Tên: Lưu Trường Dương

MSSV: 19120489

BÀI BÁO CÁO VỀ CÁC THUẬT TOÁN SORT

1.Selection sort

-Thuật toán Selection sort sắp xếp một mảng bằng cách đi tìm phần tử có giá trị nhỏ nhất(trường hợp xếp theo dãy tăng) trong đoạn đoạn chưa được sắp xếp và đổi chỗ phần tử nhỏ nhất đó với phần tử đứng đầu đoạn chưa được sắp xếp(đoạn đã cắt ra phần đã được sắp xếp trước đó hay vị trí ngay sau đoạn đã được sắp xếp trước đó). Thuật toán sẽ chia mảng làm 2 mảng con, 1 mảng con đã được sắp xếp. 1 mảng con chưa được sắp xếp. Sau mỗi lần lặp thì phần tử bên mảng chưa được sắp xếp sẽ được di chuyển về cuối đoạn đã sắp xếp.

Giải thuật:

Bước 1: $i = 1$

Bước 2: Tìm phần tử $a[min]$ nhỏ nhất trong dãy $a[i..N]$.

Bước 3: Hoán vị $a[min]$ và $a[i]$

Bước 4: Nếu $i \leq N - 1$ thì $i = i + 1$. Lặp lại Bước 2

Ngược lại: Dừng.

a[i]	3	4	8	9	5	7	6
i	0	1	2	3	4	5	6

Hình 1

Như ở hình 1 đoạn(mảng) đã được sắp xếp là đoạn từ i=0 đến i=1, đoạn(mảng) chưa được sắp xếp là đoạn từ i=2 đến i=6.

- Ví dụ:

Lần lặp	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
	6	5	4	1	8	9	2
1	1	5	4	6	8	9	2
2	1	2	4	6	8	9	5
3	1	2	4	6	8	9	5
4	1	2	4	5	8	9	6
5	1	2	4	5	6	9	8
6	1	2	4	5	6	8	9

Đầu tiên tìm phần tử nhỏ nhất trong mảng chưa được sắp xếp a[i](i từ 0 đến 6) là a[3] = 1. Sau đó đổi chỗ nó với phần tử đầu tiên của mảng chưa được sắp xếp là a[0]. Lúc này độ dài của mảng đã được sắp xếp được tăng thêm 1 ô là a[0] và độ dài của mảng chưa được sắp xếp bị giảm 1 ô (từ i=0 đến i=6 thành i=1 đến i=6). Tương tự tìm phần tử nhỏ nhất trong mảng chưa được sắp xếp là a[i](i từ 1 đến 6) là a[6] = 2, rồi đổi chỗ nó với phần tử đầu tiên của mảng chưa được sắp xếp là a[1],… Cứ như vậy cho đến hết, cứ mỗi lần lặp thì mảng đã được sắp xếp sẽ tăng độ dài thêm 1 ô còn mảng chưa được sắp xếp sẽ giảm đi 1 ô. Có trường hợp ở lần lặp thứ 3, mảng không thay đổi do a[2] =4 là giá trị nó nhất của mảng chưa được sắp xếp, và nó nằm ở vị trí đầu tiên của mảng nên nó không cần phải đổi vị trí đi đâu nữa.

- Độ phức tạp của thuật toán là O(n^2) vì có 2 vòng lặp lồng nhau.

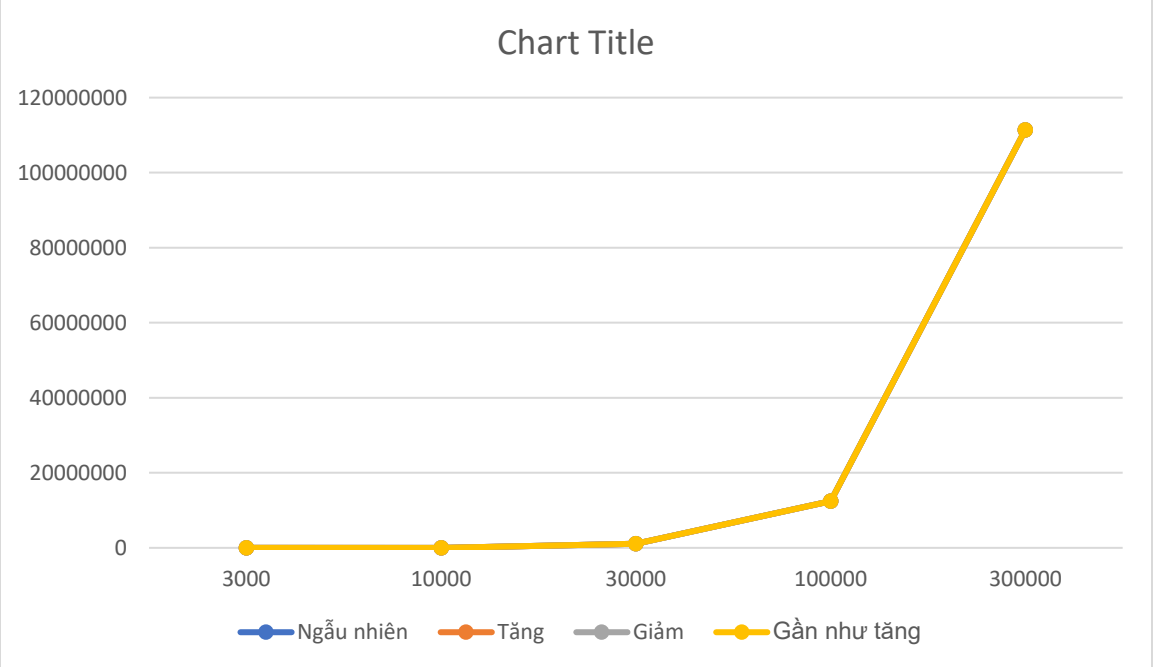
- Số lần so sánh trong trường hợp tốt nhất là n(n-1)/2

- Số lần so sánh trong trường hợp xấu nhất là 3n(n-1)/2

- Không gian bộ nhớ sử dụng: O(1) do thuật toán này không cần phải xin cấp phát thêm bộ nhớ

- Thuật toán có độ phức tạp về mặt thời gian là O(n2) trong mọi trường hợp (vì trong mọi trường hợp thuật toán sẽ tốn (n-1) + (n-2) + … + 1 = n(n-1)/2 = O(n2) phép so sánh, và mỗi lần duyệt, ta luôn phải hoán vị 1 lần (1 hoán vị tương đương với 3 phép gán), nghĩa là thuật toán sẽ tốn 3(n-1) + 3(n-2) + … + 3 = 3n(n-1)/2 = O(n^2) phép gán.)

- Biểu đồ thời gian chạy thử:



-Qua đồ thị ta thấy dù mảng ban đầu là thế nào thì tốc độ xử lý của thuật toán cũng gần như là như nhau

-Khi số lượng phần thử của mảng còn nhỏ thì mảng xử lý với thời gian tương đối nhanh, nhưng khi số lượng phần tử của mảng tăng thì thời gian xử lý tăng nhanh O(n^2)

-> Từ đó ta thấy thuật toán sắp xếp chọn chỉ nên áp dụng với những mảng có số lượng phần tử nhỏ, còn với những mảng có số lượng phần tử lớn thì thuật toán xử lý rất lâu.

=> Nhận xét:

- Ưu điểm:

+ Thuật toán đơn giản, dễ thực hiện

+ Có số lần hoán đổi các vị trí ít

+ Không cần xin thêm bộ nhớ

+ Không phụ thuộc vào dữ liệu đã cho ban đầu

- Nhược điểm:

+ Chỉ áp dụng trong trường hợp số lượng phần tử cần sắp xếp ít, số lượng càng nhiều thời gian sử dụng càng tăng nhanh O(n^2)

+ Hiệu suất không bằng các thuật toán sắp xếp khác

2.Insertion Sort

- Ý tưởng: sắp xếp chèn là thuật toán sắp xếp đơn giản hoạt động như các chúng ta sắp xếp các quân bài trên tay. Mảng sẽ chia thành 2 phần: 1 phần được sắp xếp và 1 phần chưa được sắp xếp. Các giá trị từ phần chưa được sắp xếp sẽ được xếp bào đúng vị trí chính xác trong phần đã được sắp xếp(phần nằm trước nó) sao cho dãy đã được sắp xếp vẫn đảm bảo tính chất của 1 dãy tăng dần.

1. Đầu tiên ta sẽ lấy phần được sắp xếp là đoạn chứa a[0]

2. Sau đó chạy vòng lặp, vòng lặp đầu tiên bắt đầu tại vị trí a[i](i=1). Duyệt ngược từ vị trí i đến vị trí 0, đặt a[i] vào vị trí khi duyệt sao cho dãy số từ a[0] đến a[i] vẫn đảm bảo tính tăng dần. Lúc này số phần tử của phần được sắp xếp tăng lên 1 và vị trí i cũng tăng lên 1 đơn vị.

3. Cứ tiếp tục vòng lặp cho đến khi duyệt hết các phần thử trong mảng

Giải thuật:

Bước 1: $i = 1$

Bước 2: $x = a[i]$;

Tìm vị trí pos phù hợp để a[pos-1] < a[i] < a[pos+1]

Bước 3: Dời chỗ các phần tử từ a[*pos*] đến *a*[*i*-1] sang phải 1 vị trí để dành chỗ cho *a*[*i*].

Bước 4: *a*[*pos*] = *x*; // có đoạn *a*[0]..*a*[*i*] đã được sắp

Bước 5: $i = i + 1$;

Nếu $i \le n-1$: lặp lại bước 2.

Ngược lại: Dừng

- Ví dụ:

a[i] / lần lặp	0	1	2	3	4	5	6	7
	5	8	9	6	4	1	3	2
1	5	8	9	6	4	1	3	2
2	5	8	9	6	4	1	3	2
3	5	8	9	6	4	1	3	2
4	5	6	8	9	4	1	3	2
5	4	5	6	8	9	1	3	2
6	1	4	5	6	8	9	3	2
7	1	3	4	5	6	8	9	2
	1	2	3	4	5	6	8	9

Ghi chú: -màu xanh lá là vị trí i đang xét, màu đỏ là những số đứng trước vị trí i và lớn hơn a[i].

- Phía trước i là đoạn đã được sắp xếp, phía sau i là đoạn chưa được sắp xếp

- Ở lần lặp 1, vị trí i=1 ta thấy 8 đứng ở vị trí đầu tiên của dãy chưa được sắp xếp nên ta sẽ so sánh với những phần tử đứng trước nó. Nhưng không có phần tử nào đứng trước mà lớn hơn nó nên nó vẫn giữ nguyên vị trí. Tương tự với lần lặp thứ 2

- Ở lần lặp thứ 3, vị trí i=3 ta so sánh 6 lần lượt với các phần từ đứng trước nó là 9, 8 và 5. Do 9 và 8 lớn hơn 6 và 6 lớn hơn 5 nên 2 phần tử 9, 8 sẽ dịch sang bên phải 1 đơn vị và phần tử 6 sẽ chèn vào vị trí sau phần tử 5.

-Tương tự như vậy với các vòng lặp còn lại cho đến khi duyệt tới phần tử cuối cùng i=n-1

-Độ phức tạp của thuật toán:

+ Trường hợp tốt nhất độ phức tạp của thuật toán là O(n) sử dụng n-1 phép so sánh và 0 lần hoán vị

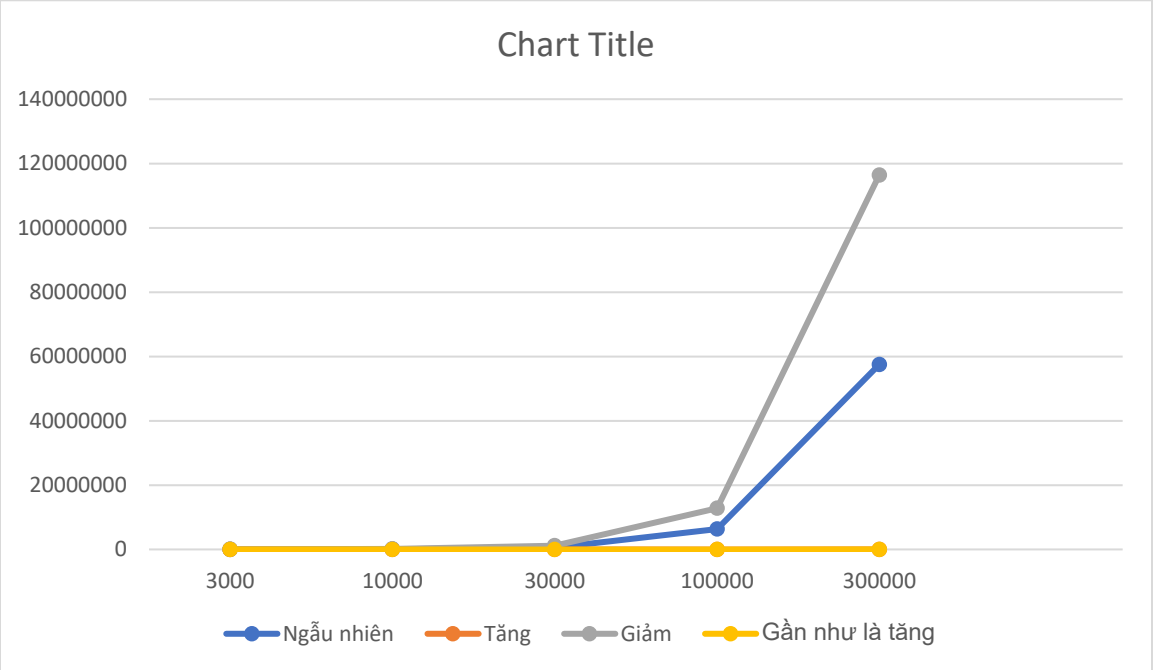
+ Trường hợp trung bình độ phức tạp của thuật toán là O(n^2) thuật toán sử dụng n^2/4 phép so sánh và n^2/4 lần hoán vị

+ Trường hợp xấu nhất độ phức tạp của thuật toán là O(n^2) thuật toán sử dụng n^2/2 phép so sánh và n^2/2 lần hoán vị

- Không gian bộ nhớ sử dụng là O(1) do thuật toán này không cần phải xin cấp phát thêm bộ nhớ

- Trường hợp tốt nhất (best case) : O(n) khi là một mảng đã được sắp xếp theo thứ tự đang cần sắp xếp của
thuật toán, insertion chỉ thực hiện n-1 lần so sánh và không thực hiện phép hoán đổi
- Trường hợp xấu nhất & trung bình (worst case & average case): O(n2) rơi vào trường hợp mảng được sắp xếp
thứ tự ngược hoặc mảng ngẫu nhiên.

- Biểu đồ thời gian chạy thử:



-Qua biểu đồ ta có thể thấy rằng khi số lượng phần tử của mảng ít thì tốc độ xử lý của sắp xếp chèn là khá nhanh và không có quá nhiều khác biệt.

- Khi số lượng phần tử của mảng càng tăng thì trường hợp mảng ngẫu nhiên và mảng giảm có thời gian tăng nhanh O(n^2), mảng giảm có thời gian xử lý luôn cao hơn mảng

ngẫu nhiên. Trong khi đó thời gian xử lý mảng tăng và mảng gần như tăng, tăng rất ít. Thời gian xử lý của mảng gần như tăng luôn cao hơn thời gian sử lý của mảng tăng nhưng chênh lệnh không đáng kể.

-> Từ đó ta thấy thuật toán sắp xếp chèn chỉ phù hợp với mảng đã được sắp xếp một phần hoặc mảng có kích thước nhỏ

⇨ Nhận xét:
Ưu điểm:
+ Thuật toán đơn giản
+ Hiệu suất tốt khi xử lý mảng nhỏ
+ Sắp xếp chèn là sắp xếp tại chỗ nên yêu cầu không gian là tối thiểu
Nhược điểm:
+ Có những thuật toán sắp xếp tốt hơn nó
+ Vì độ phức tạp trung bình là O(n^2) nên sắp xếp chèn không phù hợp để giải quyết tốt một mảng lớn. Mà chỉ thực sự hữu ích khi sắp xếp mảng ít phần tử.

3. Binary-Insertion Sort

- Ý tưởng: Các bước của Binary-Insertion Sort tương tự như Insertion Sort. Nhưng khi chèn một phần tử mới vào phần mảng đã được sắp xếp, chúng ta sử dụng thuật toán tìm kiếm nhị phân để tìm vị trí thích hợp cho nó, thay vì phải ngược từ vị trí i về vi trí 0. Việc tìm kiếm nhị phân sẽ giảm số lần lặp trong trường hợp xấu nhất của Insertion sort là O(n) xuống còn O(logn).

-Các bước:

1. Đầu tiên ta sẽ lấy phần được sắp xếp là đoạn chứa a[0]

2. Sau đó chạy vòng lặp, vòng lặp đầu tiên bắt đầu tại vị trí a[i](i=1). Dùng tìm kiếm nhị phân để tìm kiếm vị trí phù hợp để đặt a[i] vào phần mảng đã được sắp xếp, các phần tử từ vị trí pos được trả về cho đến phần tử i-1 sẽ dịch qua phải 1 ô, phần tử đang xét sẽ chèn vào vị trí pos. Lúc này số phần tử của phần được sắp xếp tăng lên 1 và vị trí i cũng tăng lên 1 đơn vị.

3. Tiếp tục vòng lặp cho đến khi duyệt đến phần tử cuối cùng (i=n-1)

Giải thuật:

Bước 1: $i = 1$

Bước 2: $x = a[i]$; Tìm vị trí *pos* thích hợp trong đoạn $a[0]$ đến $a[i-1]$ bằng tìm kiếm nhị phân để chèn $a[i]$ vào.

Bước 3: Dời chỗ các phần tử từ a[*pos*] đến *a*[*i*-1] sang phải 1 vị trí để dành chỗ cho *a*[*i*].

Bước 4: *a*[*pos*] = *x*; // có đoạn *a*[0]..*a*[*i*] đã được sắp

Bước 5: *i* = *i* +1;

Nếu *i* ≤ *n-1* : lặp lại bước 2.

Ngược lại: Dừng

- Về tìm kiếm nhị phân, đối với mảng đã được sắp xếp thì ta sẽ so sánh phần tử đang xét với phần tử ở giữa của phần mảng đã được sắp xếp, nếu phần tử đang xét lớn hơn thì ta sẽ xét tiếp nửa trên còn lại của mảng đã được sắp xếp, ngược lại ta sẽ xét nửa dưới còn lại của mảng đã được sắp xếp.

Ví dụ:

a[i] / lần lặp	0	1	2	3	4	5	6	7
	8	5	7	1	2	9	4	3
1	5	8	7	1	2	9	4	3
2	5	7	8	1	2	9	4	3
3	1	5	7	8	2	9	4	3
4	1	2	5	7	8	9	4	3
5	1	2	5	7	8	9	4	3
6	1	2	4	5	7	8	9	3
	1	2	3	4	5	7	8	9

Ghi chú: -màu xanh lá là vị trí i đang xét, màu đỏ là những số đứng trước vị trí i và lớn hơn a[i].

- Phía trước i là đoạn đã được sắp xếp, phía sau i là đoạn chưa được sắp xếp

- Đầu tiên ta sẽ xét phần tử a[1]. Vì phần mảng đã được sắp xếp chỉ có 1 phần tử nên khi tìm kiếm nhị phân sẽ trả về vị trí = 0 (do 5<8). Phần tử 8 sẽ dịch qua phải 1 ô và phần tử 5 sẽ được chèn vào vi trí 0

- Sau đó xét tiếp phần tử a[2]. Tìm kiếm nhị phân sẽ trả về vị trí = 1, do đó phần tử ở vị trí i=1 sẽ dịch sang phải 1 ô (từ vị trí 1 đến vị trí i-1=1 sẽ dịch sang phải 1 ô). Tương tự ở lần lặp 2 sẽ trả về pos = 0 và ở lần lặp 3 trả về pos = 1 , ….

- Tương tự cho đến khi duyệt đến phần tử cuối cùng

- Độ phức tạp của thuật toán:

+ Trường hợp tốt nhất : O(n) so sánh và O(1) hoán đổi

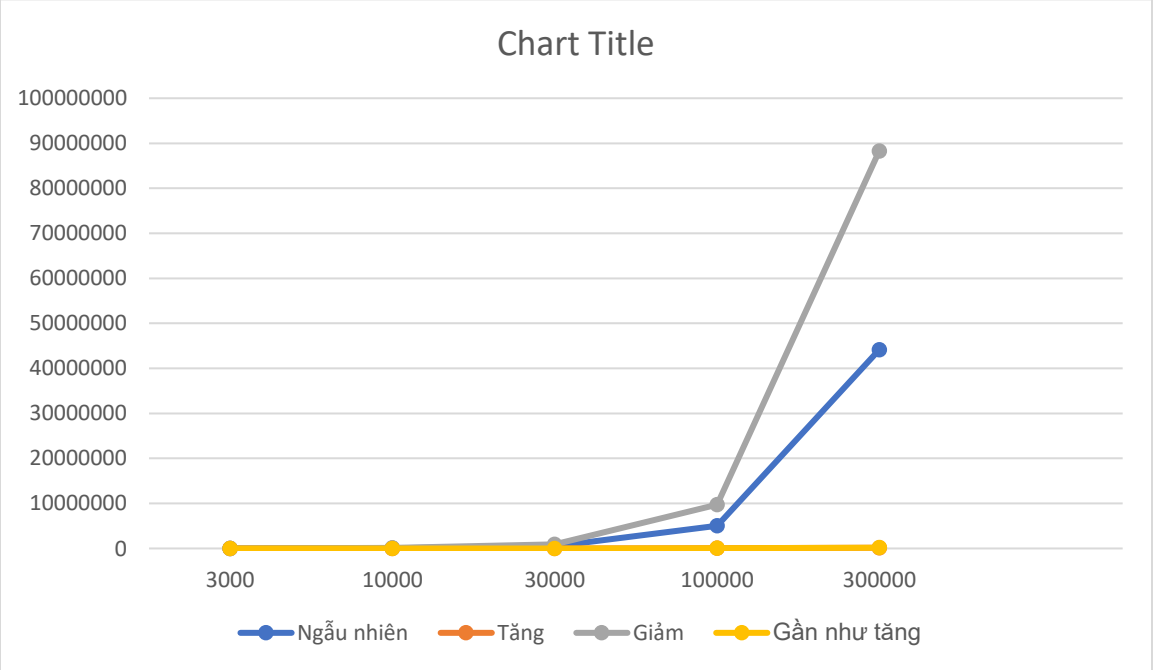
+ Trường hợp trung bình: O(n log(n)) so sánh và hoán đổi

+ Trường hợp xấu nhất: O(n log (n)) so sánh và hoán đổi

- Độ phức tạp không gian: O(1)

- Trường hợp tốt nhất (best case) : O(n logn) vì tìm kiếm nhị phân sẽ thực hiện log2(i+1) lần so sánh với
$1 \leq i < n$ để tìm ra vị trí thứ i, và chi phí cho việc chèn là O(1) với mảng đã sắp xếp
- Trường hợp xấu nhất & trung bình (worst case & average case): O(n2) (vì thực hiện O(n2) chèn phần tử)
Tuy vậy nhưng sắp xếp chèn sử dụng tìm kiếm nhị phân thực hiện ít lần so sánh hơn sắp xếp chèn thông thường,tuy
vậy nhưng thời gian thực thi của cả thuật toán vẫn xấp xỉ thuật toán chèn thông thường

-Biểu đồ thời gian xử lý:



- Qua biểu đồ ta thấy rằng thuật toán xử lý nhanh khi có số lượng phần tử nhỏ

- Khi số lượng phần tử tăng thì thời gian xử lý đối với mảng ngẫu nhiên và mảng giảm tăng nhanh, mảng giảm vẫn có thời gian xử lý lớn hớn mảng ngẫu nhiên. Còn với mảng tăng và mảng gần như tăng thì khi số lượng phần tử tăng thì thời gian xử lý cũng tăng, nhưng không quá đáng kể và ít hơn nhiều so với 2 trường hợp kia. Mảng tăng có thời gian xử lý luôn nhanh hơn thời gian xử lý của mảng gần như tăng

-> Vậy ở trường hợp tốt nhất (mảng tăng) thì thuật toán xử lý không có sự thay đổi nhiều khi tăng số lượng phần tử, và là trường hợp có thời gian xử lý nhanh nhất trong cả 4 trường hợp. Ngược lại mảng giảm là trường hợp xấu nhất, thuật toán sẽ xử lý chậm khi số lượng phần tử tăng O(nlogn) là cũng là trường hợp có thời gian xử lý dài nhất. Vì vậy chỉ nên sử dụng khi kích thước mảng nhỏ mặc dù đã giảm thời gian xử lý đi khá nhiều so với Insertion Sort.

- ⇨ Nhận xét:
  Ưu điểm:
  + Xử lý nhanh với những mảng có số lượng phần tử nhỏ
  + Độ phức tạp không gian là O(n)
  + Xử lý nhanh hơn Insertion Sort ở trường hợp xấu nhất
  Nhược điểm:
  + Cài đặt phức tạp hơn Insertion Sort
  + Chỉ giúp sắp xếp nhanh hơn trường hợp xấu nhất so với Insertion Sort nhưng khi số lượng phần tử của mảng lớn thì vẫn tốn nhiều thời gian để xử lý

4. Bubble Sort

- Ý tưởng: Bắt đầu duyệt từ cuối mảng ngược lại đầu mảng, so sánh từng phần tử với phần tử phía bên trái của nó, nếu nhỏ hơn thì sẽ đổi vi trí của 2 phần tử này. Tiếp tục như vậy cho đến khi phần tử nhỏ nhất nổi lên đầu mảng,.Tiếp tục như vậy sau n-1 bước ta sẽ thu được mảng tăng dần.

Giải thuật:

Bước 1: $i = 0$; // cho i duyệt bắt đầu từ đầu mảng

Bước 2: $j = N-1$; // cho j duyệt chạy ngược từ cuối mảng về đến vị trí i

Trong khi $i<j$

Nếu $a[j] < a[j-1]$: $a[j] \leftrightarrow a[j-1]$;

$j = j - 1$;

Bước 3: $i = i + 1$

Nếu $i < N - 1$: Lặp lại bước 2.

Ngược lại: Hết dãy. Dừng.

- Ví dụ:

a[i] / lần lặp	0	1	2	3	4	5	6	7
	8	5	7	1	2	9	4	3
1	1	8	5	7	2	3	9	4
2	1	2	8	5	7	3	4	9
3	1	2	3	8	5	7	4	9
4	1	2	3	4	8	5	7	9
5	1	2	3	4	5	8	7	9
6	1	2	3	4	5	7	8	9

- Ở lần lặp đầu tiên

8	5	7	1	2	9	4	3

Sẽ đổi vị trí của 4 và 3 với nhau

8	5	7	1	2	9	3	4

Tiếp theo sẽ đổi vị trí của 9 và 3 với nhau

8	5	7	1	2	3	9	4

Tiếp theo vì 3>2 và sau đó là 2>1 nên 2 và 3 giữ nguyên vị trí

8	5	7	1	2	3	9	4
8	5	7	1	2	3	9	4

Vì 1<7 nên 1 và 7 sẽ đổi chỗ cho nhau

8	5	1	7	2	3	9	4

Vì 1<5 nên 1 và 5 sẽ đổi chỗ cho nhau

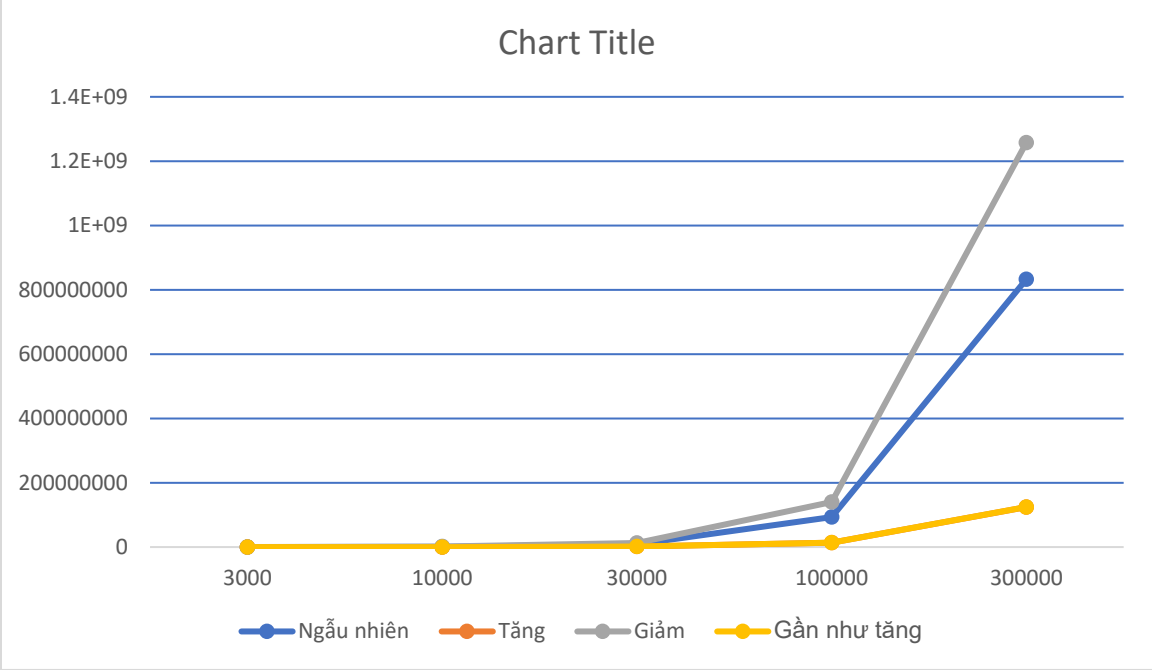
8	1	5	7	2	3	9	4

Vì 1< 8 nên 1 và 8 sẽ đổi chỗ cho nhau

1	8	5	7	2	3	9	4

- Độ phức tạp của thuật toán:
  + Trường hợp tốt nhất O(n)
  + Trường hợp trung bình O(n^2)
  + Trường hợp xấu nhất O(n^2)
- Độ phức tạo về không gian O(1)

- Biểu đồ thời gian xử lý:



-Qua biểu đồ ta thấy rằng thuật toán có thời gian xử lý rất lớn
- Đối với trường hợp mảng ngẫu nhiên và giảm. Khi số lượng phần tử từ 3000 đến 10000 thì thời gian tăng không đang kể. Từ 10000 đến 30000 phần tử thời gian xử lý bắt đầu tăng nhanh hơn và từ 30000 đến 100000 phần tử thời gian xử lý tăng lên

nhanh hơn nữa. Từ 100000 đến 300000 phần tử thời gian tăng đột biến lên rất nhiều lần O(n^2).
- Đối với trường hợp mảng tăng và mảng gần như tăng thì khi số lượng phần tử tăng, thời gian xử lý cũng tăng nhưng tăng không nhanh bằng 2 trường hợp trên.
- Thời gian xử lý mảng tăng là nhanh nhất và mảng giảm là chậm nhất
-> Chỉ nên sử dụng khi số lượng phần tử trong mảng nhỏ hơn 10000, khi số lượng phần tử lớn hơn, thời gian xử lý sẽ rất lâu

=> Nhận xét:

Ưu điểm:

+ Dễ thực hiện

+ Không xin cấp phát thêm không gian

Nhược điểm:

+Xử lý các mảng có số phần tử lớn chậm

+ Yêu cầu n^2 bước xử lý cho n phần tử cần sắp xếp

+ Không phù hợp với thực tế, khó áp dụng vì nếu sort số lượng ít thì Insertion Sort sẽ nhanh hơn còn sort nhiều thì sẽ có nhiều loại sort nhanh hơn nữa.

5. ShakerSort

- Ý tưởng: Cũng dựa trên nguyên tắc đổi chỗ trực tiếp như Bubble Sort, nhưng tìm cách khắc phục nhược điểm của Bubble Sort. Đưa phần tử nhỏ nhất về đầu mảng còn phần tử lớn nhất về cuối mảng từ đó giảm thời gian sắp xếp mảng do giảm được độ lớn của phần mảng chưa được sắp xếp ở lần lặp tiếp theo.

Giải thuật

Bước 1: *l*eft= 0; *right = n-1*; // từ *l* đến *r* là đoạn cần sắp xếp

k = 0; // ghi nhận vị trí k xảy ra hoán vị sau cùng

Bước 2:

Trong khi left < right

Cho i chạy từ left cho đến right

Nếu $a[i] > a[i+1]$: $a[i] \leftrightarrow a[i+1]$;

k=i ; // Lưu lại vị trí hoán vị

// đưa phần tử lớn nhất xuống cuối mảng

right=k; // loại hết các phần tử đã có thứ tự ở cuối mảng

Cho i chạy từ right về left // right có giá trị mới

Nếu $a[i] < a[i-1]$: $a[i] \leftrightarrow a[i-1]$;

// đưa phần tử nhỏ nhất lên đầu mảng

k=i ; // Lưu lại vị trí hoán vị

left = k // loại các phần tử ở đã có thứ tự ở đầu mảng

Bước 3: Nếu left< right : Lặp lại bước 2.

Ngược lại: Dừng.

Ví dụ:

a[i] / lần lặp	0	1	2	3	4	5	6	7
	8	5	7	1	2	9	4	3
1.1	5	7	1	2	8	4	3	9
1.2	1	5	7	2	3	8	4	9
2.1	1	5	2	3	7	4	8	9
2.2	1	2	3	5	4	7	8	9
3.1	1	2	3	4	5	7	8	9

*p/s: 1.1 và 1.2 là lần lặp 1 nhưng tách ra 2 lượt cho dễ hình dung

Khi bắt đầu lần lặp 1, tương tự như ở Bubble Sort, đẩy phần tử lớn nhất mảng về cuối mảng, ở đây là phần tử 9. Sau đó xét mảng mới là mảng đã bỏ phần tử cuối này(right = 6). Tiếp theo đẩy phần tử nhỏ nhất là 1 nổi lên đầu mảng mới (i= 0:6). Sau đó xét mảng mới là mảng đã bỏ phần tử đầu này(left =1).

Tiếp tục vòng lặp mới với left =1 và right =6 tương tự như bước trên cho tới khi left >= right thì dừng vòng lặp.

- Độ phức tạp của thuật toán:

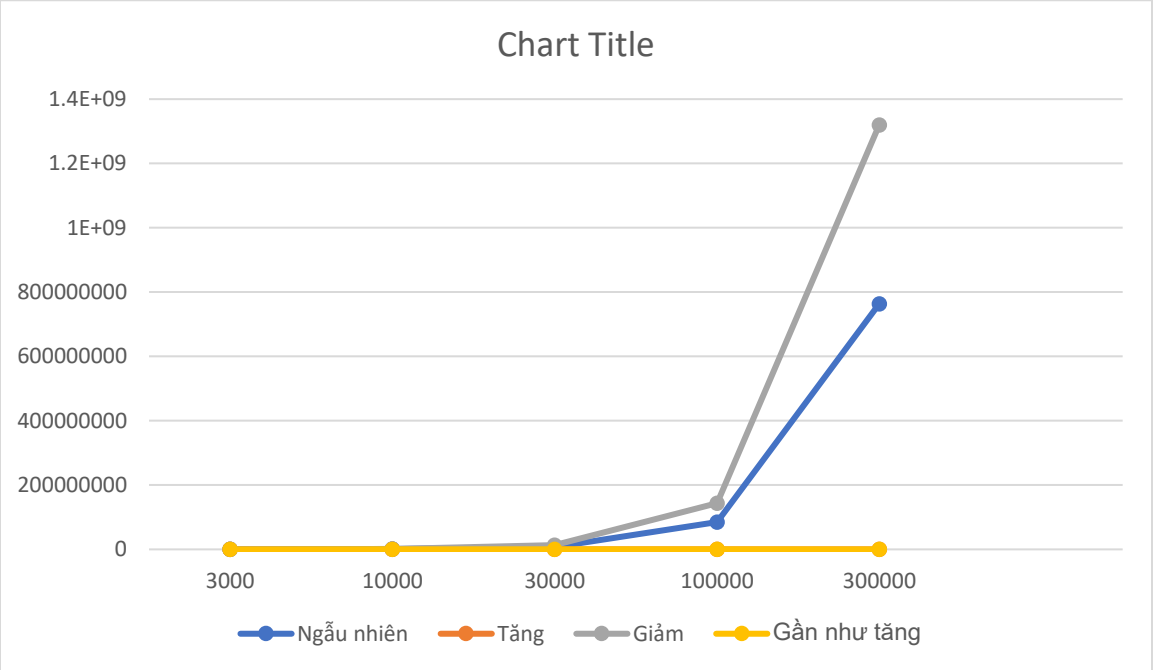
+ Trường hợp tốt nhất là: O(n)

+ Xấu nhất là :O(n^2)

+Trung bình là: O(n^2)

- Độ phức tạp không gian O(1)

-Biểu đồ thời gian xử lý:



- Qua biểu đồ ta thấy rằng từ 3000 – 10000 phần tử thì tốc độ xử lý của thuật toán nhanh. Từ 10000 đến 30000 phần tử thì thời gian xử lý tăng nhanh. Từ 30000-100000 phần tử thì thời gian xử lý tăng rất nhanh. Từ 100000 đến 300000 phần tử thời gian tăng đột biến lên rất nhiều lần O(n^2).

- Đối với trường hợp mảng tăng và mảng gần như tăng thì khi số lượng phần tử tăng, thời gian xử lý cũng tăng nhưng tăng không nhanh bằng 2 trường hợp trên. Trường hợp biến tăng thì thời gian gần như bằng 0
- Thời gian xử lý mảng tăng là nhanh nhất và mảng giảm là chậm nhất
-> Chỉ nên sử dụng khi số lượng phần tử trong mảng nhỏ hơn 10000, khi số lượng phần tử lớn hơn, thời gian xử lý sẽ rất lâu

=> Nhận xét:

Ưu điểm:

+ Xử lý nhanh hơn Bubble Sort

+ Không xin cấp phát thêm không gian

Nhược điểm:

+Xử lý các mảng có số phần tử lớn chậm

+ Yêu cầu n^2 bước xử lý cho n phần tử cần sắp xếp

+ Không phù hợp với thực tế, khó áp dụng vì nếu sort số lượng ít thì Insertion Sort sẽ nhanh hơn còn sort nhiều thì sẽ có nhiều loại sort nhanh hơn nữa.

6. Shell Sort

- Ý tưởng: Là một biến thể của Insertion Sort. Chia mảng thành nhiều phần với mỗi mảng con là các phần tử cách nhau 1 khoảng cách h trong mảng chính. Sắp xếp các phần tử trong dãy con bằng Insertion Sort. Sau đó giảm khoảng cách h và chia mảng mới nhận được thành các mảng con với mỗi mảng con là các phần tử cách nhau 1 hoảng cách h (h mới đã được giảm đi) trong mảng chính,… Cứ tiếp tục như vậy cho đến khi mảng sắp xếp thành công.

Giải Thuật:

Bước 1: Khởi tạo khoảng cách h ban đầu bằng n/2

Bước 2: Chia mảng thành cách mảng con có khoảng cách h

Bước 3: Sort những mảng con này bằng Insertion Sort

h= h/2

Lặp lại cho tới khi h=1

- Ví dụ:

a[i] / lần lặp	0	1	2	3	4	5	6	7
	8	5	7	1	2	9	4	3
1	2	5	4	1	8	9	7	3
2	2	1	4	3	7	5	8	9
	1	2	3	4	5	7	8	9

*Ghi chú, những phần tử có cùng màu với nhau thì được chia cùng mảng con với nhau

- Đầu tiên ta lấy h=n/2 =8/2 =4, vậy khoảng cách chia thành các mảng con là 4. Ta được các mảng con là {a[0],a[4]} {a[1],a[5]} {a[2],a[6]} {a[3],a[7]}. Sau đó dùng Insertion sort để sắp xếp các phần tử của mảng con này theo thứ tự tăng dần. Ta được kết quả như lần lặp 1. Sau đó lấy khoảng cách h mới là h= h/2 =4/2 = 2, chia được thành 2 mảng con có khoảng cách là 2 như hình. Tương tự cũng dùng Insertion Sort để sắp xếp lại vị trí của 2 mảng con này lại,… cứ tiếp tục như vậy cho đến khi khoảng cách h =1 và mảng được sắp xếp xong.

- Độ phức tạp của thuật toán :

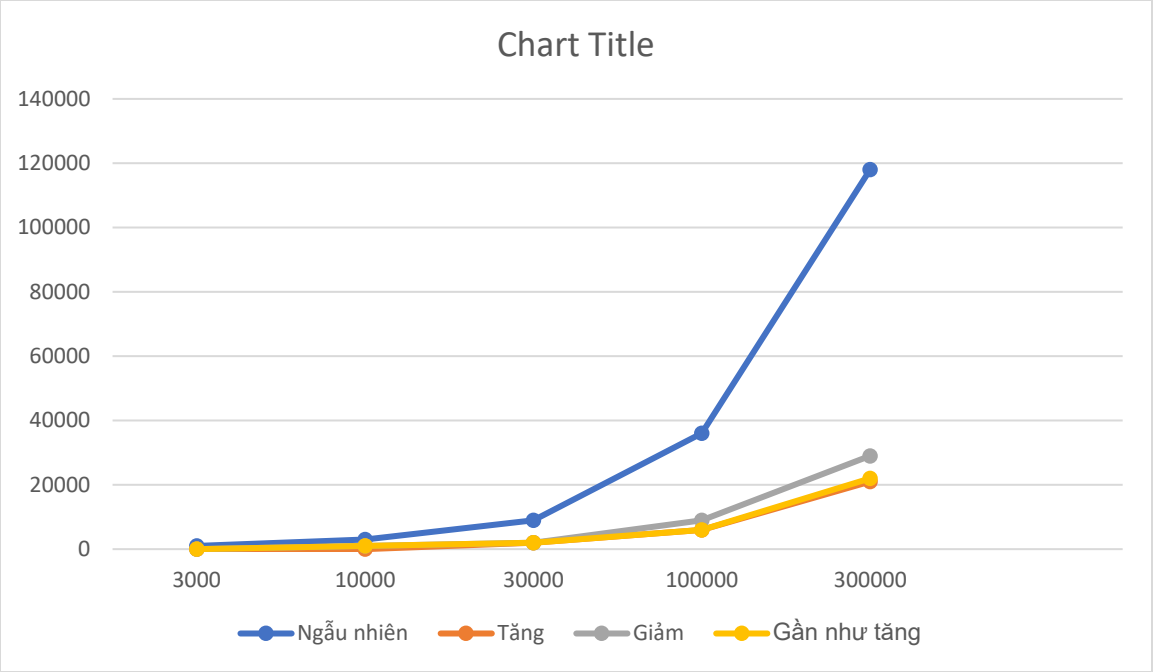
+Trường hợp tốt nhất: O(nlogn)

+Trường hợp xấu nhất: O(n*log2n)

+Trung bình: Gần với O(n) và bé hơn rất nhiều O(n^2)

- Độ phức tạp không gian là O(1)

-Biểu đồ thời gian xử lý:



- Thời gian chạy của thuật toán khá nhanh. Đối với hàm giảm thì có tốc độ chạy nhanh hơn so với các hàm khác.

- Thời gian xử lý mảng có số lượng phần tử dưới 10000 nhanh, có tính ổn định ở đây

-> Chỉ nên dùng với mảng có số lượng phần tử vừa phải, có tốc độ nhanh hơn bubble sort nên có thể áp dụng vào nhiều trường hợp hơn, hiệu quả cao hơn.

⇨ Nhận xét:
Ưu điểm:
+ Có tác dụng với những mảng có số lượng phần tử trung bình
+ Nhanh gấp 5 lần bubble sort và gấp 2 lần insertion sort
Nhược điểm:
+ thuật toán phức tạp, khó hiểu
+ chậm hơn thuật toán Quick sort, merge sort, heap sort
+ Chỉ nên dùng khi mảng có ít hơn 10000 phần tử

7. Heap Sort

- Ý tưởng: Ta sẽ vun các phần tử của mảng lại thành 1 đống , sau khi vun đống thì a[0] tương ứng với root và là giá trị lớn nhất mảng. Ta sẽ hoán vị giá trị a[0] với a[n-1] với nhau và xét tiếp mảng mới là mảng đã bỏ phần tử a[n-1], tiếp tục vun đống mảng và tìm cách để loại đi giá trị lớn nhất còn lại trong mảng đó (đẩy về vị trí cuối mảng) cứ tiếp tục như bậy cho đến khi mảng cần xét chỉ còn 1 phần tử.

- Vung đống: Ta có thể xét mỗi mảng con của a là một cây nhị phân với gốc là phần tử a[i], con bên trái a[2*i+1] và con bên phải là a[2*i+2]. So sánh gốc với 2 con của mình, nếu gốc nhỏ hơn con thì hoán vị gốc với con (nếu cả 2 con đều lớn hơn thì sẽ hoán vị với con có giá trị lớn hơn). Nếu hoán vị thì tiếp tục so sánh nó với 2 con mới của nó cho đến khi hoặc nó lớn hơn 2 con, hoặc nó là lá.

- Để vun một mảng thành đống thì ta vun từ dưới lên với phần tử i đầu tiên là i=n/2 chạy ngược lại cho đến khi i=1.

-Giải thuật:

Bước 1: Vun đống mảng,

m=n ;

// dùng để lưu lại số phần tử còn lại trong mảng chưa được sắp xếp

Bước 2: a[0] ↔ a[m] // hoán vị, đẩy phần tử lớn nhất về cuối

Bước 3: m =m-1; // không xét tới phần tử vừa được đẩy xuống cuối mảng nữa

Bước 4: Nếu m>0 thì lặp lại bước 2 // do còn phần tử chưa xét

Ngược lại : Dừng.

Ví dụ:

a[i] / lần lặp	0	1	2	3	4	5	6	7
	8	5	7	1	2	9	4	3
1	9	5	8	3	2	7	4	1
2	8	4	7	3	2	1	4	9
3	7	4	5	3	2	1	8	9
4	5	3	4	1	2	7	8	9
5	4	3	2	1	5	7	8	9
6	3	1	2	4	5	7	8	9
7	2	1	3	4	5	7	8	9
8	1	2	3	4	5	7	8	9

Ghi chú: Các lần lặp ở trên đã trả được vun đống về max-heap sau mỗi lần lặp

Các phần thử được tô màu xanh dương là những phần tử đã được loại ra

Các phần tử được tô màu đỏ là giá trị lớn nhất của mảng con chưa sắp xếp

Các phần tử tô màu xanh là những phần tử đứng cuối của mảng chưa sắp xếp

Bước 1: Vun đống mảng thành đống max-heap m=7

8	5	7	1	2	9	4	3

Hoán vị phần tử 3 và 1

8	5	7	3	2	9	4	1

Hoán vị phần tử 7 và 9

8	5	9	3	2	7	4	1

Hoán vị phần tử 8 và 9, lúc này đã tạo ra được max-heap

9	5	8	3	2	7	4	1

Bước 2: Hoán vị phần tử a[0] và a[m](m=7)

1	5	8	3	2	7	4	9

Bước 3: m=m-1 =6

Bước 4: do m>0, tiếp tục vun đống phần mảng còn lại

1	5	8	3	2	7	4	9

Hoán vị 1 với 8

8	5	1	3	2	7	4	9

Hoán vị 7 với 1, vun được một max-heap mới

8	5	7	3	2	1	4	9

Bước 5: Tương tự bước 2 hoán vị phần tử a[0] và a[m](m=6)

4	5	7	3	2	1	8	9

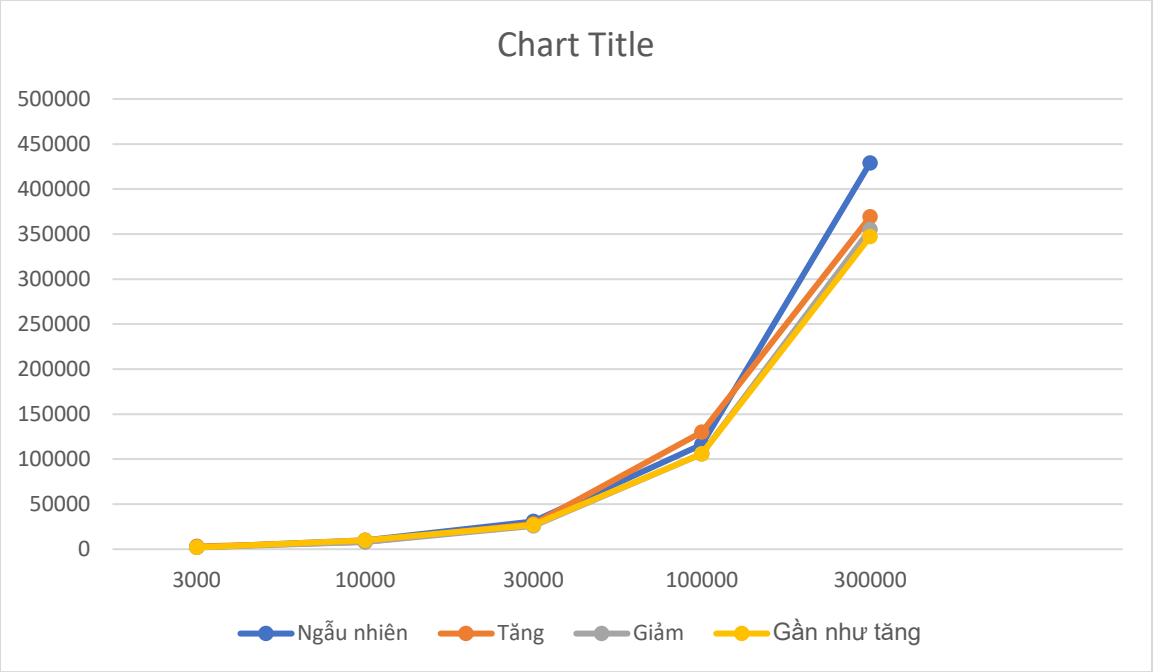
Bước 6: m=m-1=5;

Cứ lặp lại liên tục như vậy cho đến khi m=0 thì dừng.

-Độ phức tạp của thuật toán : O(nlog n) trong mọi trường hợp (nếu dữ liệu đầu vào là mảng các số bằng nhau thì độ phức tạp là O(n)

- Độ phức tạp thời gian: O(1)

- Biểu đồ thời gian xử lý



- Qua biểu đồ ta thấy rằng, trong cả 4 trường hợp, thời gian xử lý của thuật toán là gần như nhau, không có chênh lệch quá rõ ràng.
- Khi số lượng phần từ từ 3000 đến 30000 thì thời gian xử lý tăng chậm, nhưng từ 30000-100000 phần tử thời gian xử lý tăng nhanh dần và từ 100000-300000 thời gian xử lý tăng nhanh hẳn, có sự khác biệt lớn giữa thời gian
- So với những thuật toán đã nêu ở trước thì Heap sort cho thời gian xử lý gần như là nhanh hơn ở trường hợp mảng ngẫu nhiên và mảng giảm, nhưng thời gian xử lý mảng tăng và mảng gần như tăng lại chậm hơn so với các thuật toán trên (ngoài thuật toán selection sort).
  -> Heap Sort dùng được trọng khá nhiều trường hợp và tính ổn định cao, chênh lệch giữa các trường hợp là không lớn, có thể sử dụng để giải quyết các mảng có số lượng phần tử lớn. Nhưng đối với những trường hợp mảng ở trường hợp gần như là tốt nhất thì thuật toán xử lý không nhanh bằng những thuật toán khác.

⇨ Nhận xét:
Ưu điểm:
+ Có thể sử dụng trong nhiều trường hợp vì tính hiệu quả của nó

+ Được thực hiện sắp xếp tại chỗ nên không gian yêu cầu ít
+ Có tính nhất quán trong các trường hợp nên hiệu suất của nó được đảm bảo
Nhược điểm:
+ Đối với các mảng có trường hợp sắp xếp tốt nhất thì thuật toán sẽ sắp xếp lại hết vị trí của mảng
+Việc duy trì heap rất tốn tài nguyên
+Cài đặt phức tạp

8. Merge Sort

- Ý tưởng: Chúng ta sẽ chia mảng thành 2 nửa, sau đó lại chia mỗi phần thành 2 nữa. Tiếp tục như vậy cho đến khi mảng nhỏ nhất chỉ còn 1 phần tử. Gọi mảng bị chia là mảng cha, 2 mảng được chia ra từ mảng cha là mảng con. Ta sẽ so sánh cách mảng con cùng 1 mảng cha và sắp xếp lại chúng sao cho khi 2 mảng con này gộp thành mảng cha thì đã là mảng được sắp xếp. Tiếp tục như vậy cho tới khi gộp lại còn 1 mảng duy nhất, đó là mảng ban đầu đã được sắp xếp lại.

- Đối với hàm trộn: ta sẽ tạo 2 mảng con là L và R để lưu lại giá trị các mảng con đã được chia. So sánh những phần tử đứng đầu của 2 mảng L và R (2 mảng con đã được sắp xếp) để xếp lại lần lượt vào mảng cha, nếu L[0] > R [0] thì mảng cha thêm giá trị R[0] vào phần tử của mình và tiếp tục so sánh L[0] với R[1],… (trường hợp ngược lại thì thêm L[0] vào mảng cha và tiếp tục so sánh L[1] với R[0]),… cho đến khi một trong 2 mảng con L hoặc R hết phần tử, thì mảng còn phần tử sẽ được thêm hết vào mảng cha.

- Giải thuật:

Bước 1: l = 0 ; r = n-1 ;

Tạo hàm mergesort (int a[], int l , int r)

Khi (l<r) thì

m = (l + r)/2

// chia mảng thành 2 mảng con a[l…m] và a[m+1…r]

Bước 2: Gọi lại hàm merge sort để chia nửa mảng a[l…m] thành 2 nữa mảng nhỏ hơn

mergesort (a,l,m);

Bước 3: Tương tự cũng gọi lại hàm merge sort để chia nửa mảng a[m+1..r]

mergesort (a,m+1,r);

Bước 4: Trộn 2 nữa đã được sắp xếp ở bước 2 và 3

// thuật toán sẽ trộn ngược từ những mảng con có số phần tử là 1 cho đến những mảng con có số phần tử là (n-1)/2

merge (a,l,m,r);

Ví dụ:

8	5	7	1	2	9	4	3

Bước 1: Chia mảng 1 thành 2 nửa:

8	5	7	1	2	9	4	3

Bước 2: chia mảng 1.1 (mảng đỏ) thành 2 nửa

8	5	7	1	2	9	4	3

Bước 3: Chia mảng 1.1.1 (mảng đỏ) thành 2 nửa

8	5	7	1	2	9	4	3

Bước 4: Trộn 2 mảng con là màu đỏ và màu tím với nhau ra được mảng cha có sắp xếp {5;8}

5	8	7	1	2	9	4	3

Bước 5: Chia mảng 1.1.2 (mảng xanh lá) thành 2 nửa

5	8	7	1	2	9	4	3

Bước 6: Trộn 2 mảng con lại là màu lục và màu vàng lại với nhau ra được mảng cha có sắp xếp {1;7}

5	8	1	7	2	9	4	3

Bước 7: Trộn 2 mảng con màu đỏ và xanh lá lại với nhau được mảng cha có sắp xếp là {1,5,7,8}. Ở bước trộn này đầu tiên sẽ so sánh 5 và 1 vì 5 >1 nên thêm 1 vào mảng cha trước. tiếp theo so sánh 5 với 7, vì 5 <7 nên thêm 5 vào mảng cha tiếp theo . So sánh 8 với 7, vì 7<8 nên thêm 7 vào mảng cha. Mảng xanh lá đã hết phần tử mà mảng màu đỏ vẫn còn nên những phần tử còn lại của mảng màu đỏ sẽ được thêm lần lượt có thứ tự tăng dần vào mảng cha mà ở đây là phần tử 8.

1	5	7	8	2	9	4	3

Bước 8: chia mảng 1.2 (mảng đen) thành 2 nửa

1	5	7	8	2	9	4	3

Bước 9: chia mảng 1.2.1 (mảng xanh dương) thành 2 nửa

1	5	7	8	2	9	4	3

Bước 10: Trộn 2 mảng con lại là màu làm nhạt và màu lam đậm lại với nhau ra được mảng cha có sắp xếp {2;9}

1	5	7	8	2	9	4	3

Bước 11: chia mảng 1.2.2 (mảng đen) thành 2 nửa

1	5	7	8	2	9	4	3

Bước 12: Trộn 2 mảng con lại là màu tím và màu đen lại với nhau ra được mảng cha có sắp xếp {3;4}

1	5	7	8	2	9	3	4

Bước 13: tương tự bước 7 trộn mảng xanh dương với mảng màu đen được mảng cha có sắp xếp là {2,3,4,9}

1	5	7	8	2	3	4	9

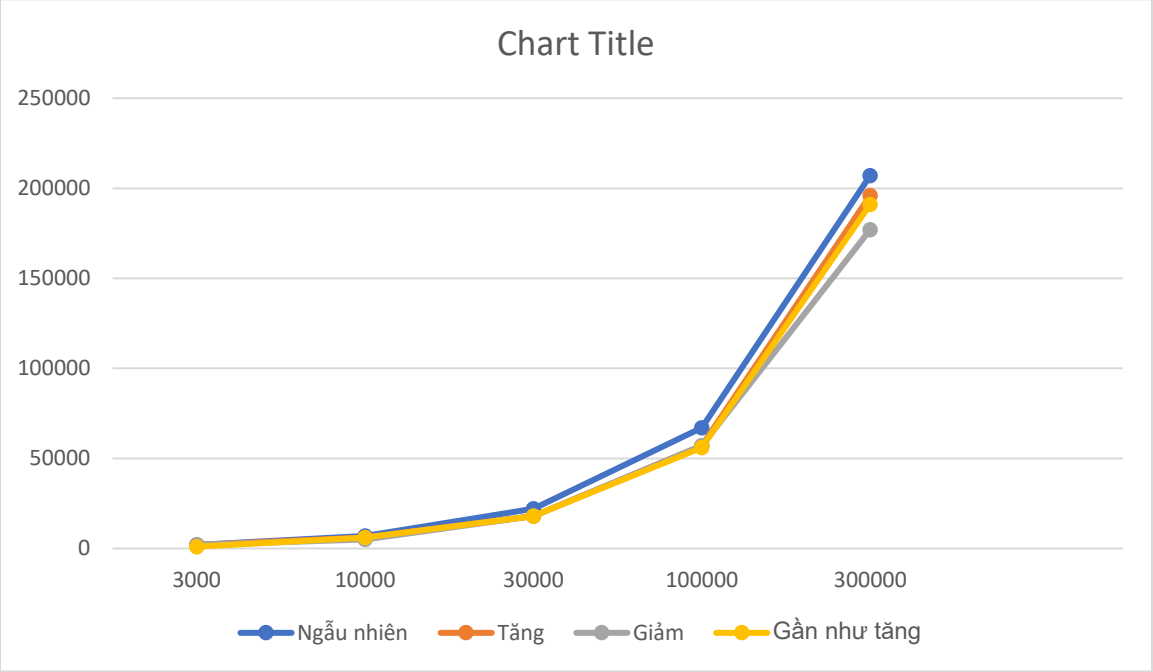
Bước 14: Tương tự gộp 2 mảng đỏ và đen được mảng cha là mảng ban đầu đã được sắp xếp {1,2,3,4,5,7,8,9}

1	2	3	4	5	7	8	9

- Độ phức tạp của thuật toán: O(nlog n) mọi trường hợp

- Không gian bộ nhớ sử dụng : O(n)

- Biểu đồ thời gian xử lý:



- Qua biểu đồ ta có thể thấy rằng thuật toán xử lý ổn định với cả 4 trường hợp, chênh lệch giữa các trường hợp là không quá nhiều

- Từ 3000-30000 phần tử, thời gian xử lý không quá nhanh so với các thuật toán khác

- Từ 30000-300000 thời gian sử lý tương đối nhanh, so với các thuật toán khác

-> Thuật toán có tính ổn định cao khi độ chênh lêch thời gian của các trường hợp là không quá nhiều. Đối với trường hợp mảng ngẫu nhiên hoặc mảng giảm thì thời gian xử lý của nó nhanh hơn so với các thuật toán ở trước nhưng đối với mảng gần như đã được sắp xếp thì nên dùng cac thuật toán khác.

=> nhận xét:

Ưu điểm:

+ Có thể áp dụng cho những mảng có kích thước bất kì vì không giống như chèn, merge sort không đọc lại mảng quá nhiều lần, mà đọc mảng theo 1 cách tuần tự

+Có tính ổn định khi độ phức tạp của thuật toán luôn là O(n logn)

Nhược điểm:

+ Yêu cầu thêm dung lượng O(n) để lưu trữ các mảng con

+ Cần nhiều dung lượng hơn so với các thuật toán khác

+ Sử lý những mảng ít phần tử chậm hơn so với các thuật toán khác

+ Vẫn chạy cả tiến trình ngay cả khi đó là mảng đã được sắp xếp

- Merge sort là thuật toán có độ phức tạp trong mọi trường hợp là O(nlogn), tuy vậy nhưng thuật toán cần không gian bộ nhớ phụ và không tận dụng được thông tin nào về đặc tính của dãy đã sắp xếp, nên trong thực tế người ta sử dụng phiên bản cải tiến của nó là *Sắp xếp trộn tự nhiên (Natural Merge Sort).*

9. Quick Sort

- Ý tưởng: Lấy một giá trị trong mảng gọi là pivot. Sau đó phân vùng mảng ban đầu bằng cách những phần tử lớn hơn phần tử pivot thì đứng sau pivot còn những phần tử mà nhỏ hơn pivot thì đứng trước pivot. Dùng đệ quy để làm tương tự với phần mảng đứng trước pivot và phần mảng đứng sau pivot. Cứ tiếp tục như vậy cho đến khi các mảng con có độ dài bằng 1

- Cách chọn phần tử để làm key của mảng:

+Luôn chọn phần tử đầu tiên của mảng

+ Luôn chọn phần tử cuối cùng của mảng

+Chọn một phần tử random

+ Chọn phần tử có vị trí nằm giữa (l-r)/2

- Giải thuật:

+ Cách chọn giá trị của pivot: dùng median-of-three

Bước 1: mid = (low + high)/2

Bước 2: Nếu (a[mid]<a[low]) thì hoán vị a[mid] và a[low]

Bước 3: Nếu (a[high]<a[low]) thì hoán vị a[high] và a[low]

Bước 4: Nếu (a[high]>a[mid]) thì hoán vị a[high] và a[mid]

Bước 5: pivot = a[hight] // giá trị pivot cần tìm

+Phân vùng:

Bước 1: tìm giá trị của pivot bằng median-of-three

l=low; r=high-1; // Xét mảng đã bỏ qua phần tử pivot

Bước 2: Trong khi (l<r) thì

Trong khi (a[l] < pivot && l<=r) l++ ;

// xét mảng từ trái qua phải tìm phần tử đầu tiên lớn hơn pivot

Trong khi (a[r]>temp && l<=r) r--;

// xét mảng từ phải qua trái tìm phần tử đầu tiên nhỏ hơn pivot

Nếu l<r : a[l] ↔ a[r];

// Hoán vị 2 vị trí vừa tìm được

l++;r--; // do 2 phần tử này vừa xét nên xét các phần tử tiếp theo

Bước 3: a[l] ↔ a[n-1];

// Hoán vị để pivot về đúng vị trí thõa mãn điều kiện phân vùng mảng

Return left;

//Hàm trả lại vị trí của pivot mới hoán vị trong mảng

+QuickSort:

Bước 1: Tạo hàm QuickSort(int arr[],int low,int high) để gọi đệ quy

Bước 2: Nếu (low<high)

Phân vùng mảng và trả về giá trị mà pivot đang đứng là pi

Gọi hàm QuickSort(a, low,pi-1)

// với pi là vị trí của pivot sau khi phân vùng

Gọi hàm QuickSort(a, pi+1,high);

// Gọi đệ qui 2 mảng con trái và phải để tiếp tục phân vùng,…

- Ví dụ:

1	4	8	9	2	3	5

Bước 1; tìm pivot

-Quá trình tìm pivot:

1	4	8	9	2	3	5

Có m=3; median-of0three:

a[3]>a[0] ; a[0]<a[6] ; a[6]<a[3]

nên pivot = a[6];

Bước 2:l=0;r=5 // giá trị l,r ban đầu

Duyệt mảng từ trái qua phải có:

a[0]<pivot nên l=1;

a[1]<pivot nên l=2;

a[2]>pivot nên l=2;

Duyệt mảng từ phải qua trái có:

a[5] < pivot nên r=5;

Hoán vị a[2] và a[5];

Lúc này l=3; r=4; // di chuyển qua vị trí tiếp theo để xét

1	4	3	9	2	8	5

Bước 3: vì l<r(3<4) nên tiếp tục:

Duyệt mảng từ trái qua phải có:

a[3]> pivot nên l=3;

Duyệt mảng từ phải qua trái có:

a[4]<pivot nên r=4;

Hoán vị a[3] và a[4];

Có l=4;r=3; // xét vị trí tiếp theo

1	4	3	2	9	8	5

Bước 4: do l>r nên thoát vòng lặp

Hoán vị a[4] và a[6] (a[l] và pivot)

Bước 5: mảng được chia lại thành 2 phần cam và xanh

1	4	3	2	5	8	9

Bước 6: xét mảng màu cam

Dùng median-of-three tìm được pivot =2

1	4	3	2	5	8	9

Bước7: l=0; r=2 // giá trị ban đầu

Duyệt mảng từ trái qua phải có:

a[0]<pivot nên l=1;

a[1]> pivot nên l=1;

Duyệt mảng từ phải qua trái có:

a[2]>pivot nên r=1;

a[1]>pivot nên r=0;

kiểm tra điều kiện thấy l>r (1>0) nên dừng duyệt

Hoán vị a[1] và pivot

1	2	3	4	5	8	9

Bước 8: Mảng con tiếp tục được chia thành 2 mảng con khác là màu cam và màu xanh dương.

1	2	3	4	5	8	9

Bước 9: Vì mảng vàng chỉ có 1 phần tử nên low = high =0 nên giữ nguyên không chia nữa.

1	2	3	4	5	8	9

Bước 10: Xét mảng xanh dương

Tìm pivot: m =2;

a[2]<a[3] (m < high) nên Hoán vị a[2] và a[3]

1	2	4	3	5	8	9

pivot = a[3] = 3;

1	2	4	3	5	8	9

Bước 11: l=2; r=2 // giá trị ban đầu

Duyệt mảng từ trái qua phải có:

a[2]> pivot nên l=2;

Duyệt mảng từ phải qua trái có:

a[2]>pivot nên r=1;

kiểm tra điều kiện thấy l>r nên thoát vòng lặp

Hoán vị a[2] và pivot

1	2	3	4	5	8	9

Bước 11: xét mảng xanh lá

Tìm pivot: m = 5;

a[5]<a[6] (a[m] < a[high]) nên Hoán vị a[5] và a[6]

pivot =a[6]

1	2	3	4	5	9	8

Bước 12: l=5; r=5 // giá trị ban đầu

Duyệt mảng từ trái qua phải có:

a[5]> pivot nên l=5;

Duyệt mảng từ phải qua trái có:

a[5]>pivot nên r=4;

kiểm tra điều kiện thấy l>r nên thoát vòng lặp

Hoán vị a[5] và pivot

1	2	3	4	5	8	9

Bước 13: Vì mảng xanh lá chỉ có 1 phần tử nên low = high =6 nên giữ nguyên không chia nữa. Ta được mảng ban đầu đã được sắp xếp

1	2	3	4	5	8	9

-Độ phức tạp của thuật toán:

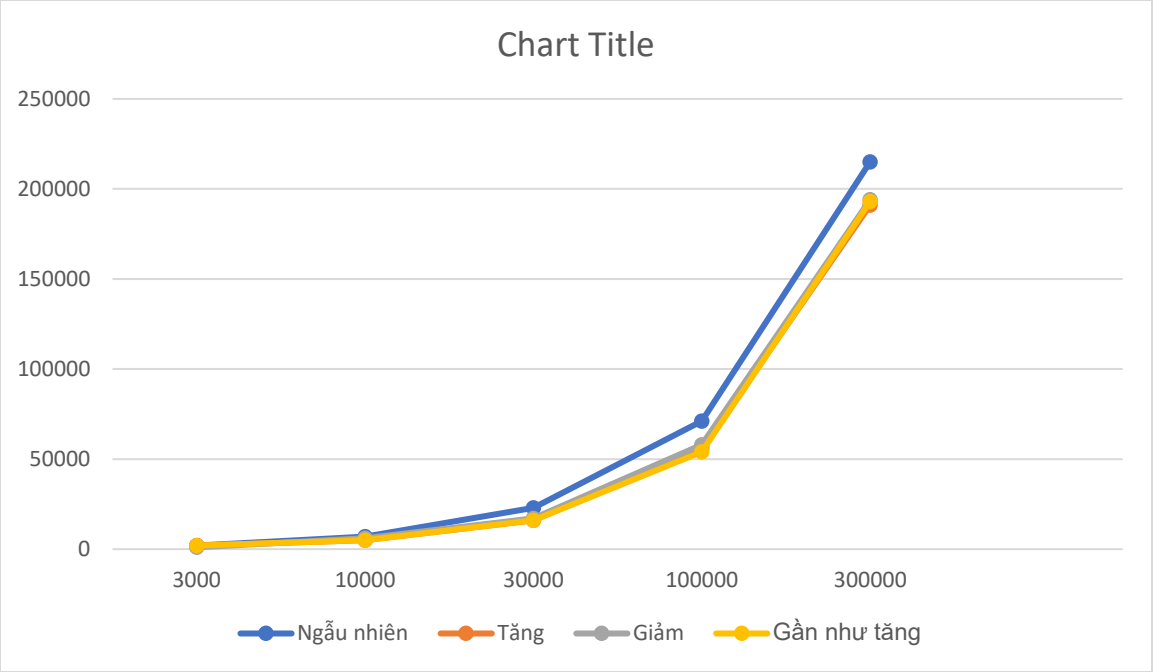
+Trường hợp tốt nhất:O(nlog n)

+ Trường hợp trung bình: O(nlog n)

+ Trường hợp tệ nhất: O(n^2)

- Không gian bộ nhớ: O(logn)

- Biểu đồ thời gian xử lý



- Qua biểu đồ ta vẫn chưa nhìn thấy rõ sự khác biệt giữa các trường hợp do số lượng phần tử chưa đủ lớn. Nhưng nếu chỉ xét với 300000 phần tử thì quicksort có khả năng xử lý nhanh. Những trường hợp mảng ngẫu nhiên hay mảng giảm đều cho về thời gian xử lý nhanh so với các thuật toán đã được nhắc tới. Thời gian xử lý những mảng gần như tăng không nhanh bằng bubble sort .

⇨ Nhận xét:
Ưu điểm:
+Đây có thể coi là thuật toán sắp xếp nhanh nhất
+ Có thể xử lý tốt những mảng có số lượng phần tử lớn
+ Không yêu cầu thêm bộ nhớ do sắp xếp tại chỗ
Nhược điểm:
+Trong trường hợp xấu nhất thời gian xử lý giống như bubble,insertion và selection sort O(n^2)
+Nếu mảng đã được sắp xếp thì buble sort sẽ hiệu quả hơn quick sort
+Ở trường hợp tệ nhất, nếu chọn pivot ở đầu và cuối mảng dễ bị tràn stack.

10. Counting Sort

- Ý tưởng: Ta sẽ dùng 1 mảng C để lưu số lần xuất hiện của các giá trị trong mảng ban đầu A với C[i] là số lần lặp và i là giá trị của phần tử đó. Sau đó dựa vào mảng A và C ta sẽ sắp xếp các phần tử theo thứ tự vào mảng B, đó cũng là kết quả của sắp xếp mảng A.

- Giải thuật

Bước 1: Tìm giá trị lớn nhất của mảng A là max

Bước 2:Khởi tạo mảng C có độ dài là [max+1] với giá trị các phần tử đều bằng 0

Bước 3: Duyệt mảng A và đếm số lượng xuất hiện của các phần tử và lưu vào mảng C với số lượng phần tử x được lưu vào C[x].

Bước 4: Duyệt qua từng phần tử của A và đặt nó vào đúng chỉ số của mảng chứa các giá trị đã sắp xếp B dựa vào C.

Ví dụ:

Bước 1:

A: max = 7

	5	7	6	4	1	3
i	0	1	2	3	4	5

C: Tạo mảng C[max+1]

	0	0	0	0	0	0	0	0
i	0	1	2	3	4	5	6	7

Bước 2:

Duyệt các giá trị trong mảng A

a[0]=5 -> c[5]++;

A:

	5	7	6	4	1	3
i	0	1	2	3	4	5

C:

	0	0	0	0	0	1	0	0
i	0	1	2	3	4	5	6	7

a[1]=7 -> c[7]++;

A:

	5	7	6	4	1	3
i	0	1	2	3	4	5

C:

	0	0	0	0	0	1	0	1
i	0	1	2	3	4	5	6	7

a[2]=6 -> c[6]++;

A:

	5	7	6	4	1	3
i	0	1	2	3	4	5

C:

	0	0	0	0	0	1	1	1
i	0	1	2	3	4	5	6	7

a[3]=4 -> c[4]++;

A:

	5	7	6	4	1	3
i	0	1	2	3	4	5

C:

	0	0	0	0	1	1	1	1
i	0	1	2	3	4	5	6	7

a[4]=1 -> c[1]++;

A:

	5	7	6	4	1	3
i	0	1	2	3	4	5

C:

	0	1	0	0	1	1	1	1
i	0	1	2	3	4	5	6	7

a[5]=3 -> c[3]++;

A:

	5	7	6	4	1	3
i	0	1	2	3	4	5

C:

	0	1	0	1	1	1	1	1
i	0	1	2	3	4	5	6	7

Bước 3: tính lại mảng c theo công thức C[i] =C[i-1] + C[i]

C:

	0	1	1	2	3	4	5	6
i	0	1	2	3	4	5	6	7

Bước 4: Duyệt qua từng phần tử A và đặt nó vào đúng chỉ số của mảng chứa các giá trị đã sắp xếp B dựa vào C:

n=C[7] = 6;

Xét A[0]=5 -> xét C[5]=4 -> xét B[4-1] = A[0] =5;

C[5]--;

A:

	5	7	6	4	1	3
i	0	1	2	3	4	5

C:

	0	1	1	2	3	4	5	6
i	0	1	2	3	4	5	6	7

B:

	0	0	0	5	0	0
i	0	1	2	3	4	5

Xét A[1]=7 -> xét C[7]=6 -> xét B[6-1] = A[1] =5;

C[7]--;

A:

	5	7	6	4	1	3
i	0	1	2	3	4	5

C:

	0	1	1	2	3	3	5	6
i	0	1	2	3	4	5	6	7

B:

	0	0	0	5	0	7
i	0	1	2	3	4	5

Xét A[2]=6 -> xét C[6]=5 -> xét B[5-1] = A[2] =6;

C[6]--;

A:

	5	7	6	4	1	3
i	0	1	2	3	4	5

C:

	0	1	1	2	3	3	5	5
i	0	1	2	3	4	5	6	7

B:

	0	0	0	5	6	7
i	0	1	2	3	4	5

Xét A[3]=4 -> xét C[4]=3 -> xét B[3-1] = A[3] =4;

C[4]--;

A:

	5	7	6	4	1	3
i	0	1	2	3	4	5

C:

	0	1	1	2	3	3	4	5
i	0	1	2	3	4	5	6	7

B:

	0	0	4	5	6	7
i	0	1	2	3	4	5

Xét A[4]=1 -> xét C[1]=1 -> xét B[1-1] = A[4] =1;

C[1]--;

A:

	5	7	6	4	1	3
i	0	1	2	3	4	5

C:

	0	1	1	2	2	3	4	5
i	0	1	2	3	4	5	6	7

B:

	1	0	4	5	6	7
i	0	1	2	3	4	5

Xét A[5]=3 -> xét C[3]=2 -> xét B[2-1] = A[5] =3;

C[3]--;

A:

	5	7	6	4	1	3
i	0	1	2	3	4	5

C:

	0	0	1	2	2	3	4	5
i	0	1	2	3	4	5	6	7

B:

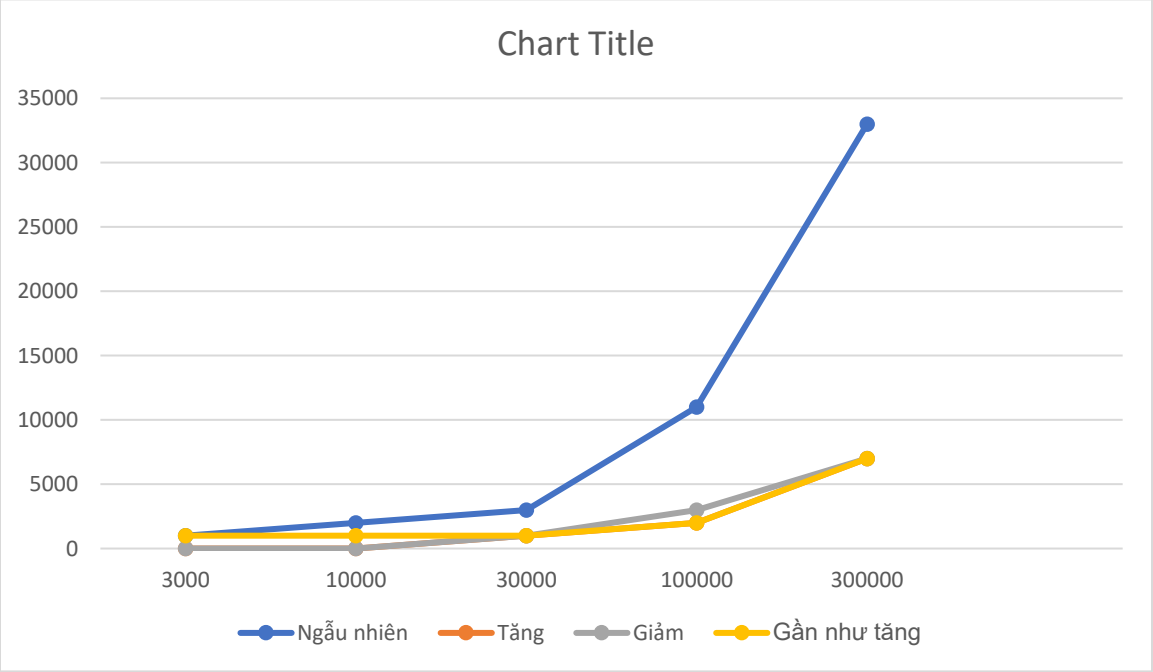
	1	3	4	5	6	7
i	0	1	2	3	4	5

Vậy kể quả của sắp xếp của mảng A là mảng B

-Độ phức tạp của thuật toán: O(n+k) với k là kích thước của mảng phụ

- Không gian sử dụng O(n+k)

-Biểu đồ thời gian xử lý:



- Qua biểu đồ ta thấy rằng tốc độ xử lý của thuật toán couting sort nhanh. Hầu như tất cả các trường hợp tốc độ chênh lệch nhau không quá nhiều

⇨ Nhận xét:
Ưu điểm:
+ Ổn định
+ Tốc độ xử lý nhanh
Nhược điểm:
+ không dùng để sắp xếp string được
+ Yêu cầu bộ nhớ nhiều (gọi thêm mảng phụ)
+ Không nên dùng khi dữ liệu lớn

11. Radix Sort

- Ý tưởng: Giả sử mỗi phần tử a[i] của mảng là một số nguyên có tối đa m chữ số. Phân loại các phần tử này lần lượt theo các chứ số hang đơn vị, hàng chục, hàng trăm,…

- Giải thuật:

Bước 1: tìm giá trị lớn nhất của mảng là max

Bước 2: k là số các chữ số của max

Bước 3: for(i=0:n-1) đặt a[i] vào ô đơn vị t tương ứng với t là chữ số thứ k của a[i]

Bước 4: Nối các phần tử lại với nhau theo thứ tự t tăng dần, các phần tử cùng 1 ô t thì phần tử nào vào trước thì nối trước.

Bước 5: k--;

Tiếp tục bước 3 cho đến khi k=0;

-Ví dụ: Mảng A

564	125	485	157	528	963

Phân theo hàng đơn vị: số nào cùng đơn vị, trong mảng xuất hiện trước thì đứng dưới.

963										
528										
157										
485										
125						485				
564				963	564	125		157	528	
A	0	1	2	3	4	5	6	7	8	9

Mảng A:

963	564	125	485	157	528

Phân theo hang chục:

528										
157										
485										
125										
564			528				564			
963			125			157	963		485	
A	0	1	2	3	4	5	6	7	8	9

Mảng A:

125	528	157	963	564	485

Phân theo hang trăm:

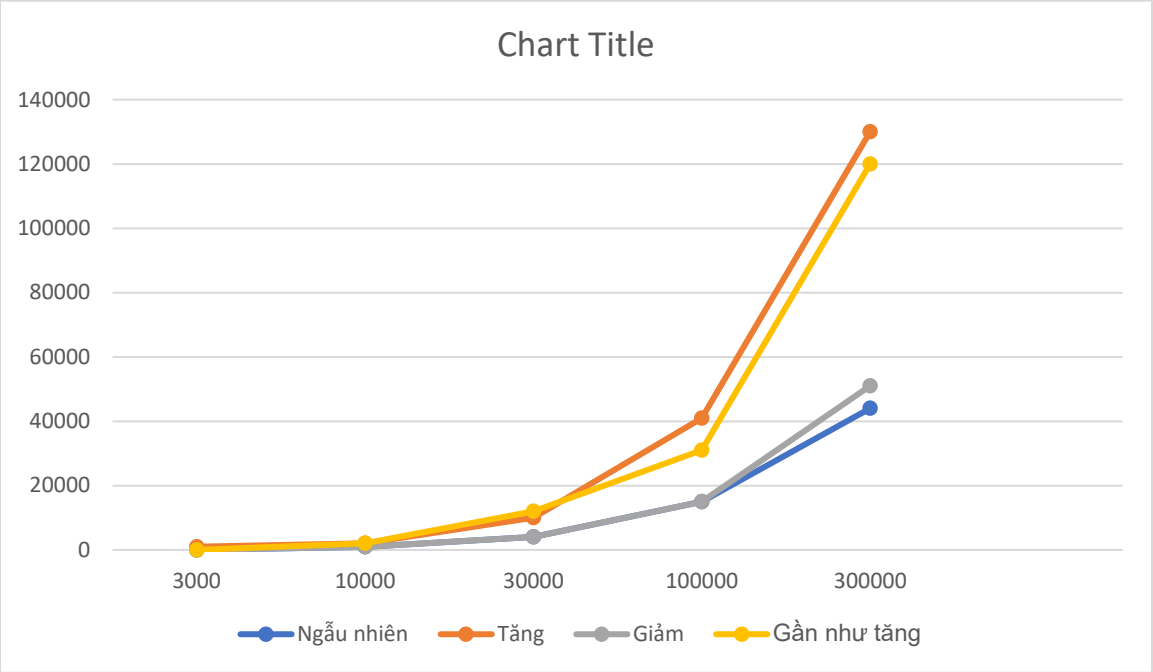
485										
564										
963										
157										
528		157				564				
125		125			485	528				963
A	0	1	2	3	4	5	6	7	8	9

Ta được mảng A được sắp xếp:

125	157	485	528	564	963

- Thuật toán này có độ phức tạp là O(d*(n+b)) với b là loại dữ liệu phân hoạch (nếu cơ số 10 thì b=10,cơ số 2 thì b=2) và d là độ dài của phần tử lớn nhất trong dữ liệu (ví dụ dãy 1...1000 có 1000 là lớn nhất vậy d = log101000 =4)
- Độ phức tạp không gian: O(d*(n+b))

- Biểu đồ thời gian xử lý:



⇨ Nhận xét:
Ưu điểm :
+Có thể nhanh hơn heap sort và quck sort
+Ổn định vì chênh lệch thời gian trong các trường hợp là không quá nhiều
+Làm việc tốt với những phần tử có giá trị nhỏ
Nhược điểm:
+ Không phù hợp cho việc sắp xếp những phần tử có giá trị lớn với vì nó phụ thuộc vào số chữ số của phần tử lớn nhất và số lượng phần tử của mảng
+ Thuật toán phức tạp, sử dụng những phép so sánh không bình thường
+ Cần thêm hàm để chỉnh độ lớn của phần tử và sử dụng các cách để tách phần tử đó thành từng phần nhỏ(đơn vị, chục, trăm,…)

12. Flash sort:

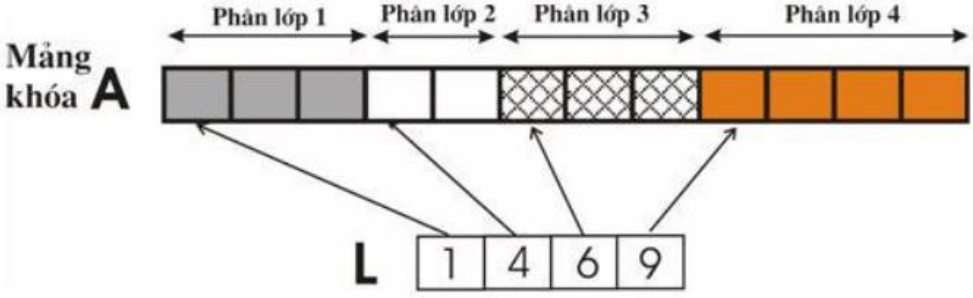
Ý tưởng: Thuật toán Flash sort là thuật toán có thể đạt đến độ phức tạp là O(n) (tuyến tính) mà các thuật toán
thông thường không đạt được (các thuật toán dựa trên so sánh có chặn dưới là O(nlogn)) Thuật toán này được phát minh vào năm 1998 bởi Karl-Dietrich Neubert, ý tưởng chính lại là phân lớp các phần tử.

Thuật toán được chia ra làm ba bước chính:
1. Phân loại các phần tử (Elements Classification):
Vấn đề 1: Xác định giá trị phần tử nhỏ nhất (minval) và vị trí phần tử lớn nhất (nmax) -> duyệt O(n)
Vấn đề 2: Để xác định xem phần tử nào thuộc phân lớp nào, ta dùng công thức sau:

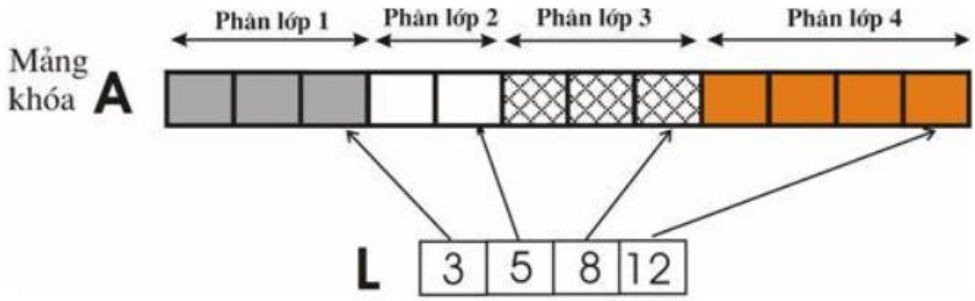
$$K(Ai) = INT((m - 1)\frac{Ai\text{-}Amin}{Amax\text{-}Amin})$$

Vấn đề 3: Ta lập một mảng L ghi vị trí xuất phát của các phân lớp trong mảng A. Ta tưởng tượng (imagine) một qui tắc rằng: một phân lớp I được coi là đã đầy khi mà L[I] nằm đúng ở vị trí xuất phát của phân lớp đó trong mảng A. Chính vì vậy ta coi như một phân lớp là rỗng khi nó nằm ở vị trí cuối của vị trí chính xác của nó trong phân lớp A, để mỗi khi ta bỏ một phần tử vào phân lớp đó, ta lùi L[I] lại 1 cho đến khi về đến đúng vị trí của nó thì nghĩa là đầy. Vậy để xác định được vị trí xuất phát cũng như vị trí kết thúc của từng phân lớp, ta phải biết được kích thước của từng phân lớp. Sau đó, để coi như các phân lớp là rỗng, vị trí bắt đầu của mỗi phân lớp phải nằm ở vị trí lẽ ra là kết thúc của nó trong mảng A. Vậy ta có công thức sau:

$$L[I] = L[I - 1] + kích\ thước\ của\ lớp\ I$$



(Hình 2 – mảng phân lớp L với các phân lớp đã đầy)



(Hình 1 – mảng phân lớp L với các phân lớp rỗng)

2.Hoán vị các phần tử (Elements Permutation):
Giai đoạn tiếp theo là thực hiện sắp xếp các phần tử vào đúng phân lớp của nó. Việc này sẽ hình thành các chu trình hoán vị: mỗi khi ta đem một phần tử ở đâu đó đến một vị trí nào đó thì ta phải nhấc phần tử hiện tại đang chiếm chỗ ra, và tiếp tục với phần tử bị nhấc ra và đưa đến chỗ khác cho đến khi quay lại vị trí ban đầu thì hoàn tất vòng lặp. Với mỗi phần tử ta tính xem nó phải nằm ở phân lớp nào, rồi bỏ nó vào cái vị trí hiện tại của phân lớp đó, và vì ta bỏ
vào phân lớp đó 1 phần tử nên phải lùi vị trí của phân lớp lại 1 đơn vị.
3. Sắp xếp các phần tử trong từng phân lớp theo đúng thứ tự (Elements Ordering):
Trong hai giai đoạn trên ta đã chia nhỏ 1 mảng thành nhiều mảng nhỏ (phân lớp) để việc sắp xếp trở nên nhanh hơn.Công việc tiếp theo là sắp xếp thứ tự các phần tử trong mỗi phân lớp. Thuật toán Straight-Insertion Sort là lựa chọn
tốt cho yêu cầu này.

(tham khảo từ bài viết trong Dr. Dobb's Journal về Flash Sort)

-Ví dụ:

A[i]	5	8	7	4	3	2
i	0	1	2	3	4	5

Bước 1: Tính K của các phần tử, chọn m=n/2

A[i]	5	6	2	7	8	1	3
i	0	1	2	3	4	5	6
K	1	1	0	1	2	0	0

Bước 2: Dùng mảng L lưu tần xuất của K giống counting sort

i	0	1	2
L[i]	3	3	1

Bước 3: Cộng dồn giống counting sort L[i] = L[i-1]+L[i]

i	0	1	2
L[i]	3	6	7

Bước 4: Lưu biến flash = A[0] =5; K = 1 -> L[1] - 1 =5 ; A[5] = 5 và flash sẽ lưu giá trị cũ của A[5]; flash =1

i	0	1	2
L[i]	3	5	7

A[i]		6	2	7	8	5	3
i	0	1	2	3	4	5	6

Bước 5: tiếp tục dò flash =1 có K=0 ->L[0] -1 = 2; A[2] =1; flash = 2

i	0	1	2
L[i]	2	5	7

A[i]		6	1	7	8	5	3
i	0	1	2	3	4	5	6

Bước 6: Tương tự: flash =2 có K=0 ->L[0] -1 = 1; A[1] =2; flash = 6

i	0	1	2
L[i]	1	5	7

A[i]		2	1	7	8	5	3
i	0	1	2	3	4	5	6

Bước 7: flash =6 có K=1 ->L[1] -1 = 4; A[4] =6; flash = 8

i	0	1	2
L[i]	1	4	7

A[i]		2	1	7	6	5	3
i	0	1	2	3	4	5	6

Bước 8: flash =8 có K=2 ->L[2] -1 = 6; A[6] =8; flash = 3

i	0	1	2
L[i]	1	4	6

A[i]		2	1	7	6	5	3
i	0	1	2	3	4	5	6

Bước 9: flash =3 có K=0 ->L[0] -1 = 0; A[0] =3;

i	0	1	2
L[i]	0	4	6

A[i]	3	2	1	7	6	5	3
i	0	1	2	3	4	5	6

Bước 10: Kiểm tra điều kiện thấy i từ 0 đến 2 thõa điều kiện i < L[K]. Nhưng đến i=3 thì không còn thõa điều kiện nên lại gán flash = A[3] = 7; có K=1 ->L[1] -1 = 3; A[3] =7;

i	0	1	2
L[i]	0	3	6

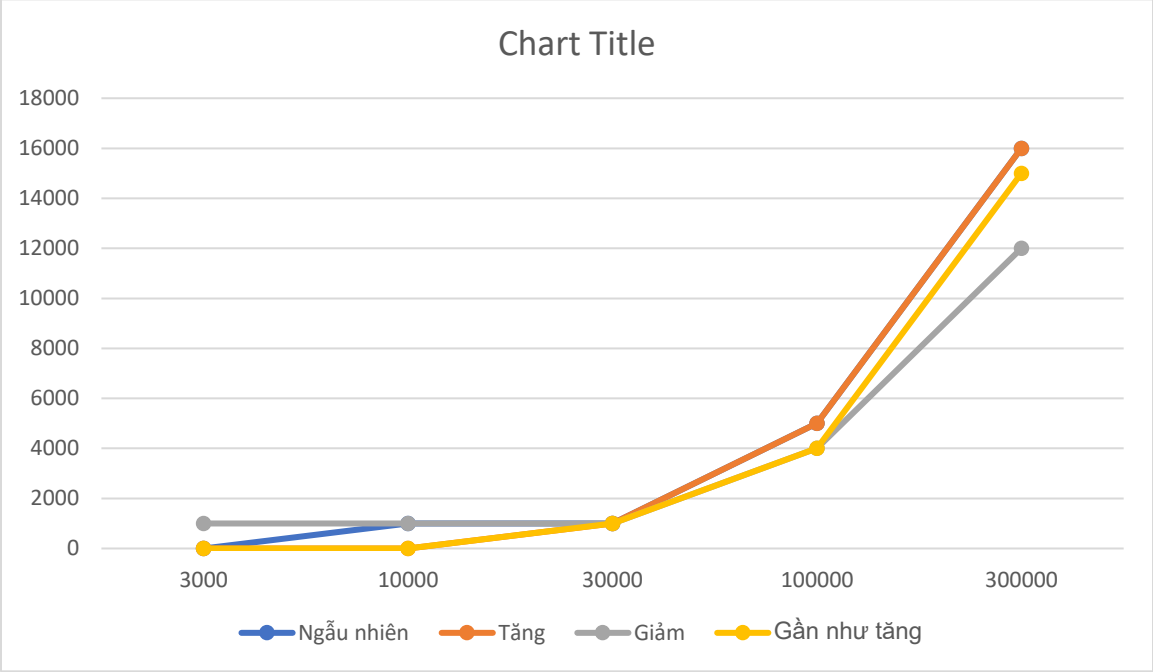
A[i]	3	2	1	7	6	5	3
i	0	1	2	3	4	5	6

Bước 11: những phần tử còn lại khi kiểm tra điều kiện như ở bước 10 thì đều thõa điều kiện nên ta tiếp tục dùng insertion sort để sắp xếp lại các mảng con tương ứng với vị trí mốc L[i]. A[0..2], A[3..5],A[6]. Và được kết quả là mảng được xếp

A[i]	1	2	3	5	6	7	8
i	0	1	2	3	4	5	6

- Độ phức tạp O(n)

- Biểu đồ thời gian xử lý:



- Qua biểu đồ ta thấy tốc độ duyệt của hàm cực kì cao
- Chưa thấy được sự chênh lệch giữa các trường hợp quá nhiều
- Là thuật toán cho thời gian xử lý trung bình tốt nhất cả bài

-> Thuật toán cho hiệu quả làm việc khá cao, nhưng cũng như counting sort và insert sort, khi làm việc với những mảng có nhiều phần tử thì tốc độ xử lý có thể không bằng nhữa thuật toán khác

=> Nhận xét:

Ưu điểm:

+ Là thuật toán có tốc độ xử lý nhanh nhất, có thể đưa độ phức tạp về O(n)

Nhược điểm:

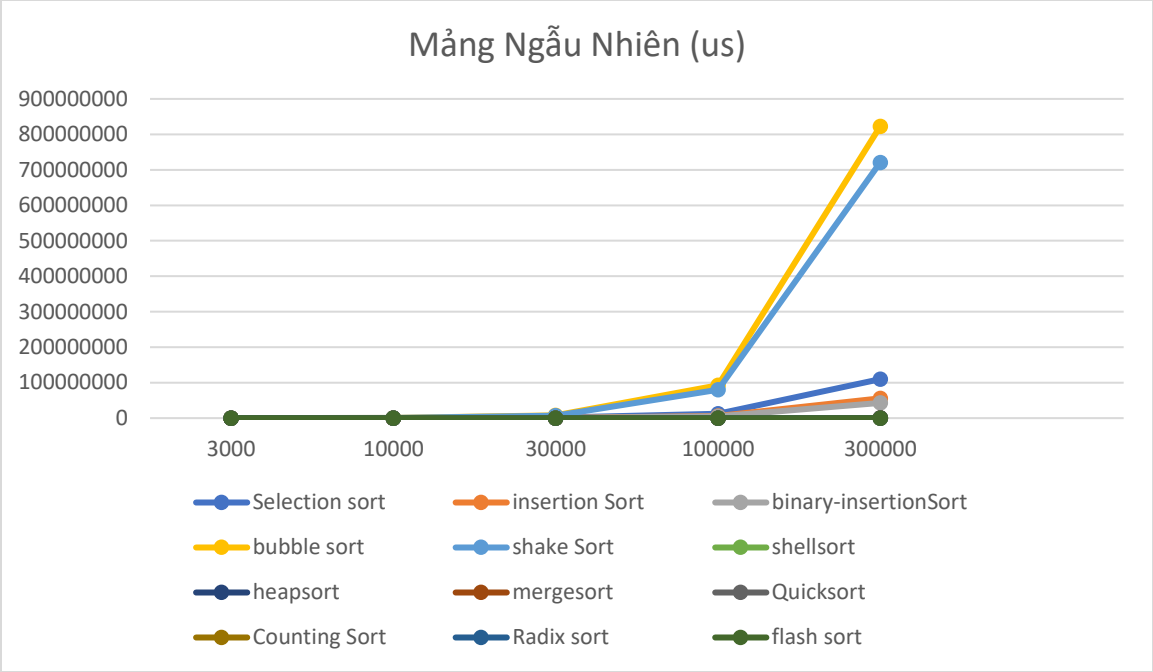
+ Khó cài đặt, thuật toán phức tạp

+ Phải nhờ thêm hàm để dùng counting sort và insertion sort

+ Đối với những dữ liệu có nhiều phần tử và giá trị phần tử cao, tốc độ không cải thiện đáng kể.

13. Biêt đồ nhận xét tổng quá

a) Mảng ngẫu nhiên:



- Khi số phần tử là 3000 thì tuật toán chạy lâu nhất là bubble sort, 3 thuật toán counting sort, radix sort, shell sort và flash sort có tốc độ gần như bằng 0

Thứ tự tốc độ :

flash sort = Radix sort = Counting Sort = shellsort < mergesort = Quicksort < heapsort < binary-insertionSort <insertion Sort < Selection sort< shake Sort< bubble sort

- Khi số phần tử là 10000 thì counting sort chạy nhanh nhất và chạy chậm nhất là buble sort

Thứ tự tốc độ :

Counting Sort < flash sort = Radix sort < shellsort < mergesort = Quicksort <heapsort < binary-insertionSort < insertion Sort < Selection sort< shake Sort< bubble sort

- Khi số phần tử là 30000 thì nhanh nhất là Counting Sort và flash sort, chậm nhất là buble sort

Thứ tự tốc độ :

Counting Sort = flash sort < Radix sort < shellsort < Quicksort <mergesort <heapsort < binary-insertionSort < insertion Sort < Selection sort< shake Sort< bubble sort

- Khi số phần tử là 100000 thì couting sort nhanh nhất, chậm nhất vẫn là bubble sort

Thứ tự tốc độ :

Counting Sort < flash sort < Radix sort < shellsort < Quicksort <mergesort <heapsort < binary-insertionSort < insertion Sort < Selection sort< shake Sort< bubble sort

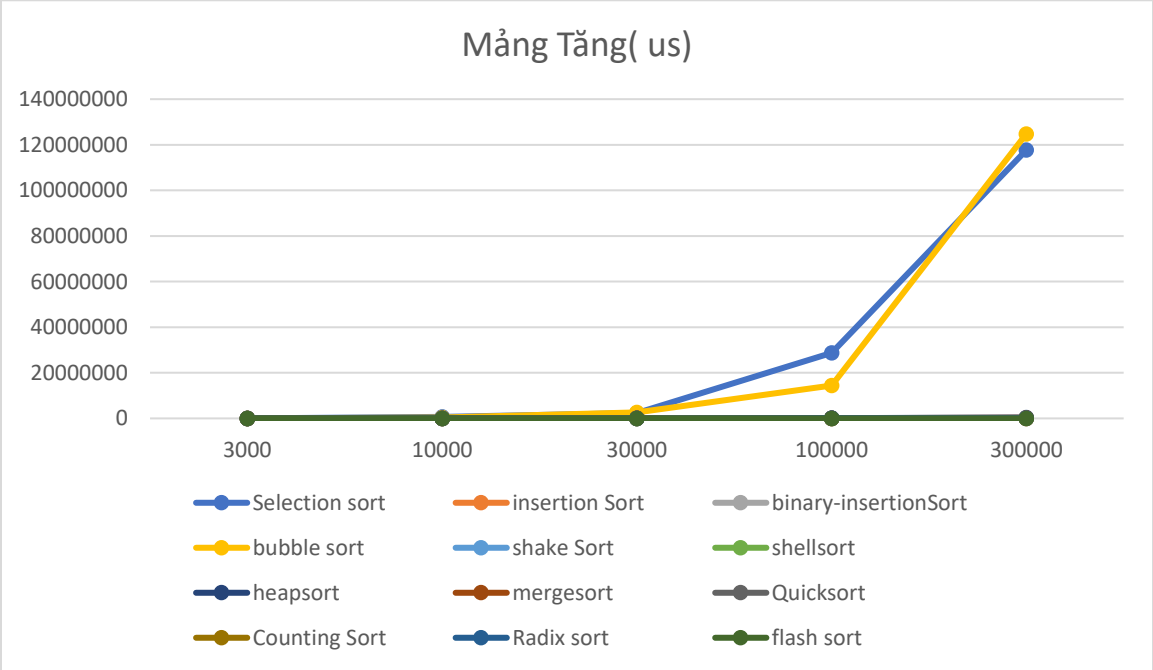
- Khi số phần tử là 300000 thì couting sort nhanh nhất, chậm nhất vẫn là bubble sort

Thứ tự tốc độ :

Counting Sort < flash sort < Radix sort < shellsort < mergesort < Quicksort < heapsort < binary-insertionSort < insertion Sort < Selection sort< shake Sort< bubble sort

Giải thích: Counting Sort chạy nhanh nhất bởi vì độ phức tạp của nó là O(n+k) với k là kích thước của mảng phụ. Mà ở đây mảng đầu vào, phần tử có kích thước lớn nhất cũng chỉ là n(do random 0:n) nên O(n) là trường hợp xấu nhất nến nó chạy nhanh nhất. Bubble sort có độ phức tạp là O(n^2) nên thời gian thực hiện sẽ lâu nhất

b) Mảng Tăng



- Khi số phần tử là 3000 thì flash sort , Counting Sort , shellsort ,shake Sort, và insertion Sort nhanh nhất, có tốc độ gần như bằng 0. Bubble sort có tốc độ chạm nhất

Thứ tự tốc độ :

flash sort = Counting Sort = shellsort =shake Sort=insertion Sort < Radix sort = Quicksort = binary-insertionSort < heapsort < mergesort < Selection sort < bubble sort

- Khi số phần tử là 10000 thì flash sort , shellsort ,shake Sort,insertion Sort có tốc độ nhanh nhất và gần như bằng 0. Selection sort có tốc độ chậm nhất

Thứ tự tốc độ

flash sort = shellsort =shake Sort=insertion Sort< Counting Sort < Radix sort = Quicksort < binary-insertionSort < mergesort < heapsort < bubble sort < Selection sort

- Khi số phần tử là 30000 thì shake Sort và insertion Sort có tốc độ nhanh nhất và gần như bằng 0. Bubble sort có tốc độ xử lý chậm nhất.

Thứ tự tốc độ :

shake Sort=insertion Sort< flash sort = Counting Sort < shellsort <Quicksort< Radix sort < binary-insertionSort < heapsort< mergesort <Selection sort< bubble sort

- Khi số phần tử là 100000 thì shake sort có tốc độ nhanh nhất và gần như bằng 0, seclection sort có tốc độ chậm nhất

Thứ tự tốc độ :

shake Sort<insertion Sort< Counting Sort< flash sort < shellsort <Quicksort< Radix sort < binary-insertionSort < mergesort < heapsort < bubble sort <Selection sort

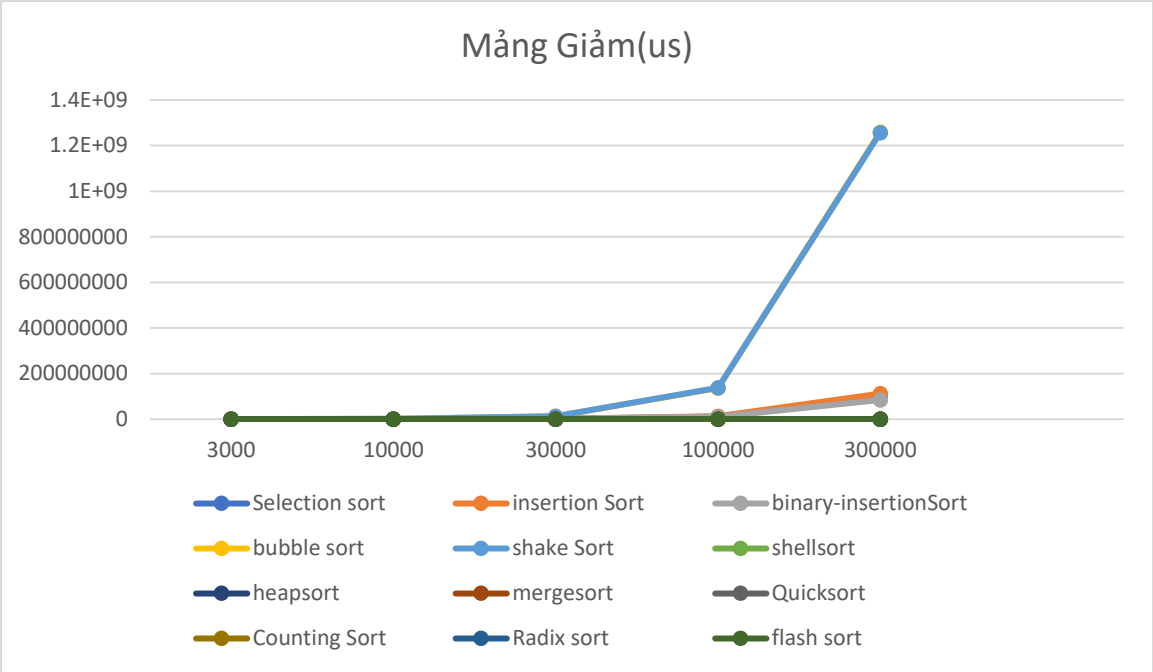
- Khi số phần tử là 300000 thì shake short có tốc độ nhanh nhất và bubble sort có tốc độ chậm nhất

Thứ tự tốc độ :

shake Sort<insertion Sort< Counting Sort< flash sort < shellsort <Quicksort< Radix sort < binary-insertionSort < mergesort < heapsort <Selection sort< bubble sort

Giải thích: Do mảng vào là mảng tăng nên shake sort và insertione sort có độ phức tạp là O(n) tương tự với flash sort vào counting sort. Còn với bubble sort và selection sort thì độ phức tạp vẫn là O(n^2) nên chạy lâu hơn

c) Mảng Giảm



- Khi số phần tử là 3000 thì Counting Sort và Radix sort có thời gian chạy nhanh nhất, ngược lại shake short có tốc độ chạy chậm nhất

Thứ tự tốc độ :

Counting Sort = Radix sort < flash sort = Quicksort = mergesort =shellsort <heapsort<binary-insertionSort<Selection sort< insertion Sort < bubble sort<shake Sort

- Khi số phần tử là 10000 thì counting sort có tốc độ chạy nhanh nhất, bubble có tốc độ chạy chậm nhất

Thứ tự tốc độ :

Counting Sort < Radix sort = flash sort =shellsort< Quicksort <mergesort < heapsort<binary-insertionSort<Selection sort< insertion Sort <shake Sort < bubble sort

- Khi số phần tử là 30000 thì Counting Sort và flash sort có tốc độ chạy nhanh nhất, < bubble sort có tốc độ chạy chậm nhất

Thứ tự tốc độ :

Counting Sort = flash sort <shellsort< Radix sort < Quicksort <mergesort < heapsort<binary-insertionSort<Selection sort< insertion Sort <shake Sort < bubble sort

- Khi số phần tử là 100000 thì Counting Sort có tốc độ chạy nhanh nhất, shake Sort

Có tốc độ chạy chậm nhất

Thứ tự tốc độ :

Counting Sort < flash sort <shellsort< Radix sort < Quicksort <mergesort < heapsort<binary-insertionSort<Selection sort< insertion Sort < bubble sort <shake Sort

- Khi số phần tử là 300000 thì Counting Sort có tốc độ chạy nhanh nhất, bubble sort

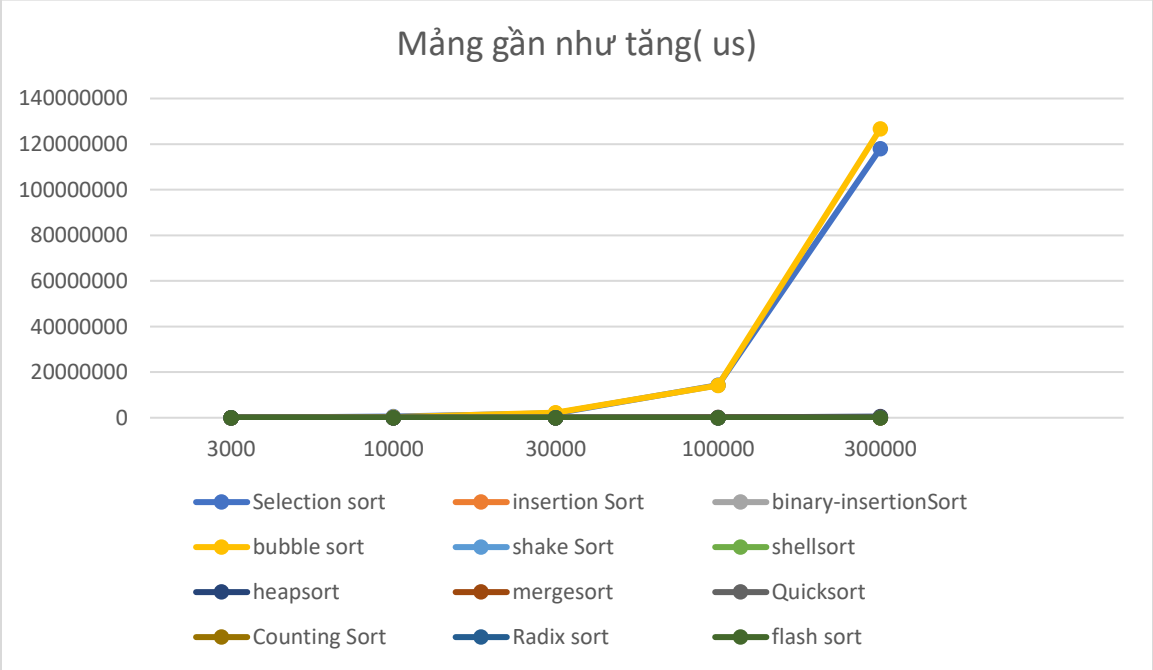
Có tốc độ chạy chậm nhất

Thứ tự tốc độ :

Counting Sort < flash sort <shellsort< Radix sort < Quicksort <mergesort < heapsort<binary-insertionSort<Selection sort< insertion Sort <shake Sort< bubble sort

Giải thích: Counting short tương tự ở trường hợp mảng ngẫu nhiên, độ phức tạp của nó là O(n+k) nên nó chạy nhanh nhất. Còn shake sort và bubble sort rơi vào trường hợp tệ nhất độ phức tạp O(n^2) nên thời gian chạy lâu nhất.

d) Mảng gần như tăng



- Khi số phần tử là 3000 thì insertion Sort,shake Sort,Counting Sort,Radix sort,flash sort có tốc độ nhanh nhất và tốc độ chạy gần như là như nhau, Selection sort, bubble sort có tốc độ chạy chậm nhất

Thứ tự tốc độ :insertion Sort=shake Sort=Counting Sort=Radix sort= flash sort<Quicksort=shellsort=binary-insertionSort<heapsort<mergesort<Selection sort = bubble sort

- Khi số phần tử là 10000 thì insertion Sort, Counting Sort,flash sort có tốc độ nhanh nhất, Selection sort có tốc độ chậm nhất

Thứ tự tốc độ :

insertion Sort =Counting Sort =flash sort<shellsort < Quicksort = Radix sort <shake Sort < binary-insertionSort<mergesort <heapsort < bubble sort<Selection sort

- Khi số phần tử là 30000 thì insertion Sort ,Counting Sort ,flash sort có tốc độ chạy nhanh nhất và ngang nhau, bubble sort có tốc độ chạy chậm nhất

Thứ tự tốc độ :

insertion Sort =Counting Sort =flash sort < Quicksort < Radix sort <shake Sort < shellsort < binary-insertionSort<mergesort <heapsort <Selection sort< bubble sort.

- Khi số phần tử là 100000 thì insertion Sort có tốc độ chạy nhanh nhất, Selection sort có tốc độ chạy chậm nhất

Thứ tự tốc độ :

insertion Sort <Counting Sort <flash sort <shake Sort< shellsort < Quicksort < Radix sort < binary-insertionSort<mergesort <heapsort < bubble sort <Selection sort.

- Khi số phần tử là 300000 thì insertion Sort có tốc độ chạy nhanh nhất, bubble sort có tốc độ chạy chậm nhất

Thứ tự tốc độ :

insertion Sort <shake Sort <flash sort < Counting Sort < shellsort < Quicksort < Radix sort < binary-insertionSort<mergesort <heapsort <Selection sort < bubble sort

Giải thích: Giống như ở phần b, Do mảng vào là mảng tăng nên shake sort và insertione sort có độ phức tạp là O(n) tương tự với flash sort vào counting sort. Còn với bubble sort và selection sort thì độ phức tạp vẫn là O(n^2) nên chạy lâu hơn

e) Nhận xét tổng thể:

- Nhìn chung thì với mỗi trường hợp cụ thể lại có những thuật toán phù hợp cho tốc độ chạy tốt nhất vì vậy tùy vào trường hợp mà ta có thể áp dụng các thuật toán khác nhau.

- Thuật toán Counting sort và flash sort tổng thể có thời gian chạy nhanh trong cả 4 trường hợp

-Bubble sort là thuật toán chạy chậm nhất trong gần như cả 4 trường hợp sau đó là Selection sort.

- Các thuật toán ổn định: Insertion Sort, Insertion-Binary Sort, Bubble Sort, Shaker Sort, Merge Sort, Counting Sort, Radix Sort.

- Còn các thuật toán còn lại không được ổn định, tốc độ có chênh lệch lớn giữa các trường hợp