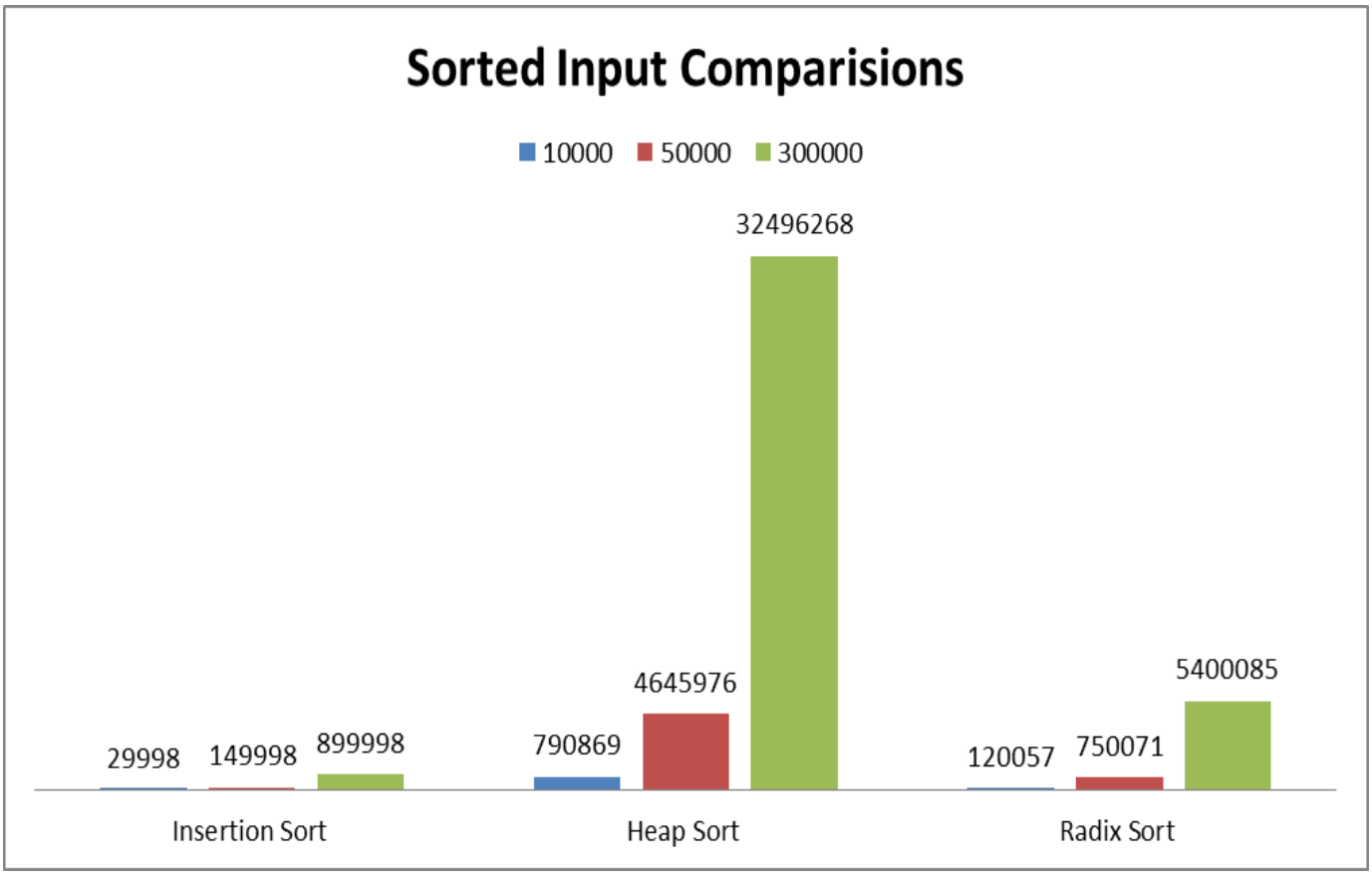
Data order: **Sorted data**												
Data size	10,000		30,000		50,000		100,000		300,000		500,000	
Resulting statics	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision
Selection Sort	56.2947	100009999	518.001	900029999	1435.12	2500049999	5741.08	10000099999	83247.2	90000299999	227403.5	250000499999
Insertion Sort	0.0307	29998	0.087	89998	0.1432	149998	0.286	299998	0.8725	899998	1.6785	1499998
Bubble Sort	351.18	100009999	2632.72	900029999	5706.3	2500049999	30521.3	10000099999	`188775	90000299999	312159	250000499999
Heap Sort	14.7926	790869	42.9479	2643913	75.8233	4645976	152.332	9908762	492.461	32496268	984.824	56228760
Merge Sort	5.2319	475242	17.3117	1559914	28.3081	2722826	63.1912	5745658	205.394	18645946	305.161	32017850
Quick Sort	47.7999	100029997	400.287	900089997	1104.85	2500149997	4379.48	10000299997	85700.7	90000899997	243690	250001499997
Radix Sort	1.2739	120057	4.8072	450071	8.3326	750071	16.4715	1500071	61.9532	5400085	100.683	9000085



Nhận xét:

- Thuật toán có thời gian chạy ngắn nhất: Insertion Sort
- Thuật toán có thời gian chạy dài nhất: Bubble Sort



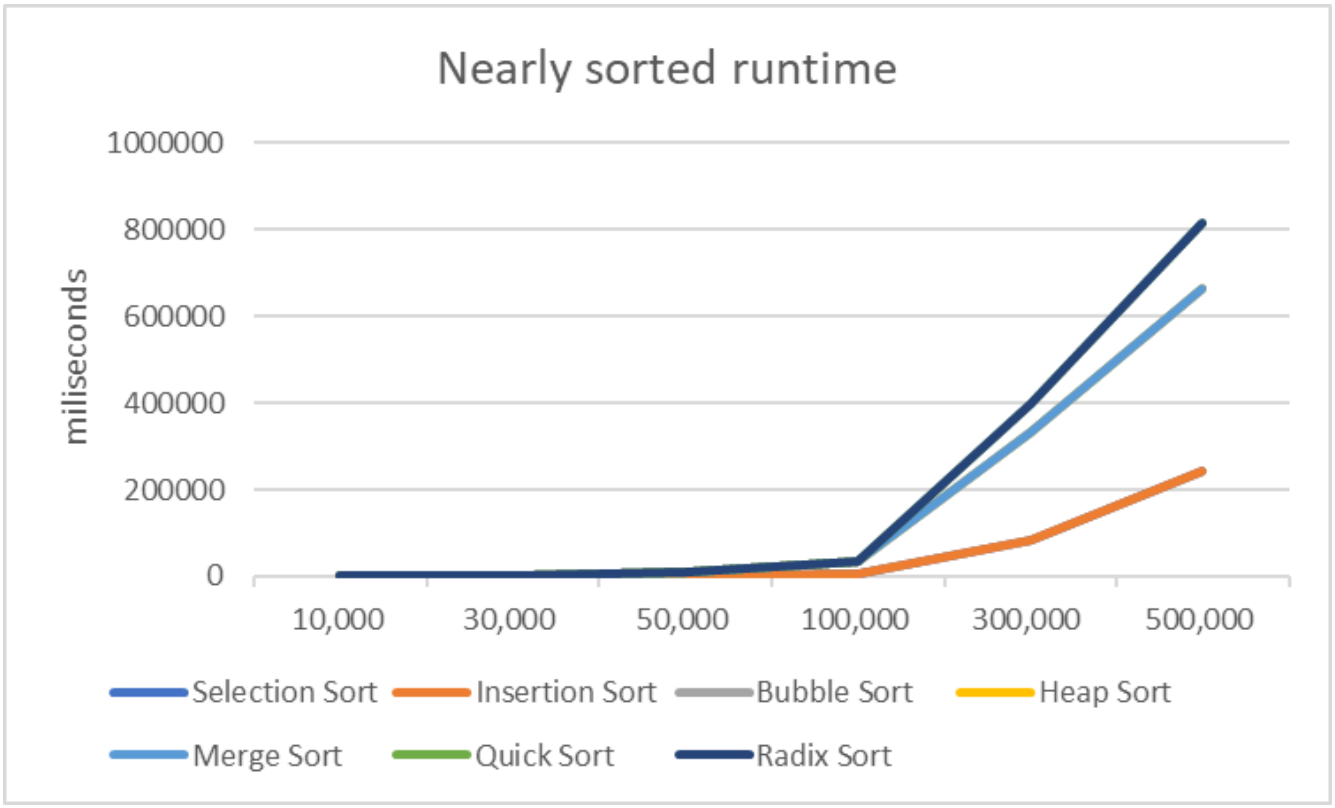


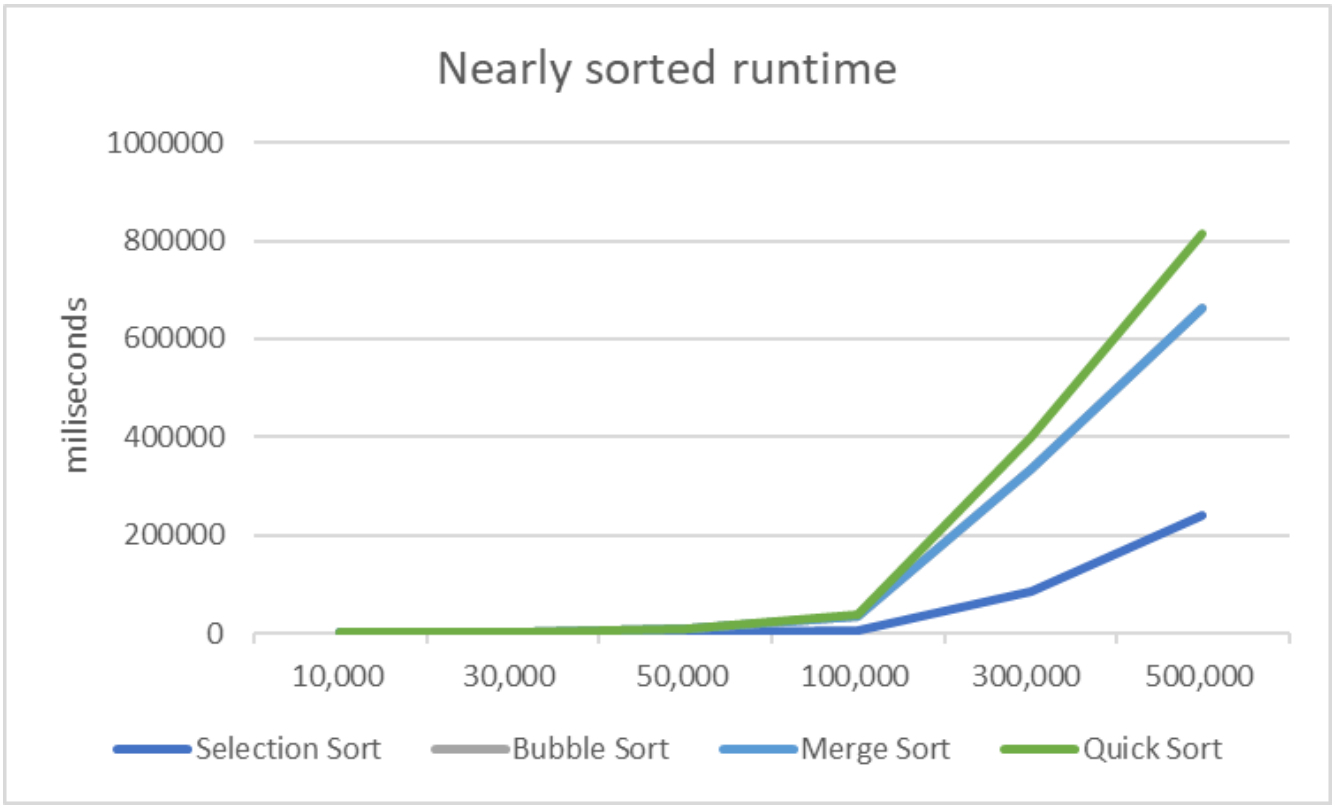
Nhận xét:

- Thuật toán có ít phép so sánh nhất: Insertion Sort
- Thuật toán có nhiều phép so sánh nhất: Quick Sort

2. Nearly sorted data

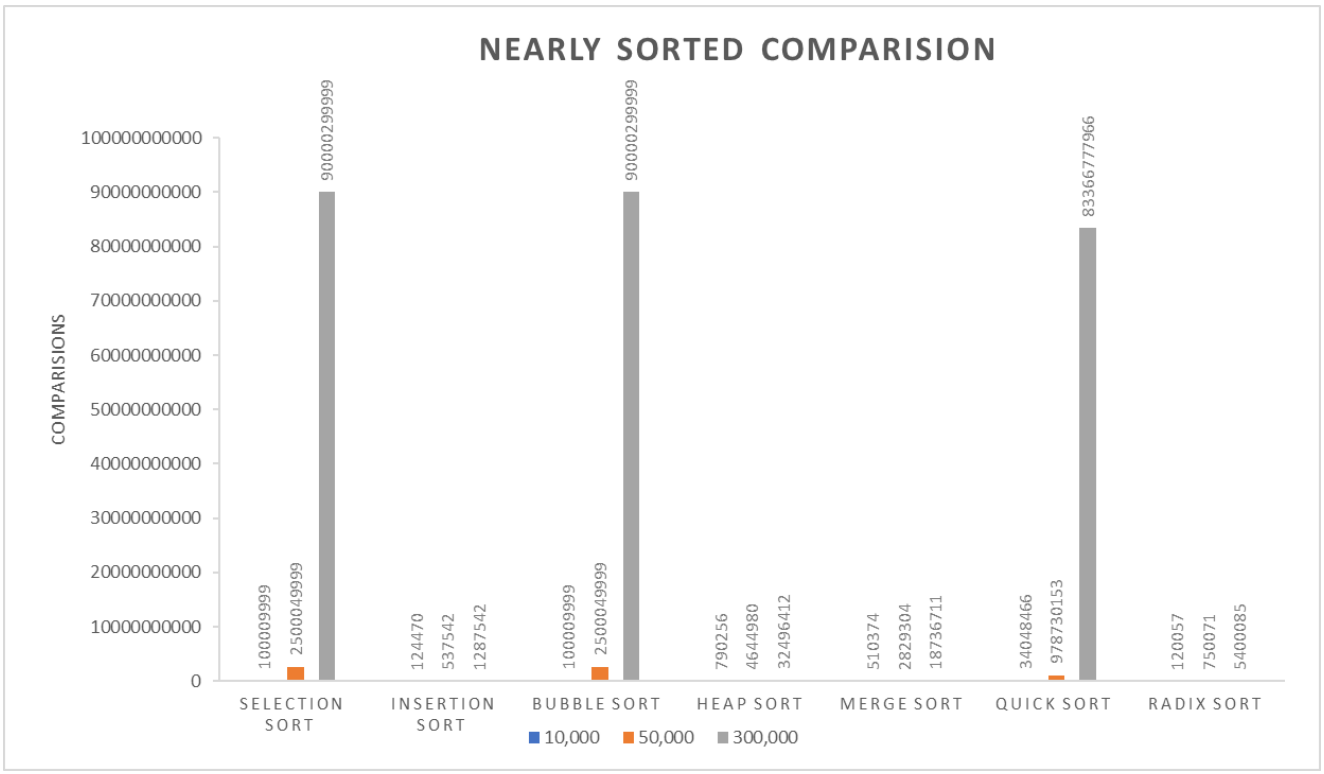
Data order: **Nearly sorted data**												
Data size	10,000		30,000		50,000		100,000		300,000		500,000	
Resulting statics	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision
Selection Sort	56.5339	100009999	504.13	900029999	1400.6	2500049999	5609.34	10000099999	83069.8	90000299999	241270	250000499999
Insertion Sort	0.1365	124470	0.8025	477542	1.3508	537542	0.7343	687542	1.4826	1287542	1.8856	1887542
Bubble Sort	368.952	100009999	2176.44	900029999	6616.57	2500049999	27785	10000099999	250692	90000299999	421887	250000499999
Heap Sort	14.3321	790256	42.9817	2643490	83.1475	4644980	209.878	9908436	625.282	32496412	1123.97	56228886
Merge Sort	5.3027	510374	25.4603	1669172	33.9491	2829304	55.1344	5852136	172.351	18736711	296.321	32133798
Quick Sort	15.8536	34048466	107.307	331318794	486.514	978730153	2711.05	6207612129	65496.5	83366777966	152196	227854751415
Radix Sort	1.2654	120057	4.7559	450071	8.0213	750071	16.372	1500071	58.248	5400085	97.803	9000085

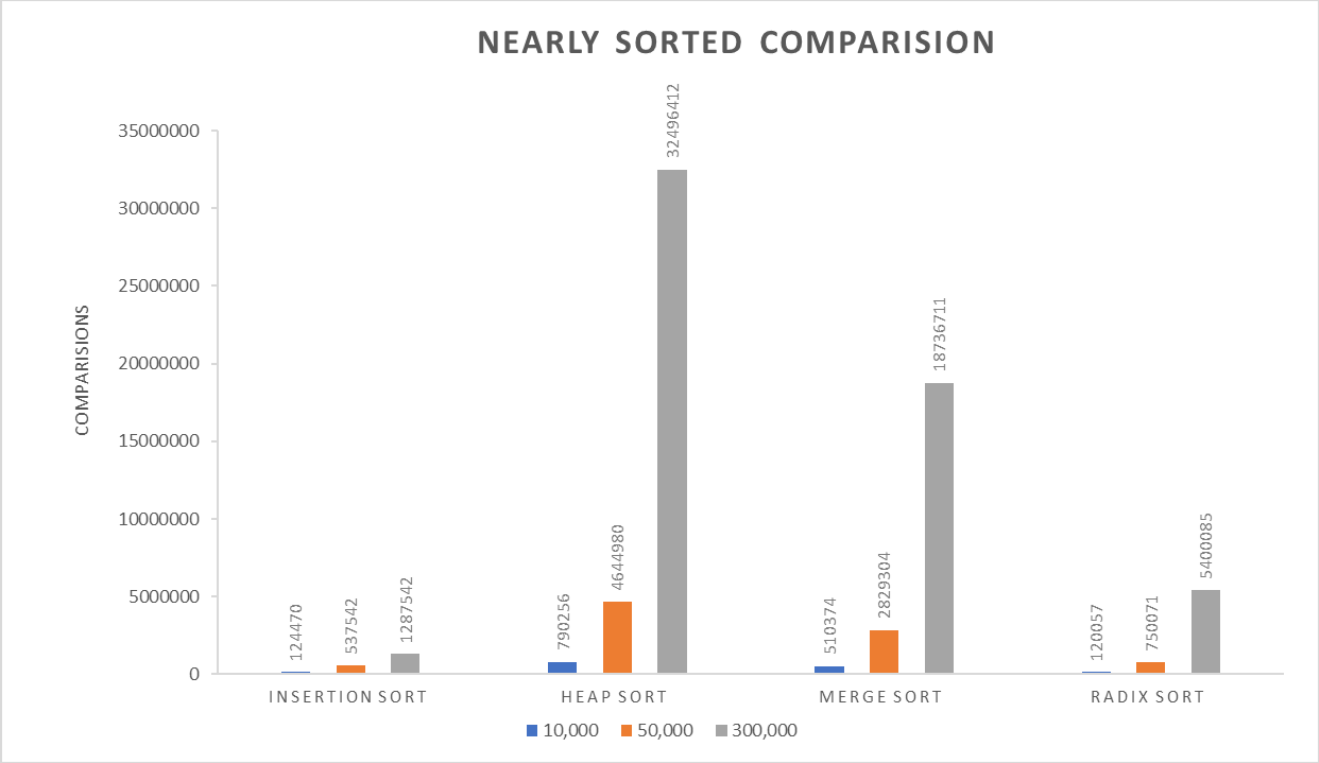




Nhận xét:

- Thuật toán có thời gian chạy ngắn nhất: Selection Sort
- Thuật toán có thời gian chạy dài nhất: Radix Sort



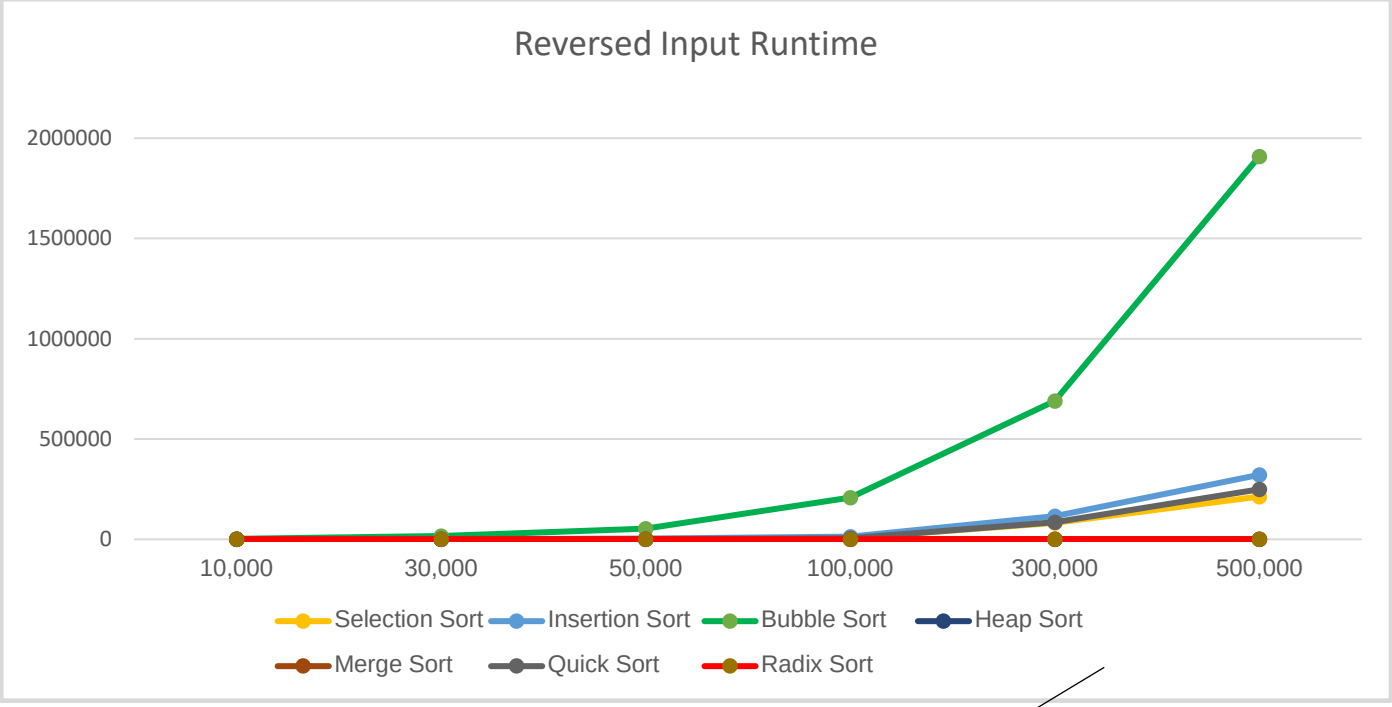


Nhận xét:

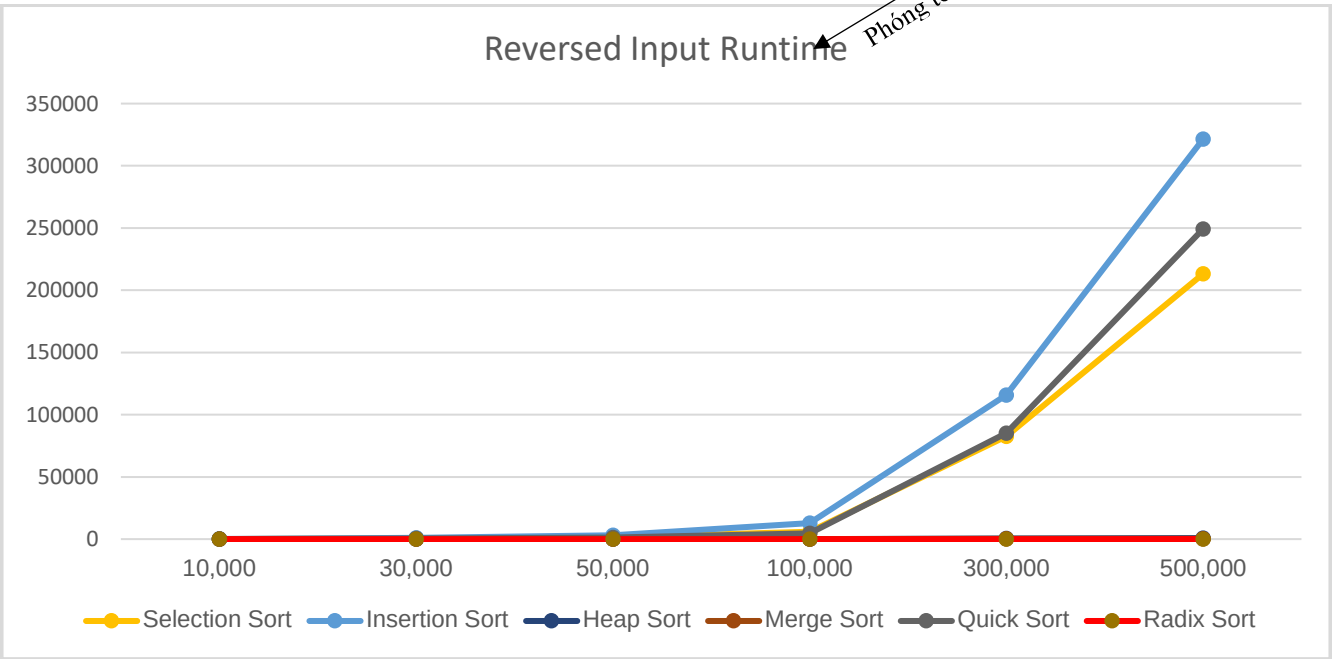
- Thuật toán có ít phép so sánh nhất: Insertion Sort
- Thuật toán có nhiều phép so sánh nhất: Bubble Sort và Selection Sort

3. Reverse sorted data

Data order: **Reverse sorted data**												
Data size	10,000		30,000		50,000		100,000		300,000		500,000	
Resulting statics	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision
Selection Sort	59.4575	100009999	532.972	900029999	1482.95	2500049999	5928.23	10000099999	82428.87	90000299999	213154	250000499999
Insertion Sort	133.302	100009999	1158.23	900029999	3194.32	2500049999	12913	10000099999	115829	90000299999	321437	250000499999
Bubble Sort	2394.39	100009999	16815.1	900029999	52699	2500049999	207647	10000099999	689489	90000299999	1909350	250000499999
Heap Sort	12.8499	712171	47.7736	2429235	83.1292	4254862	127.538	9105513	513.614	30207673	846.521	52593440
Merge Sort	5.937	476441	18.1375	1573465	27.5476	2733945	55.0035	5767897	168.361	18708313	289.259	32336409
Quick Sort	48.2588	100034997	425.214	900104997	1176.11	2500174997	4693.66	10000349997	85046.1	90001049997	249192	250001749997
Radix Sort	1.3132	120057	4.9015	450071	7.8788	750071	16.4312	1500071	58.8987	5400085	99.0274	9000085

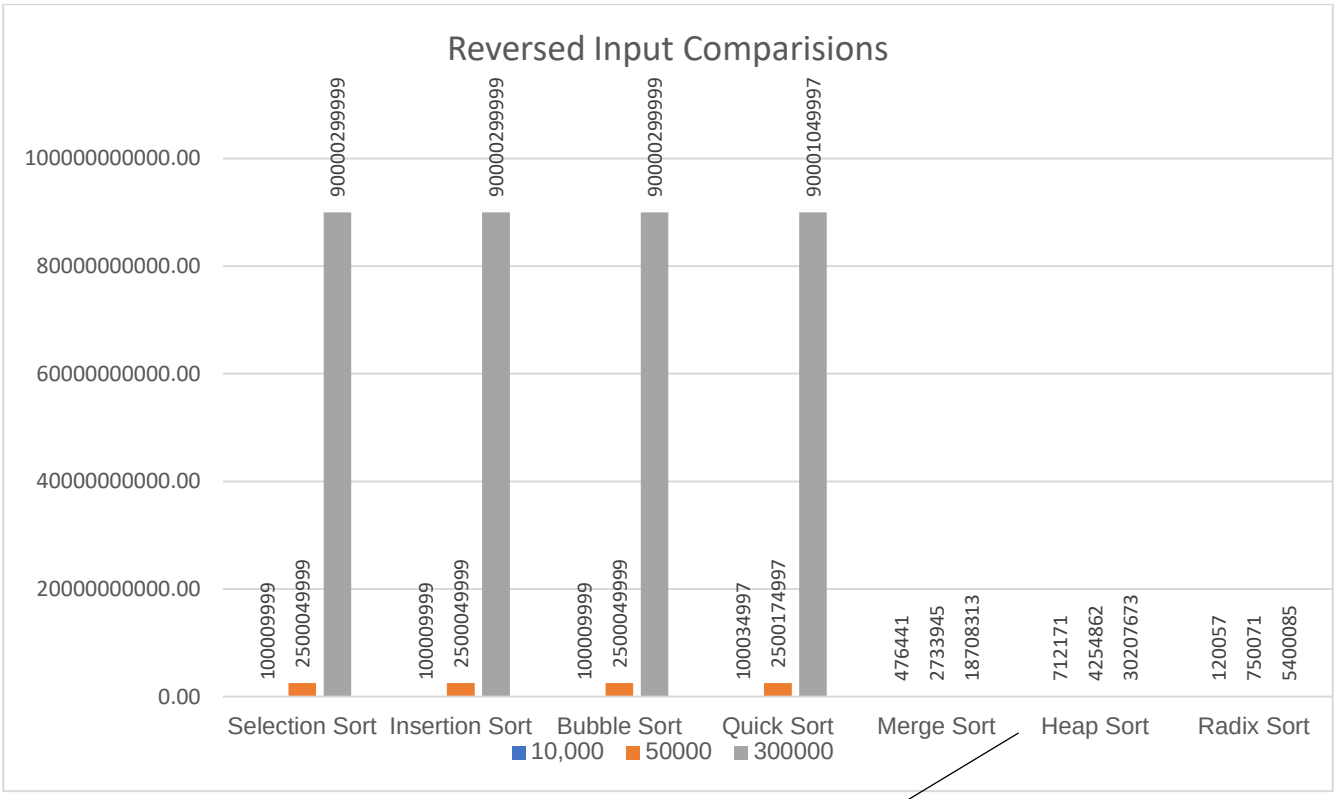


Phóng to

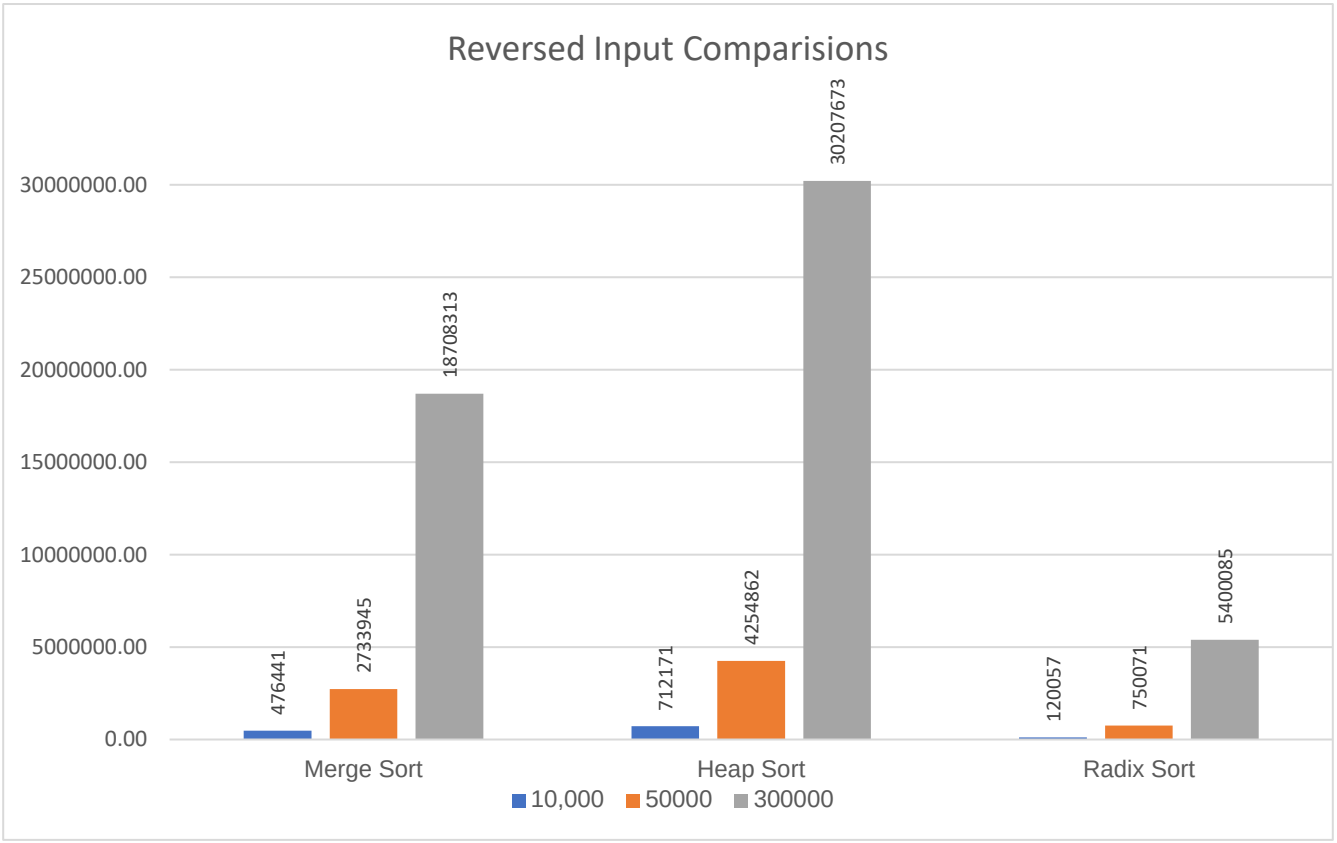


Nhận xét:

- Thuật toán có thời gian chạy ngắn nhất: Radix Sort.
- Thuật toán có thời gian chạy dài nhất: Bubble Sort.


Reversed Input Comparisions
100000000000.00
80000000000.00
60000000000.00
40000000000.00
20000000000.00
0.00
100009999
2500049999
90000299999
100009999
2500049999
90000299999
100009999
2500049999
90000299999
100034997
2500174997
90001049997
476441
2733945
18708313
712171
4254862
30207673
120057
750071
5400085
Selection Sort
Insertion Sort
Bubble Sort
Quick Sort
Merge Sort
Heap Sort
Radix Sort
10,000
50000
300000

Phóng to

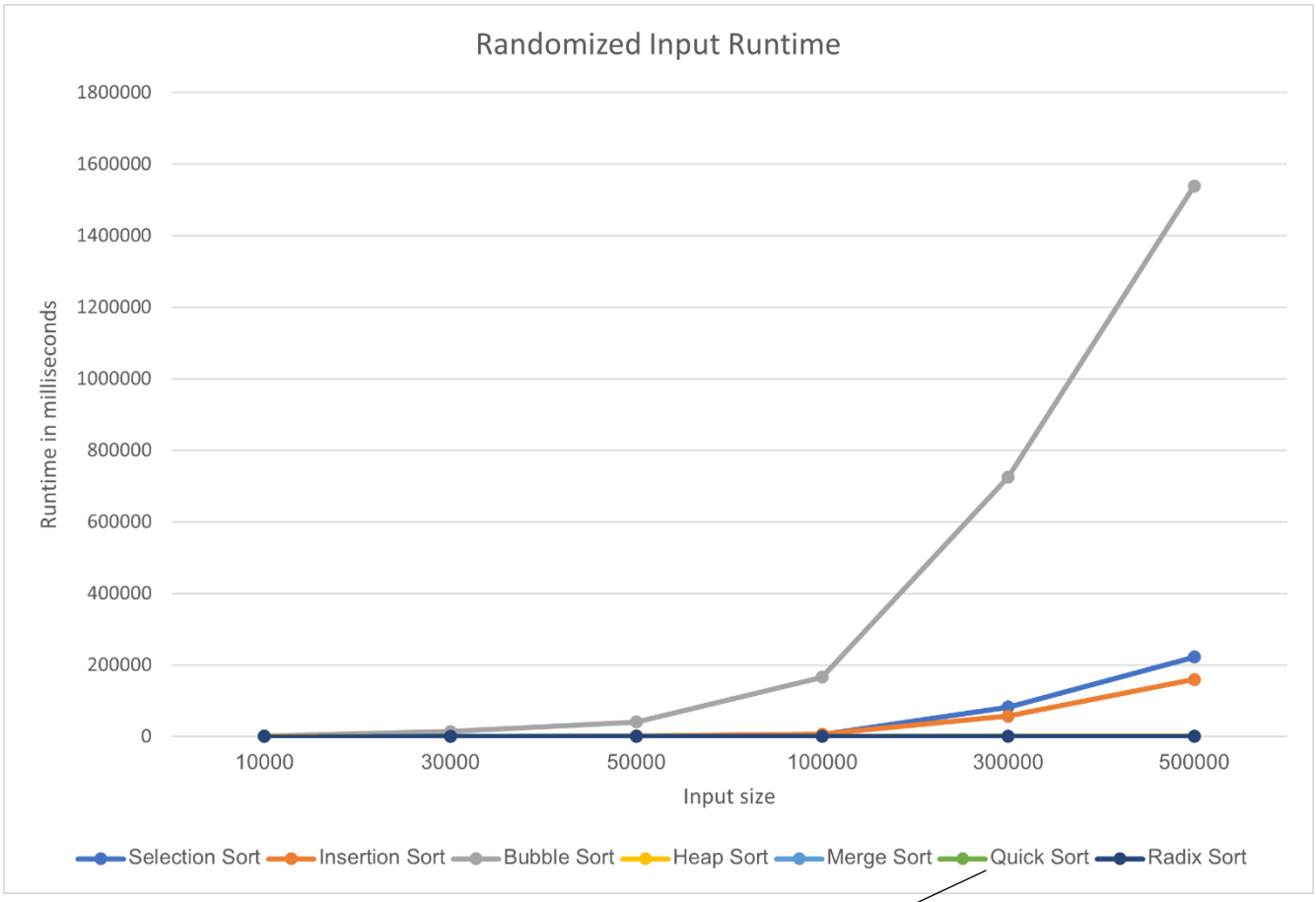

Reversed Input Comparisions
30000000.00
25000000.00
20000000.00
15000000.00
10000000.00
5000000.00
0.00
476441
2733945
18708313
712171
4254862
30207673
120057
750071
5400085
Merge Sort
Heap Sort
Radix Sort
10,000
50000
300000

Nhận xét:

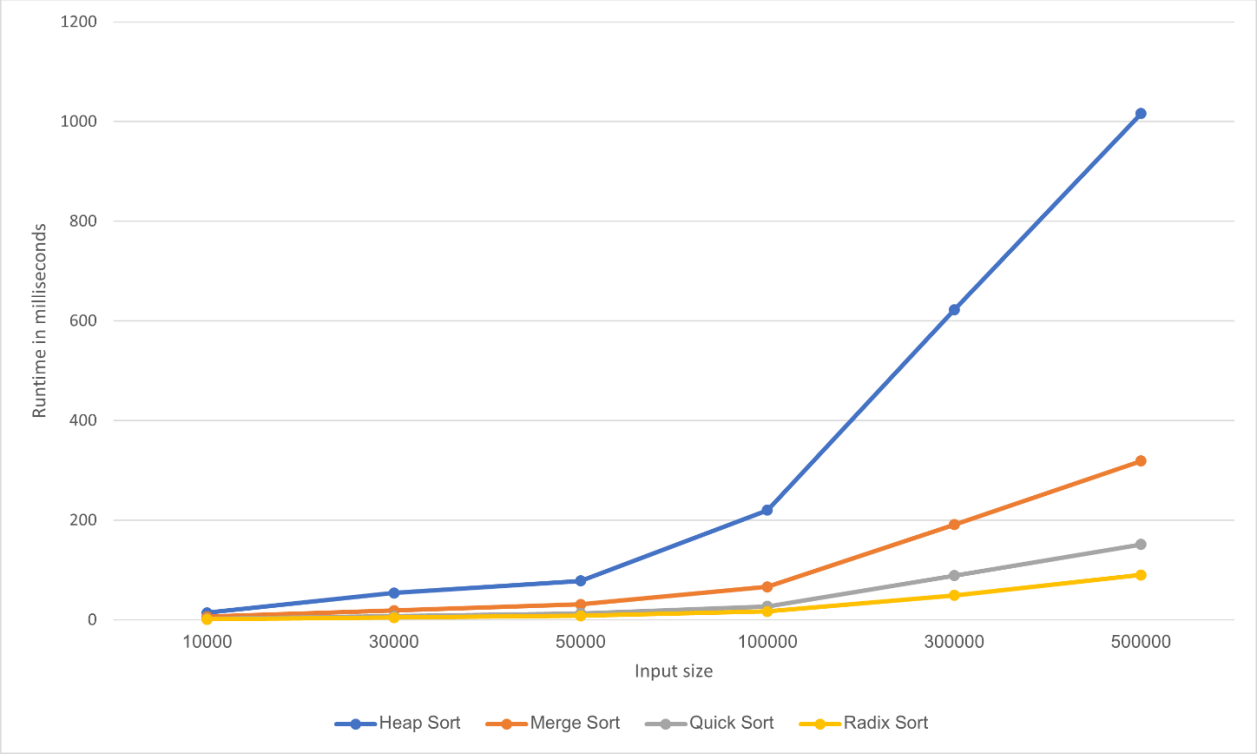
- Thuật toán có ít phép so sánh nhất: Radix Sort.
- Thuật toán có nhiều phép so sánh nhất: Selection Sort, Insertion Sort và Bubble Sort.

4. Randomized data

Data order: **Randomized data**												
Data size	10,000		30,000		50,000		100,000		300,000		500,000	
Resulting statics	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision	Running time	Comparision
Selection Sort	56.8152	100009999	506.054	900029999	1407.12	2500049999	5609.42	10000099999	82646.05	90000299999	221901	250000499999
Insertion Sort	67.488	50004153	589.326	449586208	1599.45	125133286	6395.3	500119321	57456.1	45006647931	159684	124867078878
Bubble Sort	1814.6	100009999	14350.2	900029999	40776.4	2500049999	165777	10000099999	725401	90000299999	1538770	250000499999
Heap Sort	14.305	748752	53.8222	2531266	78.2371	4443862	219.979	9488744	621.942	31290544	1015.86	54338003
Merge Sort	6.5046	583700	18.8503	1937202	31.193	3383264	66.4679	7166146	190.862	23383745	318.765	40383378
Quick Sort	2.2219	592705	7.3898	1949004	12.9062	3695186	27.1773	7954112	88.7711	25890722	151.585	48111808
Radix Sort	1.2575	120057	4.8072	450071	8.3525	750071	16.8526	1500071	49.3299	4500071	90.3043	7500071

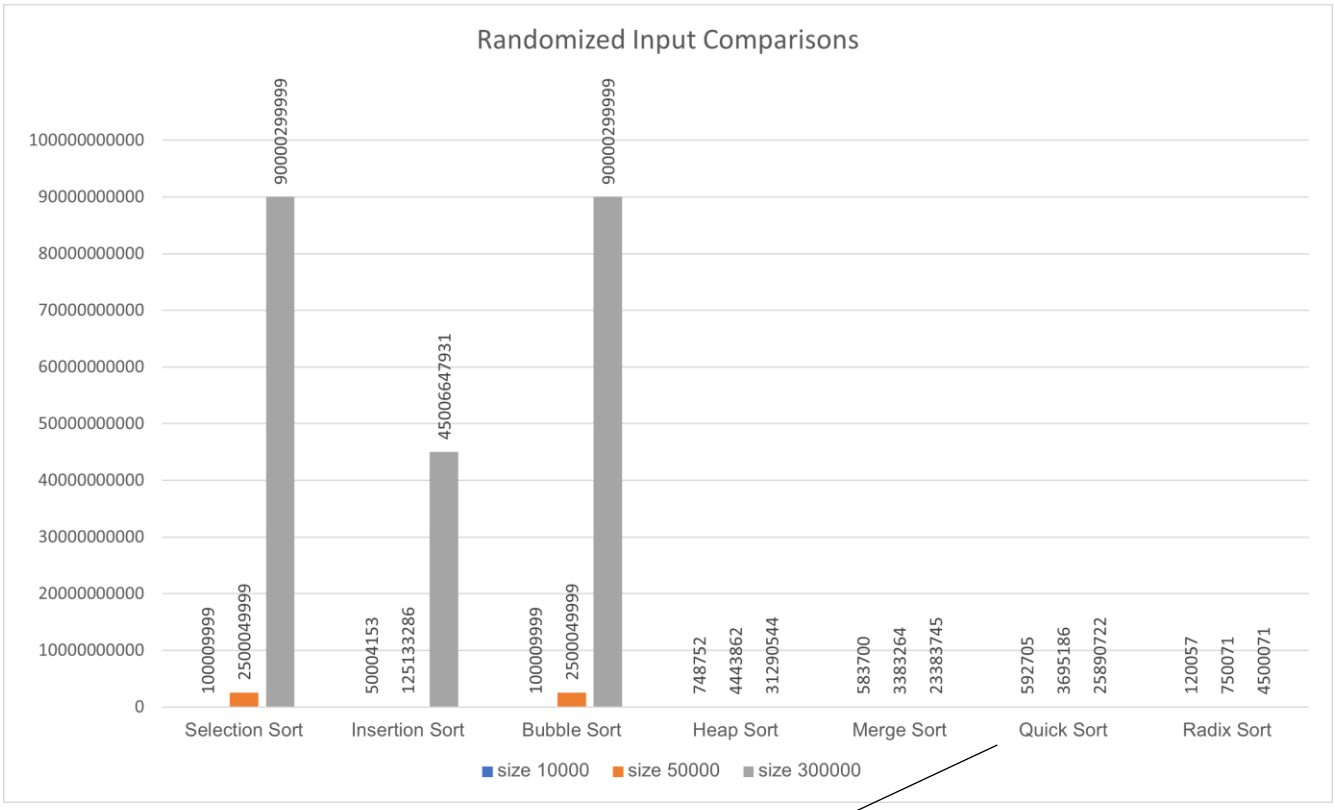


Phóng to

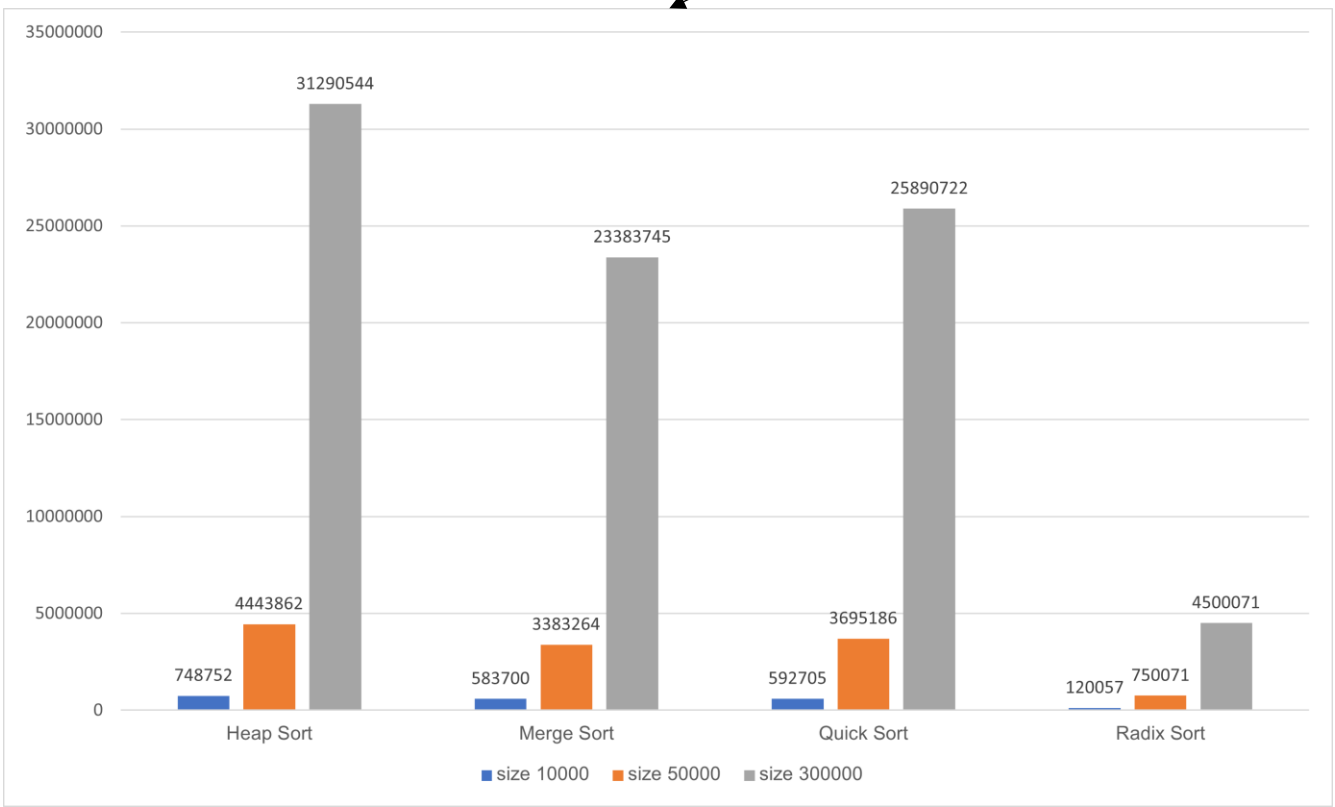


Nhận xét:

- Thuật toán có thời gian chạy ngắn nhất: Radix Sort.
- Thuật toán có thời gian chạy dài nhất: Bubble Sort. Do Bubble Sort sử dụng 2 vòng for lồng nhau, độ phức tạp thời gian của thuật toán này đã phải tốn chi phí đến $O(n^2)$, chưa kể phải liên tục thực hiện hoán vị trong quá trình duyệt mảng dẫn đến thời gian thực hiện thuật toán Bubble Sort sẽ lâu hơn đáng kể so với các thuật toán còn lại.


Randomized Input Comparisons
100000000000
90000000000
80000000000
70000000000
60000000000
50000000000
40000000000
30000000000
20000000000
10000000000
0
100009999
2500049999
90000299999
50004153
125133286
45006647931
100009999
2500049999
90000299999
748752
4443862
31290544
583700
3383264
23383745
592705
3695186
25890722
120057
750071
4500071
Selection Sort
Insertion Sort
Bubble Sort
Heap Sort
Merge Sort
Quick Sort
Radix Sort
size 10000
size 50000
size 300000

Phóng to


35000000
30000000
25000000
20000000
15000000
10000000
5000000
0
748752
4443862
31290544
583700
3383264
23383745
592705
3695186
25890722
120057
750071
4500071
Heap Sort
Merge Sort
Quick Sort
Radix Sort
size 10000
size 50000
size 300000

Nhận xét:

- Thuật toán có ít phép so sánh nhất: Radix Sort
- Thuật toán có nhiều phép so sánh nhất: Bubble Sort và Selection Sort. Hai thuật toán này có ý tưởng gần giống nhau là dùng 2 vòng for lồng nhau để duyệt qua và so sánh các phần tử trong mảng. Do đó, số phép so sánh của hai thuật toán này cũng giống nhau.

5. Nhận xét tổng thể:

Thông qua phần thí nghiệm trên với các dữ liệu có kích thước và thứ tự khác nhau, có thể rút ra kết luận như sau:

- Thuật toán có thời gian chạy ngắn nhất:
  + Đối với các kiểu dữ liệu đầu vào đã được sắp xếp sẵn (sorted data) tăng dần hoặc gần như được sắp xếp sẵn (nearly sorted data) tăng dần, thuật toán có thời gian chạy ngắn nhất là Insertion Sort
  + Đối với các kiểu dữ liệu đầu vào ngẫu nhiên (randomized data) hoặc được sắp xếp giảm dần (reverse sorted data), thuật toán có thời gian chạy ngắn nhất là Radix Sort
- Thuật toán có thời gian chạy dài nhất: Bubble Sort
- Các thuật toán sắp xếp ổn định: Bubble Sort, Insertion Sort, Merge Sort, Radix Sort
- Các thuật toán sắp xếp không ổn định : Selection Sort, Quick Sort, Heap Sort

IV. Tổ chức project, ghi chú lập trình

1. Tổ chức project

Project gồm 3 file:

- File func.cpp: chứa các hàm (hàm khởi tạo dữ liệu, hàm tính thời gian chạy của các thuật toán,…)
- File func.h: file header của một số hàm chính trong func.cpp
- File main.cpp: file chứa hàm main

2. Ghi chú lập trình

- Ở Quick Sort, do mỗi lần gọi đệ quy để chia mảng sẽ tốn thêm bộ nhớ để tạo stack nên kích thước dành cho stack mặc định của Visual Studio là 1MB sẽ không đủ cho Quick Sort thực hiện sắp xếp trên các mảng có kích thước lớn như bài tập đã yêu cầu (mảng có kích thước $n \geq 10000$), dẫn đến tràn stack. Do đó, cần tăng kích thước dành cho stack bằng cách setting lại Visual Studio: Project -> Properties ->

Configuation Properties -> Linker -> System -> Thay đổi Stack Reserve Size thành 300000000 (~300MB)

- Ở hàm tính Comparisions và Running Time của Quick sort, có 2 phần tử được truyền vào là left và right, đó chính là vị trí của phần tử đầu tiên và phần tử cuối cùng của mảng [0] và [N-1] nên khi gọi hai hàm trên ở trong hàm main (), ta sẽ truyền vào tương ứng left = 0 và right = N-1; trường hợp tương tự với hàm tính Comparisions của Merge Sort (truyền begin = 0 và end = N-1).
- Trong các hàm command (1 → 5) sẽ có trường hợp yêu cầu xuất ra cả 2 phần là Running Time và Comparisions của thuật toán, nếu 2 hàm được gọi kế tiếp nhau và đều truyền vào cùng một mảng (chung arr[]) thì sẽ xảy ra trường hợp hàm được gọi phía sau sẽ ra kết quả lỗi (vì ở hàm đầu đã sort mảng thành tăng dần rồi nên mảng được truyền vào ở hàm thứ 2 không mang cùng bộ dữ liệu với mảng ban đầu). Vậy nên trước khi gọi 2 hàm, ta sẽ tạo ra một mảng mới có bộ dữ liệu giống hệt mảng cũ, sau đó truyền 2 mảng vào hai hàm riêng biệt, để đảm bảo cả hai hàm đều chạy chung 1 bộ dữ liệu. Áp dụng điều này cho việc chạy chung 1 bộ dữ liệu cho nhiều thuật toán sắp xếp.

V. Tài liệu tham khảo

- Geeksforgeeks
- JavaTpoint
- Wikipedia