BÀI BÁO CÁO ĐỀ TÀI SORT ALGORITHM

Tên: Bùi Hồng Dương

MSSV:20120273

I. TRÌNH BÀY VỀ THUẬT TOÁN

1/ Selection Sort:

Ý tưởng: Đây là thuật toán sắp xếp bằng cách tìm phần tử nhỏ nhất của mảng sau đó hoán đổi với phần tử đầu tiên của mảng, tiếp đó tìm phần tử nhỏ thứ hai và hoán đổi với phần tử thứ hai của mảng, tìm phần tử nhỏ thứ ba rồi hoán đổi với phần tử thứ ba của mảng,…,cứ thế tiếp tục cho đến phần tử kế cuối, khi đó ta sẽ được 1 mảng đã được sắp xếp.

Thuật toán:

Bước 1: Cho chạy vòng lặp từ phần tử đầu đến phần tử kế cuối (i=0->n-2).

Bước 2: Tạo biến minI là chỉ số của phần tử nhỏ nhất trong đoạn từ a[i] đến a[n-1].

Bước 3: Cho chạy vòng lặp từ phần tử i+1 đến phần tử cuối cùng của mảng.

Bước 4: Kiểm tra trong quá trình chạy vòng lặp thứ 2, nếu có phần tử nào nhỏ hơn a[minI] thì gắn minI bằng chỉ số của phần tử đó.

Bước 5: Sau khi chạy xong vòng lặp thứ 2 thì hoán vị a[i] với a[minI] (a[minI là phần tử nhỏ nhất trong đoạn mảng chưa sắp xếp, còn a[i] là phần tử đứng đầu đoạn mảng chưa sắp xếp)

Bước 6: i=i+1.Kiểm tra xem i có còn phù hợp với điều kiện của vòng lặp thứ nhất hay không, nếu có quay lại bước thứ 2, nếu không thì kết thúc.

Đánh giá về thuật toán:

+Độ phức tạp về thời gian: Vì có 2 vòng lặp nên độ phức tạp về thời gian là O(n^2)

- Trường hợp xấu nhất (Mảng sắp xếp giảm dần): O(n^2)
- Trường hợp tốt nhất (Mảng sắp xếp tăng dần): O(n^2)
- Trường hợp trung bình: O(n^2)

+Độ phức tạp về không gian: O(1)

2/ Insertion Sort:

Ý tưởng: Chúng ta sẽ sắp xếp bằng cách duyệt từng phần tử và so sánh nó với phần tử kề trước nó, nếu bé hơn thì đổi chỗ và tiếp tục lặp lại cho đến khi phần tử đó lớn hơn phần tử kề trước nó. Mục đích của việc này là để chèn phần tử đó vào đúng vị trí trong phần mảng đã được sắp xếp. Sau khi duyệt đến phần tử cuối thì ta sẽ có một mảng đã được sắp xếp hoàn chỉnh.

Thuật toán:

Bước 1: Coi phần tử đầu tiên thuộc đoạn mảng đã được sắp xếp

Bước 2: Cho chạy vòng lặp từ 1 đến n-1 (i=1->n-1)

Bước 3: Khi duyệt đến phần tử a[i], ta gắn biến key = a[i] để lưu giá trị của a[i], gắn j = i để lưu vị trí của phần tử

Bước 4: Ta so sánh key với phần tử kề trước là a[j-1].Nếu a[j-1] lớn hơn thì ta đẩy a[j-1] lên một vị trí và giảm j xuống 1 đơn vị (j=j-1).

Bước 5: Lặp lại bước 4 cho đến khi a[j-1] <= a[j] hoặc j = 0

Bước 6: Chèn giá trị của key vào vị trí của a[j] (đã đặt key vào đúng vị trí trong đoạn mảng sắp xếp, lúc này đã có i+1 phần tử được sắp xếp)

Bước 7: i=i+1. Kiểm tra xem i có còn phù hợp với điều kiện của vòng lặp thứ nhất hay không, nếu có quay lại bước thứ 3, nếu không thì kết thúc.

Đánh giá về thuật toán:

+Độ phức tạp về thời gian:

- Trường hợp xấu nhất (Mảng sắp xếp giảm dần): Với trường hợp này, mỗi phần tử ở vị trí thứ n sẽ có n-1 phép so sánh (vì phần tử thứ n sẽ là phần tử bé nhất trong đoạn mảng đã được sắp xếp, nên nó được đưa lên đầu. Để đưa lên đầu cần tốn n-1 phép so sánh). Như vậy tổng số so sánh là n*(n-1) ~ n^2
  ->O(n^2)
- Trường hợp tốt nhất (Mảng sắp xếp tăng dần): Trường hợp này chỉ có vòng lặp bên ngoài chạy, vòng lặp bên trong không hề chạy (vì mảng đã được sắp xếp nên mọi vị trí đều lớn hơn hoặc bằng phần tử kề trước nó nên sẽ không chạy vòng lặp ở bên trong). Như vậy độ phức tập là O(n) (vì chỉ chạy 1 vòng lặp)
- Trường hợp trung bình: O(n^2)

+Độ phức tạp về không gian: O(1)

3/ Bubble Sort

Ý tưởng: Là thuật toán sắp xếp bằng cách duyệt qua các phần tử và hoán đổi 2 số liền kề nhau nếu số sau bé hơn số trước. Sau mỗi vòng lặp thì phần tử lớn nhất của phần chưa được sắp xếp sẽ về cuối của đoạn mảng chưa được sắp xếp. Khi duyệt đến phần tử cuối, ta sẽ được một mảng đã được sắp xếp.

Thuật toán:

Bước 1: Cho chạy vòng lặp từ phần tử đầu tiên đến phần tử kế cuối (i = 0 -> n-2)

Bước 2: Tạo biến flag = 0 để kiểm tra xem có sự hoán đổi ở vòng lặp thứ 2 hay không

Bước 3: Cho chạy vòng lặp thứ 2 (bên trong vòng lặp thứ nhất) từ 0 cho đến n-1-i (đến phần tử cuối của đoạn mảng chưa được sắp xếp, j = 0 -> n-1-i)

Bước 4: Kiểm tra xem phần tử a[j] có lớn hơn a[j+1] hay không, nếu có và hoán đổi vị trí 2 phần tử này đồng thời gắn flag là 1 (báo hiệu đã có sự hoán đổi), tiếp tục như thế cho đến hết vòng lặp thứ 2.

Bước 5: Kiểm tra xem flag có bằng 0 hay không, nếu có thì kết thúc thuật toán, nếu không thì tiếp tục chạy vòng lặp thứ 1.

Đánh giá thuật toán:

+Độ phức tạp về thời gian:

- Trường hợp xấu nhất (Mảng sắp xếp giảm dần): Trường hợp này, vòng lặp thứ 1 sẽ có n-1 phép so sánh, vòng lặp thứ 2 có n-2 phép so sánh,…,vòng lặp thứ n-1 sẽ có 1 phép so sánh.
  Tổng số phép so sánh là (n-1)+(n-2)+(n-3)+…+1=n(n-1)/2 ~ n^2 -> Độ phức tạp là O(n^2).
- Trường hợp tốt nhất (Mảng sắp xếp tăng dần): Trường hợp này, ta thấy chỉ chạy được vòng lặp thứ 2 được 1 lần (vì không có sự hoán đổi nên flag sẽ bằng 0) thì thuật toán kết thúc nên sẽ chỉ có n-1 ~ n phép so sánh.
  ->Độ thức tạp là O(n).
- Trường hợp trung bình: O(n^2).

+Độ phức tạp về không gian: O(1)

4/ Merge Sort

Ý tưởng: Dựa vào kĩ thuật chia để trị trong đệ quy. Ta chia mảng ra làm 2 nửa, bên trái và bên phải. Gọi đệ quy thuật toán này để sắp xếp nửa bên trái và nữa bên phải, sau đó ta trộn 2 mảng lại thành một mảng đã được sắp xếp. Nếu mảng chỉ có 1 phần tử thì coi như mảng đã được sắp xếp.

Thuật toán:

Bước 1: Ta tạo ra hàm Merge, mục đích là để trộn 2 phần mảng đã được sắp xếp thành 1 mảng được sắp xếp

Bước 1.1: Trong hàm Merge tạo ra 2 mảng mới, mảng aL là bảng copy của mảng bên trái đã được sắp xếp, mảng aR là bản copy của mảng bên phải đã được sắp xếp.

Bước 1.2: Tạo ra 3 chỉ số: iR là chỉ số của mảng aR ,iL là chỉ số của mảng aL, iA là chỉ số của mảng gộp. Lúc đầu cả 3 chỉ số này đều bằng 0

Bước 1.3: Cho đến khi nào 1 trong 2 chỉ số iR hoặc iL vượt quá số lượng phần tử trong mảng của chúng, ta sẽ so sánh 2 phần tử aR[iR] và aL[iL], nếu phần tử nào bé hơn thì ta sẽ sắp xếp phần tử đó vào vị trí a[iA] trong mảng đồng thời tăng chỉ số của mảng đó và mảng gộp lên (vd: aR[iR] > aL[iL] thì đặt a[iA] = al[iL] và tăng iL và iA lên 1)

Bước 1.4: Nếu 1 trong 2 mảng đã đưa đủ phần tử vào mảng gộp, ta đưa hết những phần tử còn lại của mảng chưa đưa đủ phần tử vào trong mảng gộp.

Bước 2: Ta tạo hàm MergeSort để chia mảng ra thành 2 phần cũng như sắp xếp mảng

Bước 2.1: Vì hàm MergeSort là hàm đệ quy nên ta sẽ cần điều kiện dừng. Ở đây điều kiện là khi chỉ số đầu bằng hoặc vượt quá chỉ số cuối thì ta sẽ dừng hàm MergeSort

Bước 2.2: Ta chia mảng ra làm 2 phần. Phần bên trái từ chỉ số đầu đến chỉ số mid (mid=đầu+cuối/2), phần bên phải bằng chỉ số mid+1 đến chỉ số cuối.

Bước 2.3: Gọi đệ quy hàm MergeSort để sắp xếp mảng bên trái và mảng bên phải, (tức là quay lại bước 2.1)

Bước 2.4: Gọi hàm Merge để gộp 2 mảng này lại với nhau thành một mảng sắp xếp hoàn chỉnh.

Đánh giá thuật toán:

+Độ phức tạp về thời gian:

- Trường hợp xấu nhất (Mảng sắp xếp giảm dần): O(n*logn)
- Trường hợp tốt nhất (Mảng sắp xếp tăng dần): O(n*logn)
- Trường hợp trung bình: O(n*logn)

+Độ phức tạp về không quan: O(n)

5/Quick Sort

Ý tưởng: Dựa vào kĩ thuật chia để trị trong đệ quy. Ta chọn một phần tử làm pivot, sau đó chia mảng thành các phần: phần bên trái pivot là các phần tử nhỏ hơn pivot, phần bên phải pivot là các phần tử lớn hơn pivot, còn phần tử pivot lúc này đã ở đúng vị trí cần sắp xếp. Ta tiếp tục gọi đệ quy để sắp xếp phần bên trái pivot và phần bên phải pivot, lúc này ta được một mảng đã được sắp xếp tăng dần.

Thuật toán:

Bước 1: Ta đi xây dựng hàm Partition để chia mảng cũng như đưa pivot về đúng vị trí. Hàm này sẽ trả về chỉ số của pivot

Bước 1.1: Ta sẽ chọn pivot là phần tử ở giữa mảng (mid =đầu+cuối/2).

Bước 1.2: Ta sẽ tạo ra chỉ số j = chỉ số đầu. Mục đích là để đưa các phần tử nhỏ hơn pivot về phía bên trái pivot và sau khi chạy xong vòng lặp thì đây chính là chỉ số của pivot

Bước 1.3: Ta cho chạy vòng lặp từ chỉ số đầu đến chỉ số mid-1 rồi từ mid+1 đến chỉ số cuối.

Bước 1.4: Trong vòng lặp, ta so sánh giá trị của phần tử a[i] với phần tử pivot, nếu nó nhỏ hơn thì ta sẽ hoán đổi vị trí của a[i] với a[j] (đưa a[i] về phía bên trái pivot) và tăng j lên 1 đơn vị, tiếp tục so sánh cho đến hết vòng lặp.

Bước 1.5: Sau khi xong vòng lặp, chỉ số j chính là vị trí của pivot ở trong mảng sắp xếp nên ta hoán đổi vị trí a[j] với a[mid] (pivot), sau đó ta trả về j (vị trí pivot đã đứng đúng vị trí).

Bước 2: Ta xây dựng hàm QuickSort để sắp xếp mảng

Bước 2.1: Vì hàm QuickSort là hàm đệ quy nên ta sẽ cần điều kiện dừng. Ở đây điều kiện là khi chỉ số đầu bằng hoặc vượt quá chỉ số cuối thì ta sẽ dừng hàm QuickSort.

Bước 2.2: Ta tạo biến pi (chính là chỉ số của pivot) và gọi hàm Partition để chia mảng và đưa pivot về đúng vị trí.

Bước 2.3: Ta gọi đệ quy hàm QuickSort để sắp xếp phần bên trái và phần bên phải.(Quay lại bước 2.1). Sau khi xong ta sẽ được một mảng đã sắp xếp

Đánh giá thuật toán:

+Độ phức tạp về thời gian:

- Trường hợp xấu nhất: Lúc này mảng đã được sắp xếp nhưng ta chọn pivot là phần tử đầu hoặc phần tử cuối. Hoặc là tất cả các phần tử của mảng đều giống nhau. Điều này sẽ tạo ra một mảng có n-1 phần tử còn mảng kia chỉ có 1 phần tử.Lúc này tiếp tục QuickSort mảng n-1 phần tử sẽ được mảng n-2 phần tử và mảng kia có 1 phần tử. Cứ tiếp tục như thế đến khi chạy hết mảng.
  Lúc này ta sẽ có (n-1) + (n-2) +(n-3) +…1=n*(n-1)/2 ~ n^2 -> Độ phức tạp là O(n^2)
- Trường hợp tốt nhất: Khi pivot là phần tử có giá trị ở giữa mảng. Độ phức tạp O(n*logn)
- Trường hợp trung bình: O(n*logn)

+Độ phức tạp về không gian: O(logn)

6/ Heap Sort

Ý tưởng: là thuật toán sắp xếp dựa trên cấu trúc dữ liệu Binary Heap.Đây là cấu trúc dữ liệu mà node cha sẽ lớn hơn các node con. Ta sẽ dùng max-heap để tìm phần tử lớn nhất (

là node gốc) sau đó hoán đổi với vị trí cuối cùng trong cây và cứ tiếp tục thế đến khi cây max-heap chỉ còn 1 phần tử. Lúc đó mảng đã được sắp xếp.

Thuật toán:

Bước 1: Ta đi xây dựng hàm heapify. Đây là hàm để kiểm tra xem các node cha có lớn hơn node con không, nếu có thì thực hiện hoán đổi và kiểm tra tiếp.

Bước 1.1: Tạo chỉ số max = I (tức node cha). Lúc này ta sẽ có node con trái sẽ bằng 2*i+1, node con phải sẽ bằng 2*i+2. Nhưng nếu 2*i+1 >= n thì không có con trái, 2*i+2 >= n thì không có con phải

Bước 1.2: So sánh giá trị của node cha lần lượt với node con trái và node con phải, nếu node con lớn hơn thì gắn max = chỉ số của node con.

Bước 1.3: Nếu max có sự thay đổi (tức có node con lớn hơn) thì đổi chỗ node cha với node con đó. Sau đó kiểm tra xem các node con mới của node cha cũ có vi phạm không bằng cách gọi lại hàm heapify.

Bước 2: Xây dựng hàm HeapSort để sắp xếp mảng

Bước 2.2: Cho chạy vòng lặp giảm từ i=n/2-1 (node đầu tiên có con) xuống 0 (I = n/2-1 <- 0). Mục đích của việc này là để xây dựng một max-heap gồm n phần tử.

Bước 2.3: Cho chạy vòng lặp từ i = n-1 xuống 0 (i = n-1 <- 0).

Bước 2.4: Trong vòng lặp, ta đổi chỗ phần tử đầu tiên của mảng (phần tử lớn nhất trong max-heap) với phần tử thứ i (là phần tử cuối cùng trong cây). Sau đó loại nó ra khỏi cây max heap.

Bước 2.5: Lúc này cây max-heap chỉ sai một vị trí đầu tiên nên ta có thể xây dựng lại max-heap bằng cách gọi hàm heapify để kiểm tra phần tử đầu tiên của mảng. Giảm i xuống 1 đơn vị và kiểm tra xem có còn đúng với điều kiện lặp không, nếu có quay lại bước 2.4.

Đánh giá thuật toán:

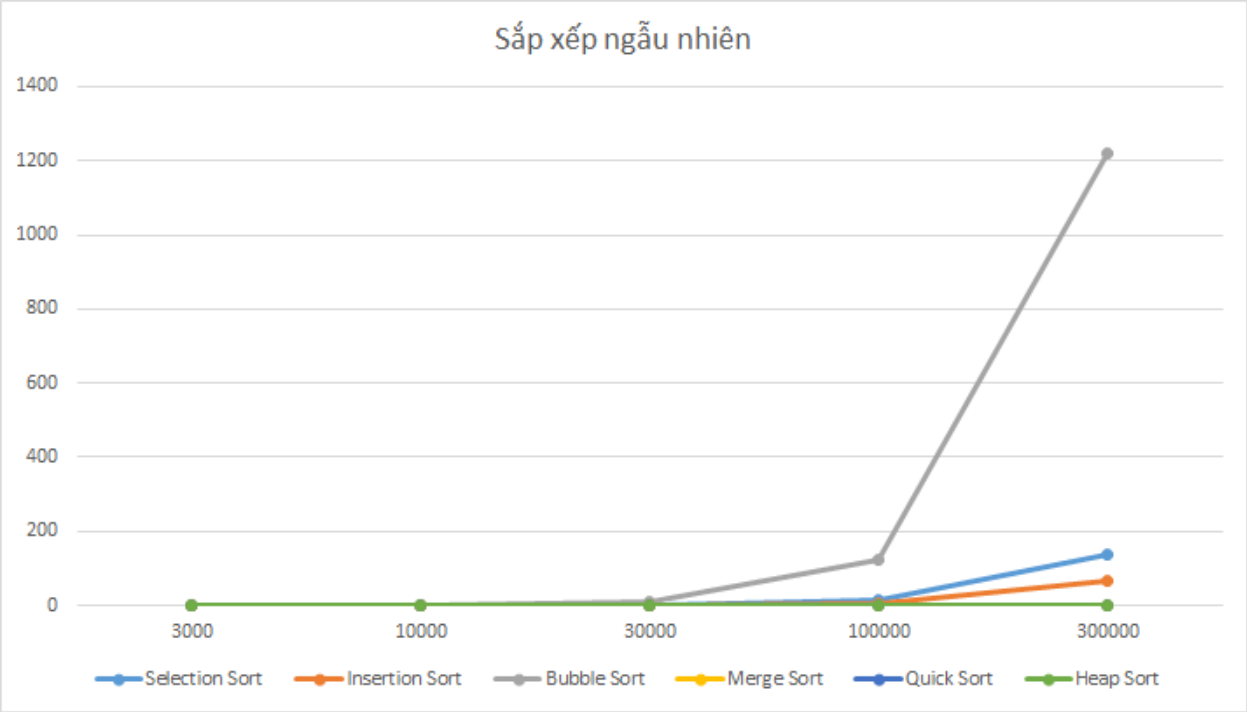
+Độ phức tập về thời gian:

- Trường hợp xấu nhất: O(n*logn)
- Trường hợp tốt nhất: O(n*logn)
- Trường hợp trung bình: O(n*logn)

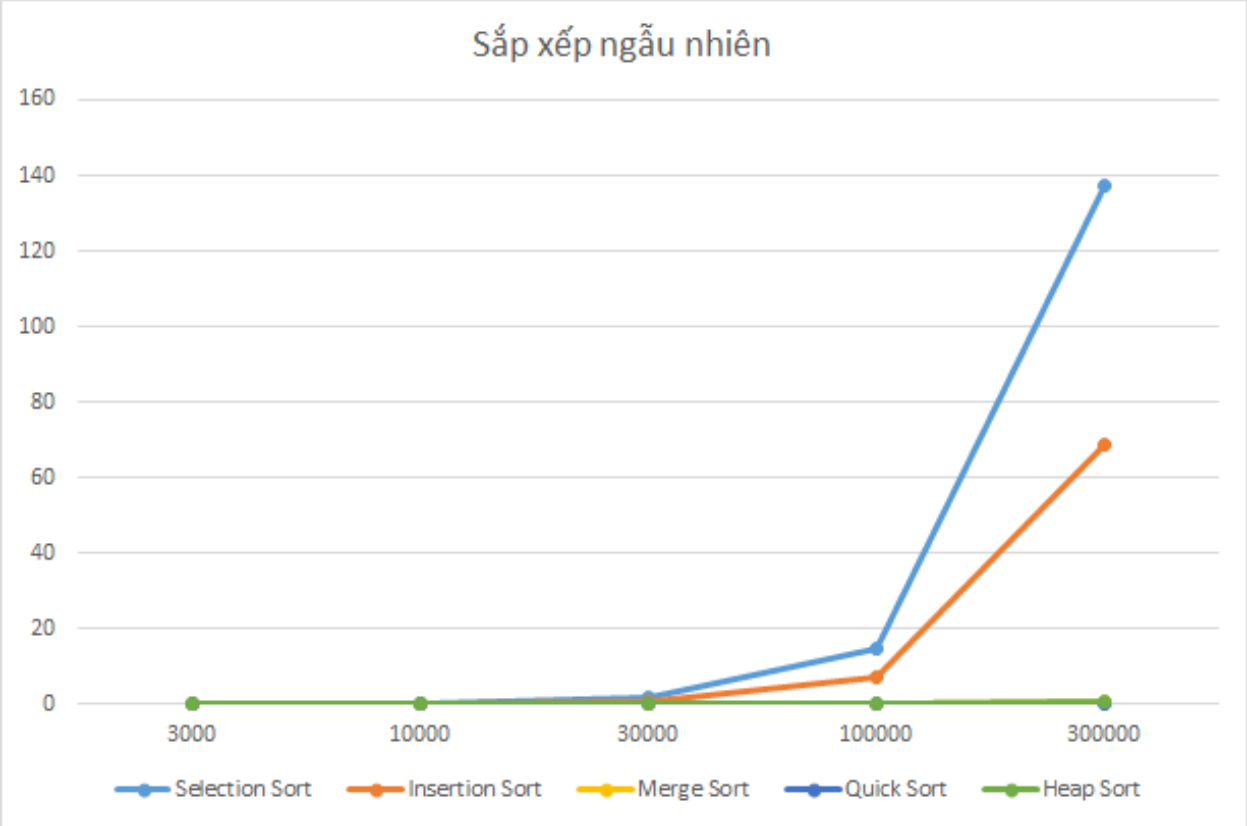
+**Độ phức tạp về không gian:** O(1)

II. KẾT QUẢ THỰC NGHIỆM

1. Với mảng được sắp xếp ngẫu nhiên



Hình 1: Biểu đồ tốc độ chạy của 6 thuật toán với dữ liệu mảng là ngẫu nhiên



Hình 2: Biểu đồ thể hiện tốc độ chạy của 5 thuật toán (trừ Bubble Sort)

+ Từ 2 biểu đồ trên, ta có thể rút ra một vài nhận xét như sau: Đối với kích thước dữ liệu đầu vào thấp như 3000, 10000 hay 30000, dường như không có sự khác biệt quá lớn giữa các thuật toán này.Tốc độ của 3 thuật toán **Quick Sort, Merge Sort, Heap Sort** có nhanh hơn 3 thuật toán còn lại nhưng cách biệt không quá đáng kể. Tuy nhiên, bắt đầu từ kích thước dữ liệu lớn hơn (100000), ta đã thấy được một sự khác biệt lớn. Các thuật toán **Insertion Sort, Selection Sort** và đặc biệt là **Bubble Sort** đã tăng thời gian chạy lên nhiều lần (cụ thể: **Insertion Sort, Selection Sort** trên 10 giây, còn **Bubble Sort** hơn 100 giây) trong khi 3 thuật toán kia vẫn giữ được sự ổn định. Với kích thước dữ liệu lớn nhất (300000), thời gian chạy của 3 thuật toán **Insertion Sort, Selection Sort và Bubble Sort** tiếp tục tăng một cách đáng kể (lần lượt là trên 60 giây, gần 140 giây và trên 1200 giây).

+ Nguyên nhân của việc này là do việc kích thước càng lớn thì số phép so sánh trong 3 thuật toán này càng nhiều, dẫn đến việc thuật toán chạy càng lâu. **Insertion Sort** chạy nhanh hơn một chút so với **Selection Sort** vì mảng được random số liệu nên có thể trong thuật toán này không phải chạy quá nhiều lần vòng lặp thứ 2 nếu đáp ứng được yêu cầu khóa key lớn hơn số sau còn **Selection Sort** dù dữ liệu thế nào thì trong mỗi vòng lặp đều phải chạy đến cuối mảng để tìm phần tử nhỏ nhất**. Bubble Sort** chạy lâu nhất vì nó phải so sánh và hoán vị các cặp phần tử kề nhau trong mỗi lần lặp nên tốn nhiều thời gian để thực hiện. Các thuật toán như **Quick Sort** và **Merge Sort** thì chia nhỏ mảng ra để sắp xếp nên

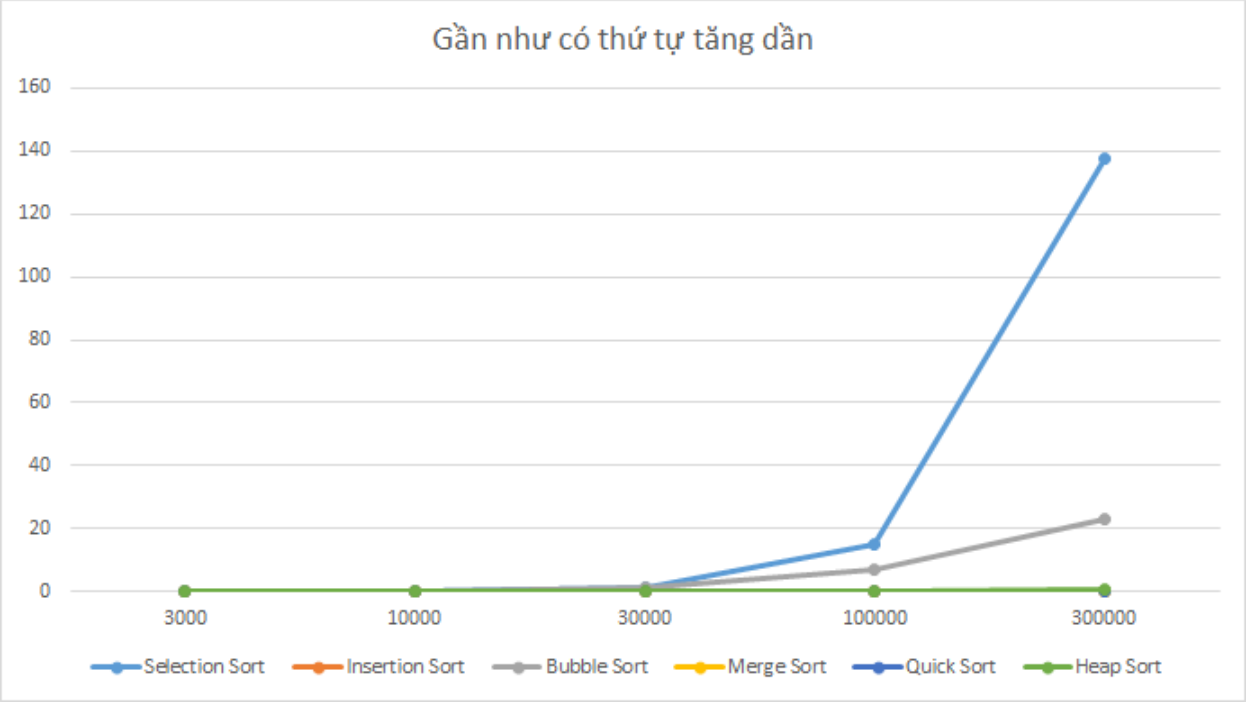
chạy rất nhanh, bù lại thì khá tốn bộ nhớ do phải gọi đệ quy nhiều lần. Còn **Heap Sort** chỉ tốn thời gian để xây dựng cây max-heap ban đầu (xét tất cả node cha) , còn những cây max-heap sau thì chỉ cần xét node gốc là có thể xây dựng lại được nên cũng chạy rất nhanh.

Kết luận:

+ Chạy nhanh : Quick Sort, Merge Sort, Heap Sort

+ Chạy lâu : Selection Sort,Bubble Sort(lâu nhất), Insertion Sort

2. Với mảng gần như có thứ tự tăng dần



Hình 3: Biểu đồ thể hiện tốc độ chạy của 6 thuật toán với mảng gần như sắp xếp

+ Ta thấy tốc độ chạy ở 3 mức kích thước dữ liệu 3000, 10000 và 30000 của 6 thuật toán không có nhiều sự khác biệt lắm. Tuy nhiên ở kích thước 100000 và 300000, 2 thuật toán là **Selection Sort** và **Bubble Sort** lại có tốc độ chạy lâu hơn so với các thuật toán còn lại. Như đã biết, **Selection Sort** dù dữ liệu có như thế nào thì qua mỗi vòng lặp đều phải chạy đến cuối mảng để tìm phần tử nhỏ nhất nên chạy khá lâu (tương đương với thời gian chạy với dữ liệu ngẫu nhiên). Còn với **Bubble Sort**, mảng gần như sắp xếp nên có thể không phải hoán đổi quá nhiều và nếu như đang chạy đến vòng lặp thứ i mà mảng đã được sắp xếp thì không phải chạy vòng lặp đó nữa nên cũng có thể tiết kiệm được thời gian. 3 thuật toán **Quick Sort, Merge Sort** và **Heap Sort** vẫn có tốc độ chạy rất nhanh và ổn định nhưng thuật toán chạy nhanh nhất lại là **Insertion Sort** (thời gian chạy luôn dưới 0.002 giây).

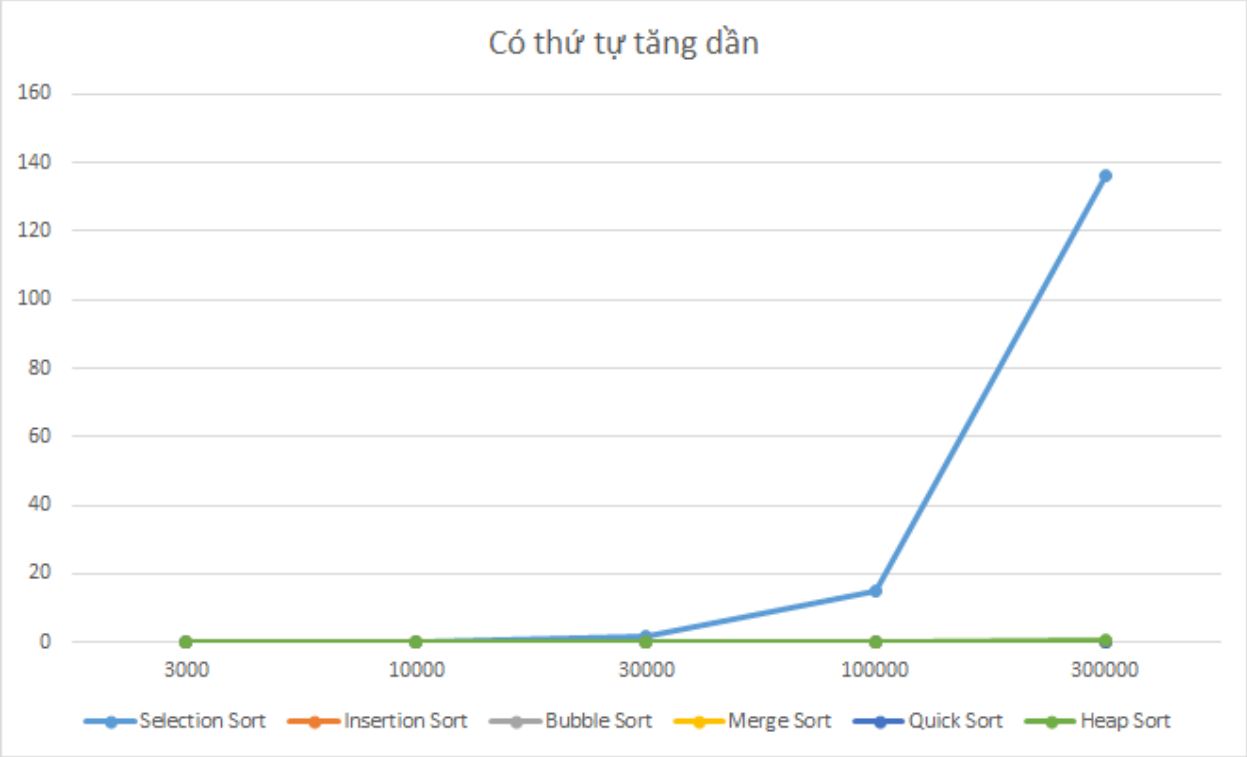
+ Nguyên nhân đến từ việc mảng đã được mảng đã gần như được sắp xếp nên việc thuật toán này phải chạy vòng lặp thứ 2 là không nhiều. Do đó thuật toán này có độ thức tạp về thời gian xấp xỉ O(n), tức là ít nhất trong tất cả các thuật toán.

Kết luận:

+Chạy nhanh: Quick Sort, Merge Sort, Heap Sort, Insertion Sort(nhanh nhất)

+Chạy chậm: Selection Sort(chậm nhất), Bubble Sort

3. Mảng có thứ tự tăng dần:



Hình 4: Biểu đồ thể hiện tốc độ chạy với mảng đã được sắp xếp

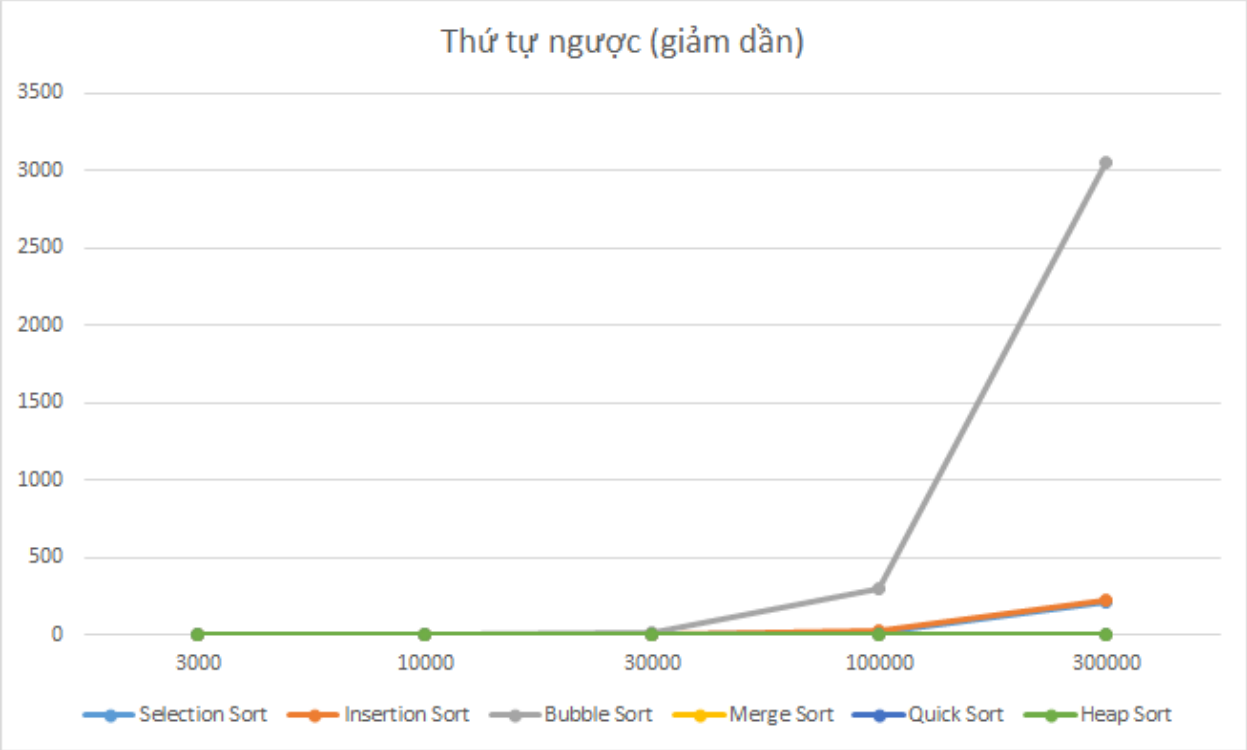
+ Ta thấy rằng **Selection Sort** vẫn chạy với thời gian tương đương như bộ dữ liệu ngẫu nhiên hay gần sắp xếp. Trong khi đó, cả 5 thuật toán còn lại đều chạy với tốc độ rất nhanh. Đặc biệt là 2 thuật toán **Insertion Sort** và **Bubble Sort** chạy nhanh nhất (tốc độ chạy đều dưới 0.002). Điều này là do khi mảng đã được sắp xếp, cả 2 thuật toán này đều không phải thực hiện việc hoán đổi nào và chỉ phải chạy 1 vòng lặp duy nhất (độ phức tạp là O(n)). **Quick Sort** (chọn pivot là phần tử chính giữa) , **Merge Sort** và **Heap Sort** vẫn giữ được sự ổn định về tốc độ chạy. Nhưng nếu như **Quick Sort** chọn pivot là phần tử đầu tiên (giá trị nhỏ nhất) hay cuối cùng(giá trị lớn nhất) , điều này sẽ dẫn đến trường hợp xấu nhất của **Quick Sort** là O(n^2) và việc chia mảng không đều (vì pivot là phần tử nhỏ nhất hoặc lớn nhất, mảng sẽ chia thành 1 phần chứa pivot còn phần mảng kia chứa các phần tử còn lại) dẫn đến việc gọi hàm nhiều lần và có thể bị hiện tượng stack over flow (lỗi tràn stack).

Kết luận:

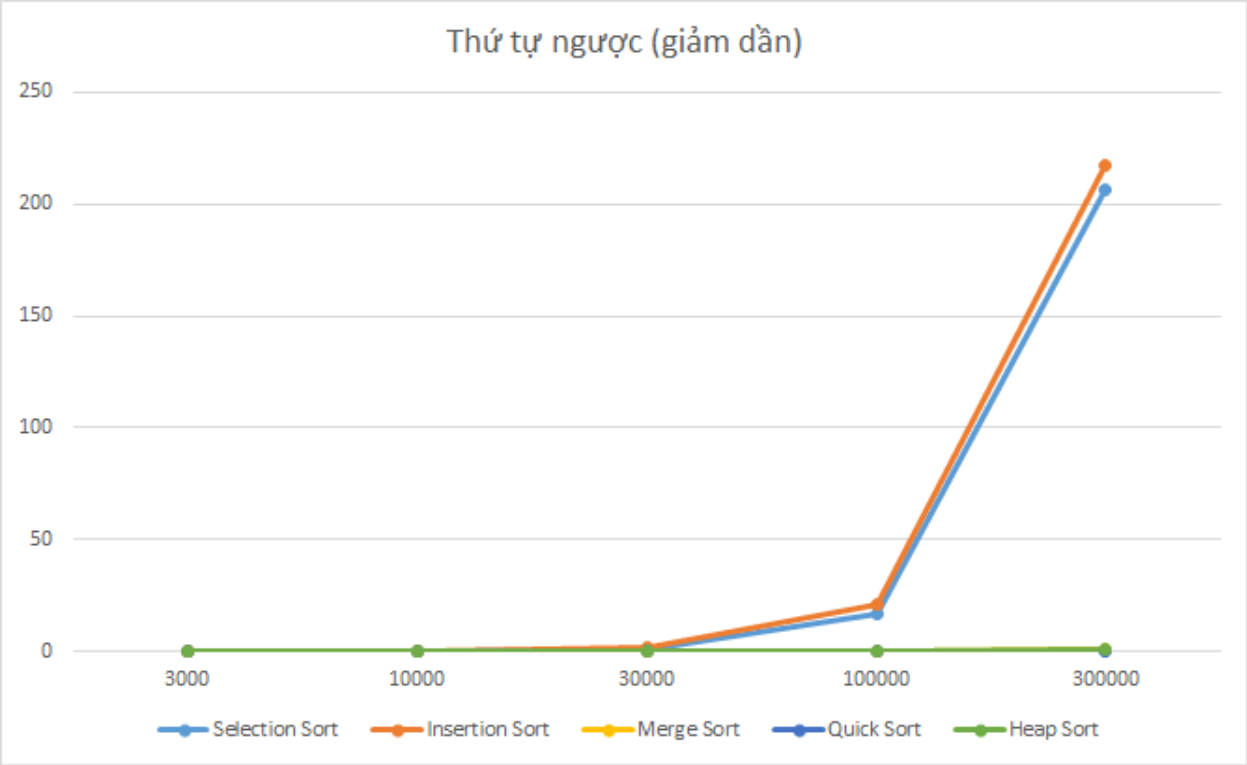
+Chạy nhanh: Bubble Sort và Insertion Sort (nhanh nhất), Quick Sort (chọn pivot là phần tử chính giữa) , Merge Sort, Heap Sort.

+Chạy chậm: Selection Sort, Quick Sort (chọn phần tử ở đầu hoặc cuối).

4. Mảng có thứ tự giảm dần



Hình 5: Biểu đồ thể hiện tốc độ chạy của 6 thuật toán với mảng được sắp xếp ngược (giảm dần)



Hình 6: Biểu đồ thể hiện tốc độ chạy của 5 thuật toán (trừ Bubble Sort) với mảng sắp xếp ngược

+ Ta thấy được sự khác biệt rõ với kích thước dữ liệu là 100000 và 300000. Thuật toán **Bubble Sort** chạy lâu nhất, gấp nhiều lần các thuật toán còn lại. Các thuật toán **Insertion Sort và Selection Sort** xếp ngay sau đó. 3 thuật toán **Quick Sort** (chọn pivot là phần tử chính giữa), **Merge Sort** và **Heap Sort** vẫn giữ được tốc độ chạy nhanh. Thuật toán **Selection Sort** tốc độ có cao hơn các lần trước, có thể là do việc gán minI liên tục nhưng sự khác biệt cũng không quá đột biến. Còn về **Insertion Sort**, đây là trường hợp xấu nhất của thuật toán này (độ phức tạp thuật toán là O(n^2), phải chạy vòng lặp thứ 2 nhiều lần dẫn đến việc tốc độ sắp xếp cũng là lâu nhất trong các trường hợp. **Insertion Sort** vẫn chạy nhanh so với **Bubble Sort** hơn vì thuật toán này không cần hoán đổi mà chỉ cần chèn phần tử vào đúng vị trí của phần mảng đã sắp xếp là được. Còn **Bubble Sort** thì phải thực hiện nhiều lần so sánh và hoán đổi phần tử (vì mảng giảm dần) dẫn đến việc thời gian phải chạy là rất lâu so với các thuật toán khác. Ở trường hợp này thì **Quick Sort** cũng có thể xảy ra trường hợp xấu nhất (đã giải thích ở trên) nên cần lưu ý việc chọn pivot.

Kết luận:

+Chạy nhanh: Quick Sort(chọn pivot là phần tử chính giữa) , Merge Sort, Heap Sort.

+Chạy chậm: Insertion Sort, Selection Sort, Bubble Sort (chậm nhất), Quick Sort(chọn phần tử ở đầu hoặc cuối).

III. KẾT LUẬN

Qua 4 trường hợp trên, ta có thể rút ra vài kết luận như sau:

- Các thuật toán **Quick Sort** (chọn pivot là phần tử chính giữa), **Merge Sort, Heap Sort** sở hữu tốc độ chạy nhanh và ổn định bất chấp kích thước dữ kiệu và trạng thái sắp xếp của mảng.
- Thuật toán **Insertion Sort** chạy nhanh ở trạng thái gần được sắp xếp hoặc đã sắp xếp bất kể kích thước nhưng chạy chậm ở trạng thái dữ liệu ngẫu nhiên và sắp xếp ngược (giảm dần) với kích thước dữ liệu lớn.
- Thuật toán **Selection Sort** chạy ổn định với bất cứ trạng thái sắp xếp nào của mảng. Chạy nhanh với kích thước dữ liệu thấp nhưng chạy lâu với kích thước dữ liệu lớn.
- Thuật toán **Bubble Sort** chạy nhanh ở trạng thái đã được sắp xếp bất kể kích thước nhưng chạy rất chậm ở trạng thái dữ liệu ngẫu nhiên và sắp xếp ngược (giảm dần) với kích thước dữ liệu lớn. Ở trạng thái gần được sắp xếp, thuật toán sỡ hữu tốc độ chạy cũng khá chậm nhưng chấp nhận được với kích thước dữ liệu lớn.
- **Quick Sort** nếu chọn pivot ở đầu hoặc cuối mảng thì sẽ chạy nhanh ở trạng thái dữ liệu ngẫu nhiên hoặc gần được sắp xếp nhưng sẽ chạy lâu và tốn bộ nhớ ở trạng thái đã được sắp xếp hoặc sắp xếp ngược.

+ Thuật toán ổn định: Quick Sort (chọn pivot là phần tử chính giữa) , Merge Sort, Heap Sort.

+ Thuật toán không ổn định: Insertion Sort, Bubble Sort , Quick Sort(chọn pivot là phần tử ở đầu hoặc cuối).

IV. TÀI LIỆU THAM KHẢO

+ Trang web: https://www.geeksforgeeks.org/sorting-algorithms/?ref=ghm

+ Trang web: https://www.programiz.com/dsa