

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN, DHQG TP. HCM
KHOA CÔNG NGHỆ THÔNG TIN



BÁO CÁO

Cài đặt và thực nghiệm thời gian thực thi của các thuật toán sắp xếp

Giáo viên hướng dẫn : Phan Thị Phương Uyên
Họ và tên sinh viên : Lê Kiệt
MSSV : 19120554
Lớp : 19CTT3B

Thứ 6, ngày 27 tháng 11, năm 2020

MỤC LỤC

A. CÁC THUẬT TOÁN ĐÃ CÀI ĐẶT	1
I. Selection sort	1
II. Insertion sort	2
III. Binary-Insertion sort	4
IV. Bubble sort	5
V. Shaker sort	6
VI. Shell sort	9
VII. Heap sort	16
VIII. Merge sort	19
IX. Quick sort	20
X. Counting sort	22
XI. Radix sort	24
XII. Flash sort	26
B. KẾT QUẢ THỰC NGHIỆM	31
I. Trạng thái sắp xếp của dữ liệu đầu vào: ngẫu nhiên	31
II. Trạng thái sắp xếp của dữ liệu đầu vào: có thứ tự tăng dần	32
III. Trạng thái sắp xếp của dữ liệu đầu vào: gần như có thứ tự tăng dần	34
IV. Trạng thái sắp xếp của dữ liệu đầu vào: có thứ tự giảm dần	35

A. CÁC THUẬT TOÁN ĐÃ CẢI ĐẶT

I. Selection sort

1. Ý tưởng

- Đúng như tên gọi, thuật toán “sắp xếp chọn” sẽ chọn ra phần tử min/max trong phần mảng chưa sắp xếp (unsorted part) và kéo nó về chỉ số hiện tại trong phần mảng đã sắp xếp (sorted part) ở mỗi lần lặp. Như vậy, mảng đã cho sẽ được hiểu như tồn tại 2 phần: phần mảng chưa sắp xếp (unsorted part – ban đầu là toàn bộ mảng đã cho) & phần mảng đã sắp xếp (sorted part – ban đầu rỗng)

2. Thuật toán từng bước

Duyệt mảng từ vị trí $i: 0 \rightarrow n-1$:

Từ vị trí $i+1$ trở đi:

 tìm ra phần tử min

 Hoán vị nó với $a[i]$

Xét ví dụ: Cho mảng gồm 5 phần tử như sau

64	25	12	22	1
Sorted part: NUL Unsorted part: 64, 25, 12, 22, 1 \rightarrow min = 1, chuyển 1 về sorted part				
1	25	12	22	64
Sorted part: 1 Unsorted part: 25, 12, 22, 64 \rightarrow min = 12, chuyển 12 về sorted part				
1	12	25	22	64
Sorted part: 1, 12 Unsorted part: 25, 22, 64 \rightarrow min = 22, chuyển 22 về sorted part				
1	12	22	25	64
Sorted part: 1, 12, 22 Unsorted part: 25, 64 \rightarrow min = 25, chuyển 25 về sorted part				
1	12	22	25	64
Sorted part: 1, 12, 22, 25 Unsorted part: 64 \rightarrow min = 64, chuyển 64 về sorted part				
1	12	22	25	64
Mảng đã được sắp xếp				

3. Đánh giá thuật toán

- Độ phức tạp thời gian: $O(n^2)$ cho mọi trường hợp
- Độ phức tạp không gian: $O(1)$ do không dùng mảng phụ để lưu trữ → đây là thuật toán sắp xếp tại chỗ

4. Ưu, nhược điểm

- Ưu điểm:
 - + Với mảng có kích thước nhỏ phù hợp, selection sort có thể thực hiện sắp xếp với tốc độ tương đối ổn
 - + So với 1 số thuật toán sắp xếp khác (merge sort,...) thì selection sort không cần dùng thêm mảng phụ để lưu trữ
- Nhược điểm:
 - + Với mảng có kích thước n lớn, selection sort chạy rất lâu do cần n^2 bước

II. Insertion sort

1. Ý tưởng

- Giả sử mảng có kích thước n , ta cũng chia mảng đã cho thành 2 phần: phần mảng chưa sắp xếp (unsorted part – luôn có ít nhất 1 phần tử, đó là phần tử đầu của mảng đã cho) & phần mảng đã sắp xếp (sorted part – ban đầu có $n-1$ phần tử còn lại). Tại mỗi phần tử trong unsorted part, ta so sánh tuyến tính với lần lượt các phần tử trong sorted part để kiểm vị trí cần chen & chen nó vào trong sorted part

2. Thuật toán từng bước

Duyệt mảng từ vị trí $i: 1 \rightarrow n-1$:

Duyệt từ vị trí $j: i \rightarrow 0$:

Nếu $a[j-1] > a[i]$:

Tịnh tiến phần tử $a[j-1]$ lên $a[j]$ (shifting)

Gán $a[j] = a[i]$ (vị trí tìm được trong sorted part là j)

Xét ví dụ: Cho mảng gồm 6 phần tử như sau. Chú thích:

a	Phần tử được sắp xếp đúng vị trí
	Phần tử đang xét

index	0	1	2	3	4	5
value	4	3	2	10	12	1
Sorted part: 4 Unsorted part: 3,2,10,12,1 → $3 < 4$ → chuyển 3 về index 0						
index	0	1	2	3	4	5
value	3	4	2	10	12	1
Sorted part: 3,4 Unsorted part: 2,10,12,1 → $2 < 4$, chuyển 2 về index 1 $2 < 3$, chuyển 2 về index 0						

index	0	1	2	3	4	5
value	2	3	4	10	12	1
Sorted part: 2,3,4 Unsorted part: 10,12,1 → 10 ở đúng vị trí						
index	0	1	2	3	4	5
value	2	3	4	10	12	1
Sorted part: 2,3,4 Unsorted part: 10,12,1 → 10 ở đúng vị trí						
index	0	1	2	3	4	5
value	2	3	4	10	12	1
Sorted part: 2,3,4,10 Unsorted part: 12,1 → 12 ở đúng vị trí						
index	0	1	2	3	4	5
value	2	3	4	10	12	1
Sorted part: 2,3,4,10,12 Unsorted part: 1 → 1<12, chuyển 1 về index 4 1<10, chuyển 1 về index 3 1<4, chuyển 1 về index 2 1<3, chuyển 1 về index 1 1<2, chuyển 1 về index 0						
index	0	1	2	3	4	5
value	1	2	3	4	10	12
Mảng đã được sắp xếp						

3. Đánh giá thuật toán

- Độ phức tạp thời gian:
 - + Trường hợp tốt nhất: $O(n)$. Đạt được khi đầu vào là mảng đã sắp xếp tăng dần
 - + Trường hợp trung bình: $O(n^2)$
 - + Trường hợp xấu nhất: $O(n^2)$. Đạt được khi đầu vào là mảng đã sắp xếp ngược (giảm dần)
- Độ phức tạp không gian: $O(1)$ do không dùng mảng phụ → đây là thuật toán sắp xếp tại chỗ

4. Ưu, nhược điểm

- Ưu điểm:
 - + Với mảng có kích thước nhỏ phù hợp, insertion sort có thể thực hiện sắp xếp với tốc độ tương đối ổn
 - + So với 1 số thuật toán sắp xếp khác (merge sort,..) thì insertion sort không cần dùng thêm mảng phụ để lưu trữ
 - + Mảng đã cho là mảng đã sắp xếp tăng dần thì chỉ tốn n bước (tuyến tính)
- Nhược điểm:
 - + Với mảng có kích thước lớn, insertion sort thực hiện sắp xếp rất lâu vì cần n^2 bước cho n phần tử

III. Binary-Insertion sort

1. Ý tưởng: đây là bản nâng cấp của Insertion sort

- Ý tưởng tương tự insertion sort, khác ở chỗ tại bước tìm vị trí trong sorted part cho phần tử đang duyệt, thay vì so sánh tuyến tính nó với các phần tử trong sorted part, ta dùng tìm kiếm nhị phân (binary search) để tìm nhanh vị trí & chèn phần tử đang duyệt vào vị trí đó trong sorted part trong $O(\log k)$, với k : số lượng phần tử trong sorted part

2. Thuật toán từng bước

Duyệt mảng từ vị trí $i: 1 \rightarrow n-1$:

```
Khởi tạo rightBoundary = i - 1; //giữ cận phải của phần mảng đã sắp xếp
current = a[i]; //giữ giá trị phần tử htaị
//tìm vị trí để chèn current trong phần mảng đã sắp xếp
/* do binary_search chỉ làm việc với mảng sắp xếp nên cần 2 biến vị trí start & end để giữ vị trí đầu - cuối của phần mảng đã sắp xếp */
Khởi tạo index = binary_search(a[], current, start = 0, end = rightBoundary)
//nếu tìm thấy idx cần chèn → tịnh tiến (shift) toàn bộ phần tử sau idx lên 1
while (idx <= rightBoundary):
    a[rightBoundary + 1] = a[rightBoundary];
    rightBoundary--;
//chèn current vào vị trí idx
a[idx] = current
```

Xét ví dụ: Cho mảng gồm 6 phần tử như sau. Chú thích:

a	Phần tử được sắp xếp đúng vị trí
	Phần tử đang xét

index	0	1	2	3	4	5
value	4	3	2	10	12	1
Sorted part: 4 Unsorted part: 3,2,10,12,1 → binary_search(sorted_part, 3) = 0						
index	0	1	2	3	4	5
value	3	4	2	10	12	1
Sorted part: 3,4 Unsorted part: 2,10,12,1 → binary_search(sorted_part, 2) = 0						
index	0	1	2	3	4	5
value	2	3	4	10	12	1
Sorted part: 2,3,4 Unsorted part: 10,12,1 → 10 ở đúng vị trí						
index	0	1	2	3	4	5
value	2	3	4	10	12	1
Sorted part: 2,3,4 Unsorted part: 10,12,1 → binary_search(sorted_part, 10) = 3						
index	0	1	2	3	4	5

value	2	3	4	10	12	1
Sorted part: 2,3,4,10 Unsorted part: 12,1 → binary_search(sorted_part, 12) = 4						
index	0	1	2	3	4	5
value	2	3	4	10	12	1
Sorted part: 2,3,4,10,12 Unsorted part: 1 → binary_search(sorted_part, 1) = 0						
index	0	1	2	3	4	5
value	1	2	3	4	10	12
Mảng đã được sắp xếp						

3. Đánh giá thuật toán

- Độ phức tạp thời gian:
 - + Trường hợp tốt nhất: $O(n)$
 - + Trường hợp trung bình: $O(n \log n)$ vì duyệt $n-1$ phần tử, với mỗi phần tử cần tốn $O(\log n)$ để tìm vị trí chính xác trong sorted part
 - + Trường hợp tệ nhất: $O(n^2)$
- Độ phức tạp không gian: $O(1)$

4. Ưu, nhược điểm

- Ưu điểm:
 - + Với mảng có kích thước lớn thì giờ đây không mất quá lâu để sort vì chỉ tốn $O(n \log n)$ (kể cả trường hợp xấu nhất)

IV. Bubble sort

1. Ý tưởng

- Với mỗi phần tử có index i , ta so sánh lần lượt với các phần tử sau có index từ $[i+1..n-1]$, nếu phần tử đang xét & phần tử sau sai thứ tự thì đổi chỗ chúng. Như vậy, phần tử bé nhất sẽ “nổi” lên trên (tức lên đầu mảng)

2. Thuật toán từng bước

Xét ví dụ: Cho mảng gồm 5 phần tử như sau. Chú thích:

a, b	2 phần tử sau khi hoán vị
	Phần tử đang “trụ”
a	Phần tử so sánh với phần tử “trụ”

	Index	0	1	2	3	4	
(current index, next index) = (0,1)	value	5	1	4	2	8	Swap(a[0], a[1])
(current index, next index) = (0,2)		1	5	4	2	8	
(current index, next index) = (0,3)		1	5	4	2	8	

(current index, next index) = (0,4)		1	5	4	2	8	
	Index	0	1	2	3	4	
(current index, next index) = (1,2)	value	1	5	4	2	8	Swap(a[1],a[2])
(current index, next index) = (1,3)		1	4	5	2	8	Swap(a[1],a[3])
(current index, next index) = (1,4)		1	2	5	4	8	
	Index	0	1	2	3	4	
(current index, next index) = (2,3)	value	1	2	5	4	8	Swap(a[2],a[3])
(current index, next index) = (2,4)		1	2	4	5	8	
	Index	0	1	2	3	4	
(current index, next index) = (3,4)	value	1	2	4	5	8	
current index = 4 → kết thúc	Index	0	1	2	3	4	
	value	1	2	4	5	8	
Mảng đã được sắp xếp: [1,2,4,5,8]							

3. Đánh giá thuật toán

- Độ phức tạp thời gian:
 - + Trường hợp tốt nhất: $O(n)$. Đạt được khi đầu vào là mảng đã được sắp xếp tăng dần
 - + Trường hợp trung bình: $O(n^2)$
 - + Trường hợp tệ nhất: $O(n^2)$
- Độ phức tạp không gian: $O(1)$

4. Ưu, nhược điểm

- Ưu điểm:
 - + Rất dễ cài đặt
 - + Không cần mảng phụ để lưu trữ
 - + Chạy tốt khi đầu vào là mảng có kích thước nhỏ
- Nhược điểm:
 - + Chạy rất lâu với mảng đầu vào có kích thước lớn

V. Shaker sort

1. Ý tưởng: là bản nâng cấp của bubble sort

- Dựa trên nguyên tắc đổi chỗ trực tiếp của bubble sort, nhưng thay vì mỗi lần duyệt cố gắng “nổi bọt” phần tử bé lên đầu mảng, shaker sort sẽ cố gắng để vừa đẩy phần tử bé về

đầu mảng, vừa đẩy phần tử lớn về cuối mảng → đúng với tinh thần “lắc qua lắc lại” (shaker)

2. Thuật toán từng bước

- Bước 1: khởi tạo $left = 0$, $right = n-1$,
 $newStart = 0$ (vị trí bắt đầu duyệt mới)
- Bước 2:
 While($left < right$):
 //lượt 1: “lắc” từ trái qua phải
 Lặp từ vị trí i : $left \rightarrow right-1$:
 Nếu $a[i] > a[i+1]$:
 Hoán vị $a[i], a[i+1]$
 $newStart = i$
 //lượt 2: “lắc” từ phải qua trái
 $right = newStart$
 Lặp từ vị trí i : $right \rightarrow left+1$:
 Nếu $a[i] < a[i-1]$:
 Hoán vị $a[i], a[i-1]$
 $newStart = i$
 $left = newStart$

Xét ví dụ: Cho mảng gồm 6 phần tử như sau. Chú thích:

a, b	2 phần tử sau khi hoán vị
a	index chỗ đó cố định (không thay đổi được)

Từ trái → phải để “nổi” 12 về cuối mảng	Index	0	1	2	3	4	5	
	Value	12	2	8	5	1	6	Swap($a[0], a[1]$)
	Index	0	1	2	3	4	5	
	Value	2	12	8	5	1	6	Swap($a[1], a[2]$)
	Index	0	1	2	3	4	5	
	Value	2	8	12	5	1	6	Swap($a[2], a[3]$)
	Index	0	1	2	3	4	5	
	Value	2	8	5	12	1	6	Swap($a[3], a[4]$)
	Index	0	1	2	3	4	5	
	Value	2	8	5	1	12	6	Swap($a[4], a[5]$)
	Index	0	1	2	3	4	5	
	Value	2	8	5	1	6	12	
Index 5 cố định !								

Từ phải → trái để “nổi” 1 về đầu mảng	Index	0	1	2	3	4	5	
	Value	2	8	5	1	6	12	
	Index	0	1	2	3	4	5	
	Value	2	8	5	1	6	12	Swap(a[2],a[3])
	Index	0	1	2	3	4	5	
	Value	2	8	1	5	6	12	Swap(a[1],a[2])
	Index	0	1	2	3	4	5	
	Value	2	1	8	5	6	12	Swap(a[0],a[1])
	Index	0	1	2	3	4	5	
	Value	1	2	8	5	6	12	
Index 0 cố định !								
Trái → phải để “nổi” 8 về kế cuối mảng	Index	0	1	2	3	4	5	
	Value	1	2	8	5	6	12	
	Index	0	1	2	3	4	5	
	Value	1	2	8	5	6	12	Swap(a[2],a[3])
	Index	0	1	2	3	4	5	
	Value	1	2	5	8	6	12	Swap(a[3],a[4])
	Index	0	1	2	3	4	5	
	Value	1	2	5	6	8	12	
Index 4 cố định !								
Phải → trái để “nổi” 2 về kế đầu mảng	Index	0	1	2	3	4	5	
	Value	1	2	5	6	8	12	
	Index	0	1	2	3	4	5	
	Value	1	2	5	6	8	12	
Index 1 cố định !								
Trái → phải để nổi 6 về phía cuối mảng	Index	0	1	2	3	4	5	
	Value	1	2	5	6	8	12	
Index 2, 3 cố định !								
Còn lại mỗi index 2 đã đứng đúng vị trí	Index	0	1	2	3	4	5	
	Value	1	2	5	6	8	12	
Mảng đã được sắp xếp								

3. Đánh giá thuật toán

- Độ phức tạp thời gian:
- + Trường hợp tốt nhất: $O(n)$

- + Trường hợp trung bình: $O(n^2)$
- + Trường hợp xấu nhất: $O(n^2)$
- Độ phức tạp không gian: $O(1)$
- 4. **Ưu, nhược điểm**
 - Ưu điểm:
 - + Shaker sort chiếm ưu thế hơn bubble sort trong trường hợp các phần tử trong mảng gần như có thứ tự tăng dần (nghĩa là giảm phép so sánh thừa trong bubble sort). VD: [2,3,4,5,1] → shaker sort chỉ cần 1 lần duyệt, bubble sort cần 4 lần duyệt
 - Nhược điểm:
 - + Trong trường hợp mảng có ngẫu nhiên phần tử với thứ tự đảo lộn thì Bubble Sort và Shaker Sort cho thời gian sắp xếp gần tương đương nhau
 - + Với mảng đầu vào có kích thước lớn thì bubble sort hay shaker sort đều xử lý khá chậm và tốc độ xử lý tương đương nhau

VI. Shell sort

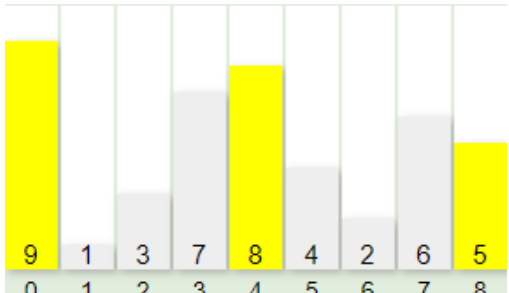
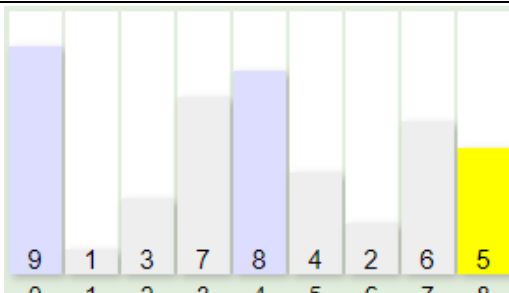
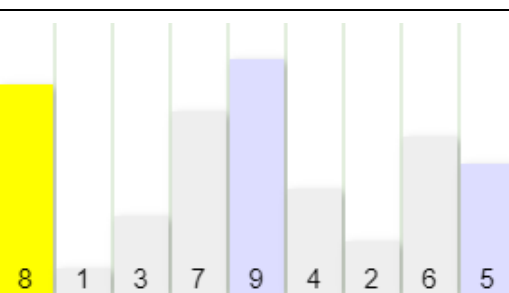
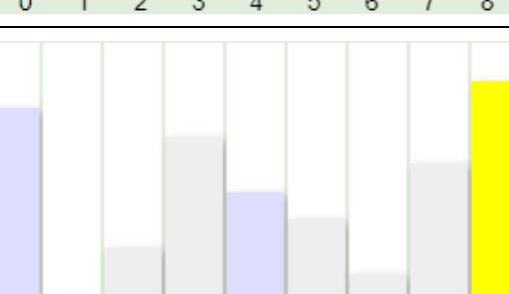
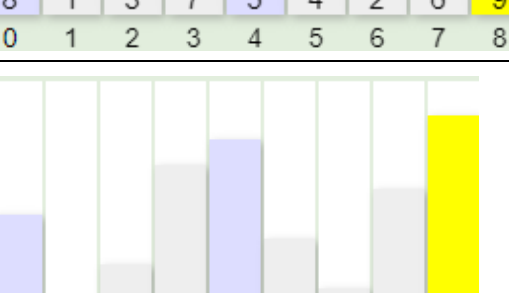
1. **Ý tưởng:** đây là thuật toán được dựa trên thuật toán sắp xếp chèn (insertion sort) & sắp xếp chọn (selection sort)
 - Đầu tiên, thuật toán này sử dụng thuật toán sắp xếp chọn trên các phần tử có khoảng cách (gap) xa nhau, sau đó sắp xếp các phần tử có khoảng cách hẹp hơn cho tới khi khoảng cách giữa 2 phần tử là 0 (nghĩa là chính phần tử đó)
2. **Thuật toán từng bước**

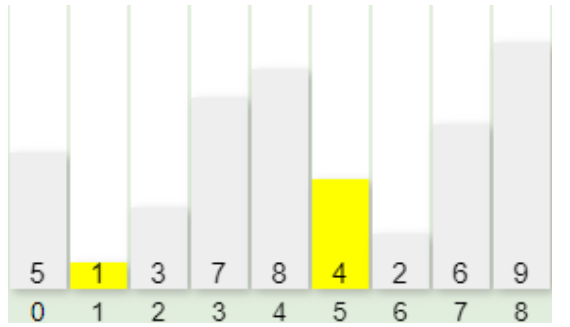
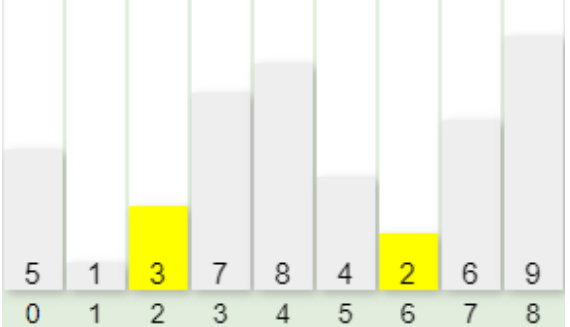
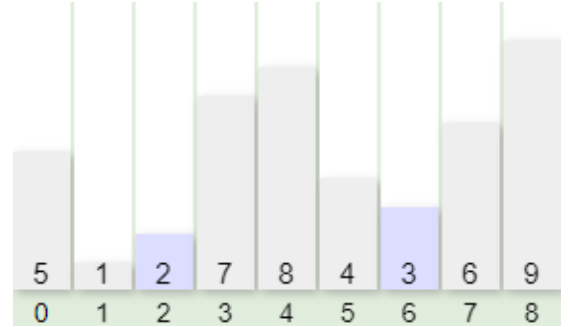
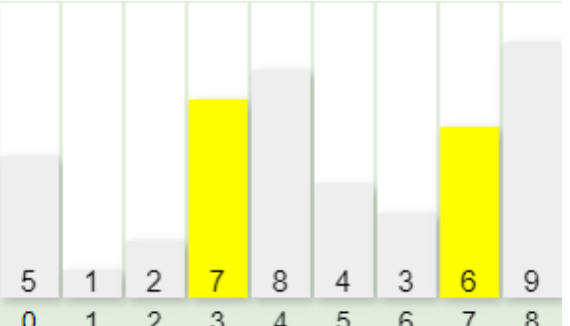
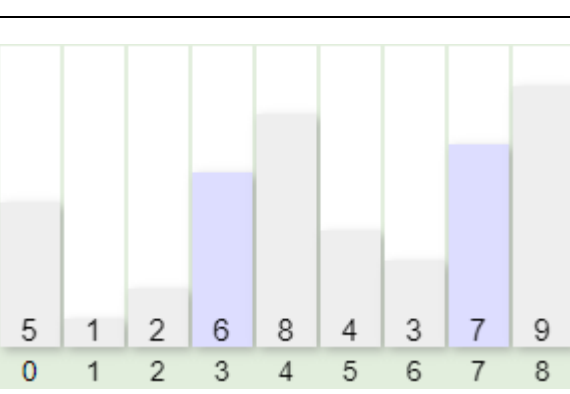
```
//duyet gap, mỗi lần duyệt gap giảm 2 lần so với trước đó
For(gap:  $\frac{n}{2} \rightarrow 0, \text{gap}/=2$ )
    //duyet từng phần tử
    For(i: gap  $\rightarrow$  n-1, i++)
        //so sánh a[i] với phần tử cách nó i-gap về phía trái, a[i-gap]
        Khởi tạo j, temp = a[i]
        For(j: i  $\rightarrow$  gap, j-=gap)
            Nếu a[j - gap] > temp
                a[j] = a[j-gap]
        //j bây giờ là vị trí đúng của temp
        a[j] = temp
```

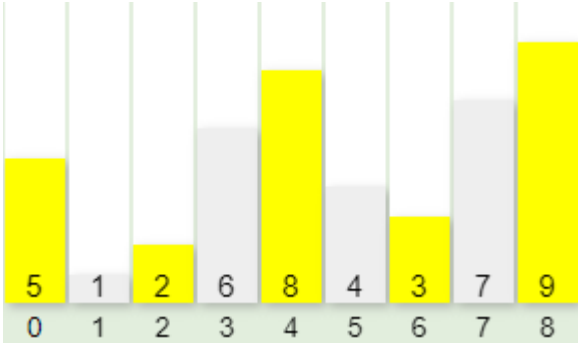
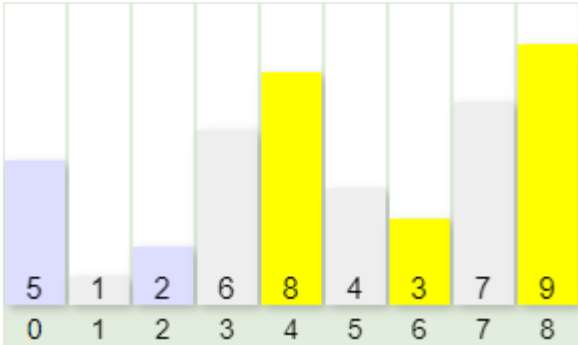
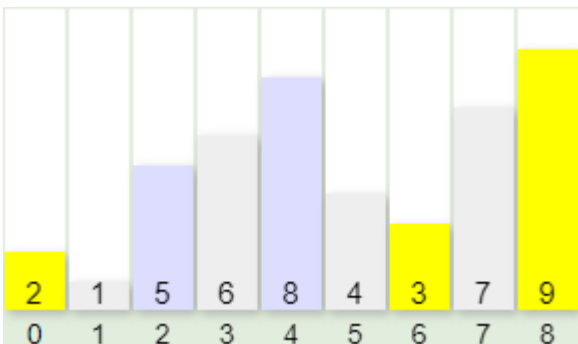
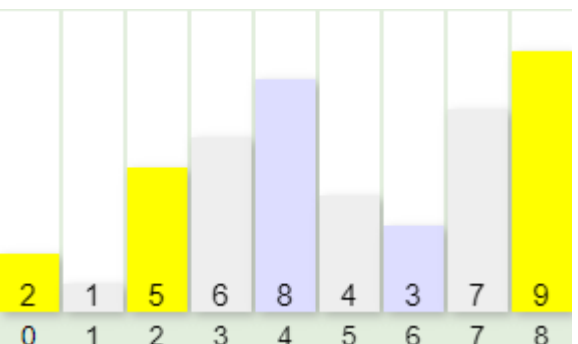
Xét ví dụ: Cho mảng gồm 5 phần tử như sau: [9, 1, 3, 7, 8, 4, 2, 6, 5] →

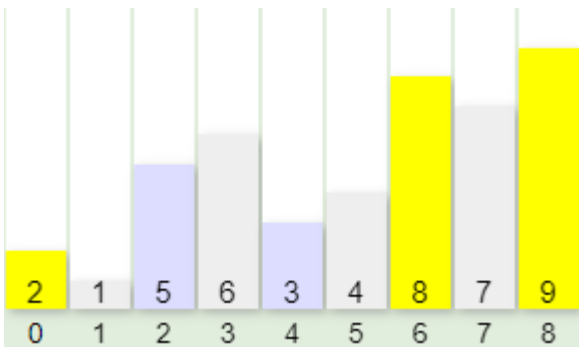
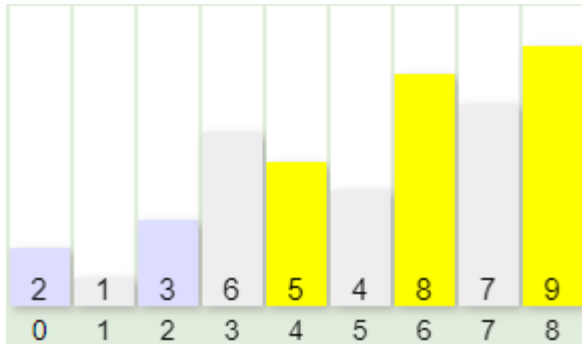
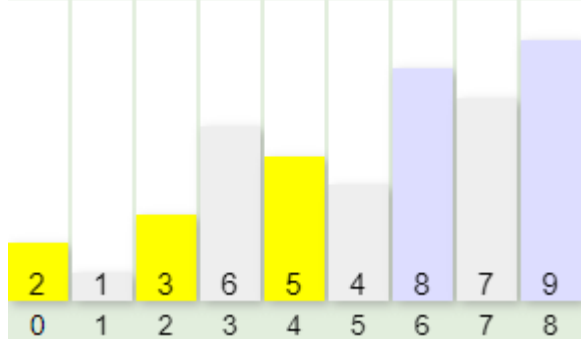
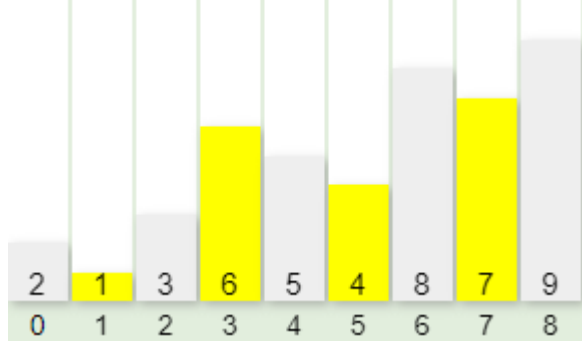
n = 9. Chú thích:

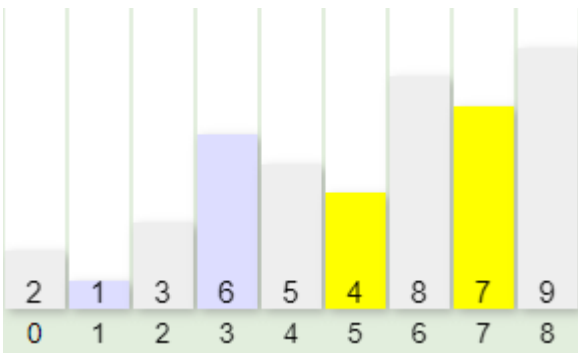
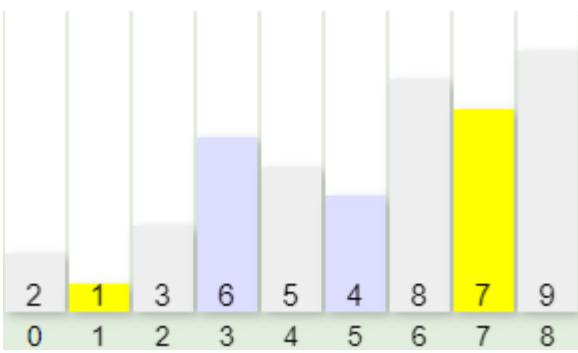
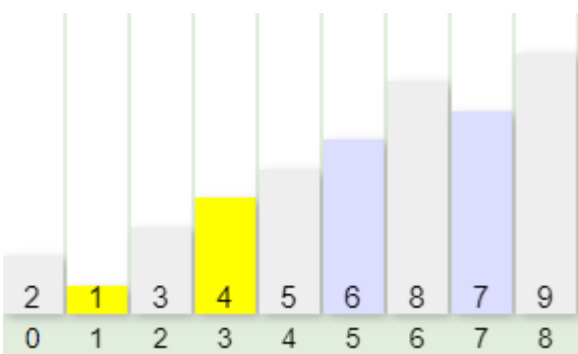
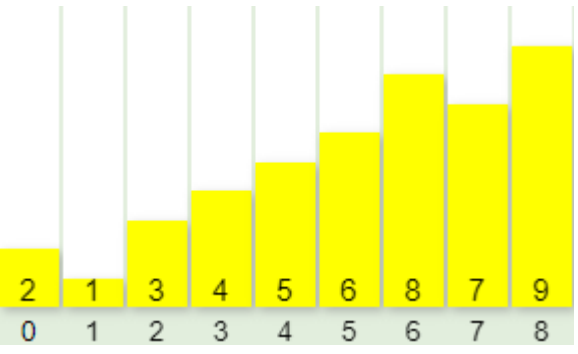
- + Màu tím: 2 phần tử đang được so sánh
- + Màu vàng: thể hiện phần tử đang chờ được so sánh hoặc thể hiện các phần tử có chung gap

Gap	Phần tử đang xét	Array	Mô tả
$n/2 = 4$	a[0]		Bắt đầu tại phần tử đầu tiên, đánh dấu mỗi phần tử cách nhau 4 đơn vị (cột vàng)
			Cặp đầu tiên là (a[0],a[4]), thấy $a[0] > a[4]$ nên swap(a[0],a[4])
			Cặp tiếp theo là (a[4],a[8]), do $a[4] > a[8] \rightarrow$ swap(a[4],a[8])
			Do giá trị ở vị trí 4 đã bị thay thế, mà vị trí 4 có thứ tự liên quan tới vị trí 0 nên kiểm tra a[0],a[4]. Do $a[0] > a[4]$ nên swap(a[0],a[4])
			Đã xong a[0]

a[1]		<p>Bắt đầu tại phần tử a[1]), phần tử cách nó 4 đơn vị là a[5]=4. 2 phần tử này đã ở đúng vị trí → không cần hoán vị</p>
a[2]		<p>Bắt đầu tại phần tử a[2], phần tử cách nó 4 đơn vị là a[6]=2. Vì a[6]<a[2] → swap(a[2],a[6])</p>
		<p>Đã xong a[2]</p>
a[3]		<p>Bắt đầu tại phần tử a[3], phần tử cách nó 4 đơn vị là a[7]=6. Vì a[7]<a[3] → swap(a[3],a[7])</p>
		<p>Đã xong a[3]</p>

$n/2^2 = 2$	a[0]		<p>Bắt đầu tại phần tử đầu tiên, đánh dấu mỗi phần tử cách nhau 2 đơn vị lần lượt là a[2], a[4], a[6], a[8]. Tiến hành so sánh a[0] với chúng</p>
			<p>Do $a[2] < a[0] \rightarrow \text{swap}(a[0], a[2])$</p>
			<p>a[2] & a[4] ở đúng vị trí nên không cần hoán vị</p>
			<p>Do $a[6] < a[4] \rightarrow \text{swap}(a[4], a[6])$</p>

			<p>Vì thay đổi $a[4]$ có thể dẫn tới thay đổi $a[2]$ nên cần quay lui để kiểm tra cặp $a[2], a[4]$. Do $a[4] < a[2] \rightarrow \text{swap}(a[2], a[4])$</p>
			<p>Vì thay đổi $a[2]$ có thể dẫn tới thay đổi $a[0]$ nên cần quay lui để kiểm tra $a[2], a[0]$. Thấy rằng chúng ở đúng vị trí nên không cần hoán vị</p>
			<p>Do bước trên so sánh $a[6], a[4]$ và quay lui để cập nhật giá trị. Sau khi cập nhật lại các giá trị thì so sánh gốc tại 2 phần tử $a[6], a[4]$ kết thúc. Tiếp tục so sánh 2 phần tử $a[8], a[9]$. Do chúng ở đúng vị trí nên không cần hoán vị</p>
	$a[1]$		<p>Bắt đầu tại phần tử $a[1]$, đánh dấu phần tử cách nhau 2 đơn vị lần lượt là $a[3], a[5], a[7]$. Tiến hành so sánh $a[1]$ với chúng</p>

		 <p>So sánh $a[1], a[3] \rightarrow$ chúng ở đúng vị trí</p>	
		 <p>So sánh $a[3], a[5]$. Do $a[5] < a[3] \rightarrow \text{swap}(a[3], a[5]) +$ quay lui về tới $a[1]$ để cập nhật lại giá trị</p>	
		 <p>Sau khi cập nhật giá trị ($a[1]=1$, $a[3]=4$, $a[5]=6$), so sánh tiếp $a[5], a[7] \rightarrow$ chúng ở đúng vị trí</p>	
$n/2^3 = 1$	$a[0]$	 <p>Bắt đầu tại phần tử đầu tiên, đánh dấu phần tử cách nhau 1 đơn vị lần lượt là $a[1]$, $a[2]$, $a[3]$, ..., $a[8]$. Tiến hành so sánh $a[0]$ với chúng</p>	

			Do $a[0] > a[1] \rightarrow \text{swap}(a[0], a[1])$
			So sánh $a[2], a[3] \rightarrow$ đúng thứ tự
	.	(thực hiện so sánh từng cặp $a[i], a[i+1]$)	.
	.	.	.
	.	.	.
	a[7]		So sánh $a[7], a[8] \rightarrow$ đúng thứ tự \rightarrow không cần hoán vị
Mảng đã được sắp xếp !			

- Tham khảo đồ thị: <https://www.w3resource.com/ODSA/AV/Sorting/shellsortAV.html>

3. Đánh giá thuật toán

- Độ phức tạp thời gian:
 - + Trường hợp tốt nhất: $O(n \log n)$. Đạt được khi mảng đầu vào đã được sắp xếp tăng dần
 - + Trường hợp trung bình: $\sim O(n^{1.25}) = O(n \log n)$
 - + Trường hợp tệ nhất: $O(n^2)$
- Độ phức tạp không gian: $O(1)$

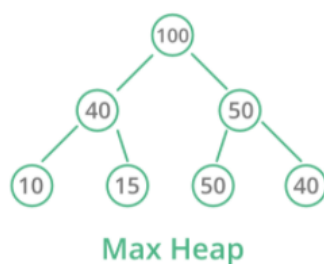
4. Ưu, nhược điểm

- Ưu điểm:
 - + Thích hợp với mảng đầu vào có kích thước trung bình (medium-size arrays)
- Nhược điểm:
 - + Đây là 1 thuật toán phức tạp, ở phần trên gap được mặc định là chia đôi sau mỗi lần lặp, thực tế gap có thể thay đổi tùy vào lập trình viên, và do đó độ phức tạp thời gian cũng thay đổi theo gap
 - + Chạy chậm hơn merge sort, quick sort & heap sort

VII. Heap sort

1. Ý tưởng

- Heap sort là một ứng dụng của cấu trúc dữ liệu heap. Cụ thể nó sử dụng tính chất của 1 max-heap/min-heap (bài báo cáo sử dụng max-heap) để tạo nên 1 heap hoàn chỉnh, sau đó sắp xếp bằng việc đổi chỗ node rễ (root) và node cuối cùng & max-heap cho mảng mới đó lần nữa.
 - + Tính chất của max-heap: node gốc trong max heap là node có giá trị lớn nhất. VD:



- Có thể hình dung heap dưới dạng cây để thực hiện thao tác trên mảng

2. Thuật toán từng bước

*Cách tạo 1 max heap (Heapify): lặp từ nửa đầu mảng ($\lfloor \frac{n}{2} \rfloor .. 0$) (vì trong max-heap, node cha luôn nằm ở nửa mảng đầu), với mỗi node (index i) so sánh với 2 node con ($2i+1$ & $2i+2$). Nếu ít nhất 1 node con lớn hơn node cha \rightarrow swap 2 cha con & gọi đệ qui để heapify cho “node con mới đổi” trở xuống

*Thuật toán heap sort:

- Bước 1: tạo ra max heap từ mảng đầu vào (heap construction)

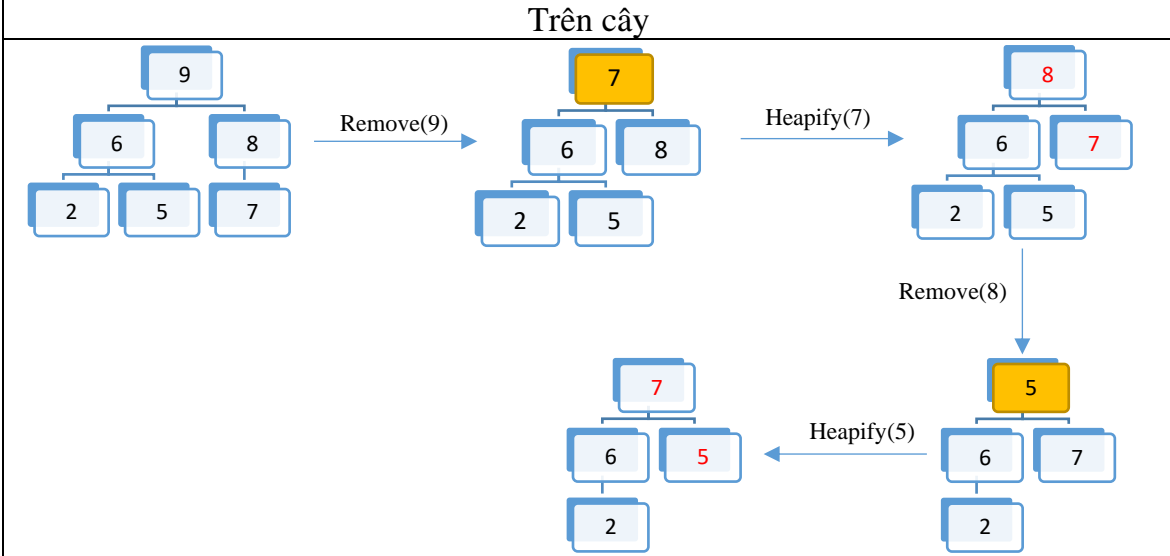
- Bước 2: Lặp lại bước này n-1 lần
 - + Hình dung trên cây: thay thế node rễ (root node) với node cuối cùng & xóa node cuối cùng
 - + Trên mảng: swap phần tử đầu tiên với phần tử cuối cùng
 - + Kích thước của heap (cả cây & mảng giảm 1)
 - + Xây lại heap từ node rễ (hay a[0]), nghĩa là Heapify(node_rễ)

Xét ví dụ: Cho mảng gồm 6 phần tử như sau: [2,9,7,6,5,8]. Chú thích

	node cha	a	b	2 phần tử cần swap ở bước 2
	node con tương ứng			node cố định (sau khi “xóa” node đó trong cây)
a,b:	2 phần tử sau khi hoán vị			

Tạo heap							Biểu diễn cây (kết quả cuối cùng)
	Trên mảng					Mô tả	
Index	0	1	2	3	4	5	
Value	2	9	7	6	5	8	
	2	9	8	6	5	7	
	2	9	8	6	5	7	
	9	2	8	6	5	7	
	9	6	8	2	5	7	

Xóa phần tử đầu



Trên mảng					
9	6	8	2	5	7
7	6	8	2	5	9
8	6	7	2	5	9
8	6	7	2	5	9
5	6	7	2	8	9
7	6	5	2	8	9
7	6	5	2	8	9
2	6	5	7	8	9
6	2	5	7	8	9
6	2	5	7	8	9
5	2	6	7	8	9
5	2	6	7	8	9
2	5	6	7	8	9
2	5	6	7	8	9

3. Đánh giá thuật toán

- Độ phức tạp thời gian: $O(n \log n)$ cho mọi trường hợp
- Độ phức tạp không gian: $O(1)$

4. Ưu, nhược điểm

- Ưu điểm:
 - + Không sử dụng mảng phụ để lưu trữ mã vẫn chạy trong $O(n \log n)$, trong khi merge sort cần sử dụng không gian phụ để giảm thời gian chạy còn $O(n \log n)$ & quick sort cần nhiều stack

3. Đánh giá thuật toán

- Độ phức tạp thời gian: $O(n \log n)$ cho mọi trường hợp
- Độ phức tạp không gian: $O(n)$ do sử dụng mảng phụ để lưu nửa đầu & nửa sau của mảng cần chia ban đầu

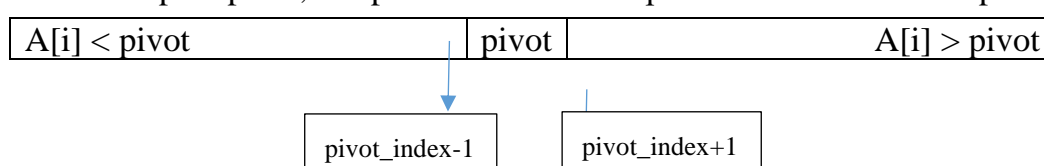
4. Ưu, nhược điểm

- Ưu điểm:
 - + Do có độ phức tạp thời gian $O(n \log n)$ nên có thể được áp dụng để sắp xếp cho mảng với kích thước bất kỳ
 - + Tốt hơn nhóm thuật toán $O(n^2)$
 - + Là thuật toán stable
- Nhược điểm:
 - + Cần thêm mảng phụ để lưu trữ, kích thước mảng phụ có thể lên tới khoảng bằng n

IX. Quick sort

1. Ý tưởng

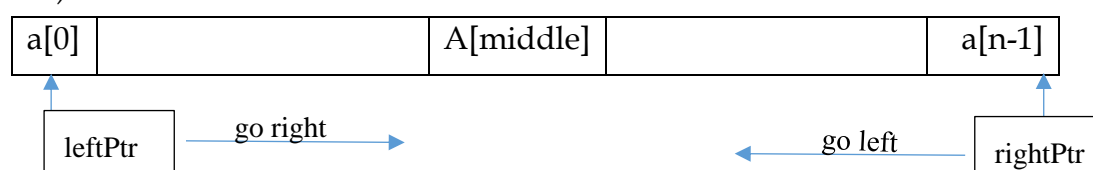
- Đây là thuật toán chia để trị: chọn 1 điểm chốt (pivot) để các phần tử nào lớn hơn pivot sẽ đưa về bên phải pivot, các phần tử nào bé hơn pivot sẽ đưa về bên trái pivot.



Sau khi đưa về đúng kiểu như hình trên thì tiếp tục chia mảng thành 2 phần, 1 phần có index $[0..\text{pivot_index}-1]$, phần còn lại có index $[\text{pivot_index}+1..n-1]$

2. Thuật toán từng bước

- Bước 1: Chọn pivot: Trong báo cáo này, pivot của mảng sẽ là phần tử ở giữa
- Bước 2: Khởi tạo 2 con trỏ, leftPtr & rightPtr, đặt ở 2 cực của mảng (index 0 & $n-1$)



- Bước 3: Với leftPtr, cho con trỏ này chạy về bên phải để tìm phần tử đầu tiên $\geq \text{pivot}$ (gọi phần tử do leftPtr tìm được là x)
- Bước 4: Với rightPtr, cho con trỏ này chạy về bên trái để tìm phần tử đầu tiên lớn $< \text{pivot}$ (gọi phần tử do rightPtr tìm được là y)
- Bước 5: Swap(x, y) để x nằm bên trái pivot, y nằm bên phải pivot (đúng với ý tưởng quick sort)

- **Bước 6:** Lặp lại các bước trên tới khi leftPtr & rightPtr trùng hoặc bước qua nhau (rightPtr <= leftPtr)
- **Bước 7:** Mảng ban đầu đã đúng ý tưởng quick sort, chia mảng đó thành 2 phần, 1 phần [0..rightPtr], phần kia [leftPtr+1..n-1]

Xét ví dụ: Cho mảng gồm 6 phần tử như sau. Chú thích:

leftPtr	a	pivot
rightPtr	a, b	2 số đã hoán vị

Mô tả											Mô tả
leftPtr>rightPtr → swap		3	1	2	6	9	0				
		0	1	2	6	9	3				
		0	1	2	6	9	3				
leftPtr trùng rightPtr		0	1	2	6	9	3				
leftPtr & rightPtr đi qua nhau		0	1	2	6	9	3				
		0	1			2	6	9	3		
leftPtr trùng rightPtr		0	1			2	6	9	3		leftPtr>rightPtr → swap
leftPtr & rightPtr đi qua nhau	NUL	0	1			2	3	9	6		leftPtr trùng rightPtr
	<Mảng đã sắp xếp>					2	3	9	6		leftPtr & rightPtr đi qua nhau
			2	3			9	6			
leftPtr trùng rightPtr			2	3			9	6	9		leftPtr trùng rightPtr
leftPtr & rightPtr đi qua nhau		NUL	2	3			NUL	6	9		leftPtr & rightPtr đi qua nhau
	<Mảng đã sắp xếp>					<Mảng đã sắp xếp>					
Sau khi các phụ được sắp xếp → mảng ban đầu sẽ cũng sẽ được sắp xếp theo (in-place sorting)											
		0	1	2	3	6	9				

3. Đánh giá thuật toán

- Độ phức tạp thời gian:
 - + Trường hợp tốt nhất: $O(n \log n)$. Đạt được khi pivot được chọn là phần tử trung vị
 - + Trường hợp trung bình: $O(n \log n)$
 - + Trường hợp tệ nhất: $O(n^2)$. Xảy ra khi pivot được chọn là phần tử lớn nhất/bé nhất trong mảng
- Độ phức tạp không gian: $O(1)$

4. Ưu, nhược điểm

- Ưu điểm:
 - + Thường sử dụng để xử lý mảng đầu vào với kích thước lớn
 - + Không tốn thêm không gian lưu trữ như merge sort
- Nhược điểm:
 - + Nếu rơi vào trường hợp tệ nhất, quick sort chạy chỉ nhanh tương đương với trường hợp trung bình của bubble, insertion hay selection sort
 - + Nếu mảng đầu vào đã được sắp xếp thì bubble sort vẫn chạy tốt hơn quick sort

X. Counting sort

1. Ý tưởng

- Đây là thuật toán sắp xếp không dựa vào phép so sánh như 9 thuật toán sắp xếp phía trên. Bằng việc đếm số lần xuất hiện của từng phần tử trong mảng cần sắp xếp $a[]$ & lưu kết quả được lưu vào mảng phụ (kích thước mảng phụ là phần tử lớn nhất trong $a[]$). Sau đó từ mảng phụ & mảng $a[]$ ban đầu (unsorted) sắp xếp thành công mảng $a[]$ (sorted)

2. Thuật toán từng bước

- Bước 1: Đếm số lần xuất hiện của từng phần tử trong mảng cần sắp xếp $a[]$. Kết quả được lưu vào mảng $count[]$ (có kích thước bằng phần tử lớn nhất của $a[]$)
- Bước 2: cập nhật mảng $count[]$. Lần này, $count[i]$ (trừ $count[0]$) thể hiện giới hạn chỉ số của phần tử i sau khi sắp xếp

$$count[i] = count[i-1] + 1$$

- Bước 3: Duyệt ngược từ $a[]$ (để bảo toàn tính stable(*)) và kết hợp mảng $count[]$ ở bước 2 để ở mỗi lần duyệt, $result[--count[a[i]]]$ sẽ là vị trí đúng của $a[i]$. Sau bước này, mảng đã được sắp xếp

(*): có 2 phần tử trùng nhau, lần lượt ở vị trí a_1 & a_2 ($a_1 < a_2$). Sau khi mảng đã sắp xếp, vị trí của chúng lần lượt là b_1 & b_2 và $b_1 < b_2$ (vẫn đúng thứ tự trước sau). Nếu $b_1 > b_2$ thì không thỏa mãn tính stable

Xét ví dụ: Cho mảng gồm 7 phần tử như sau. Chú thích:

	Ô chưa điền số	a	Sau khi cập nhật $count[i]$
	Ô điền số		

XÂY DỰNG & CẬP NHẬT MẢNG $count[]$

a[]	3	4	2	1	0	0	4
-----	---	---	---	---	---	---	---

count[]	Index	0	1	2	3	4
	Value	2	1	1	1	2
Cập nhật count[]		2	3	4	5	7

XÂY DỰNG MẢNG res[]

-----PASS 1-----

Duyệt ngược a[]	3	4	2	1	0	0	4
-----------------	---	---	---	---	---	---	---

count[]	0	1	2	3	4
	2	3	4	5	7 → 6

result[]	0	1	2	3	4	5	6
							4

-----PASS 2-----

Duyệt ngược a[]	3	4	2	1	0	0	4
-----------------	---	---	---	---	---	---	---

count[]	0	1	2	3	4
	2 → 1	3	4	5	6

result[]	0	1	2	3	4	5	6
		0					4

-----PASS 3-----

Duyệt ngược a[]	3	4	2	1	0	0	4
-----------------	---	---	---	---	---	---	---

count[]	0	1	2	3	4
	1 → 0	3	4	5	6

result[]	0	1	2	3	4	5	6
	0	0					4

-----PASS 4-----

Duyệt ngược a[]	3	4	2	1	0	0	4
-----------------	---	---	---	---	---	---	---

count[]	0	1	2	3	4
	0	3 → 2	4	5	6

result[]	0	1	2	3	4	5	6
	0	0	1				4

-----PASS 5-----

Duyệt ngược a[]	3	4	2	1	0	0	4
-----------------	---	---	---	---	---	---	---

count[]	0	1	2	3	4
	0	2	4 → 3	5	6

result[]	0	1	2	3	4	5	6
----------	---	---	---	---	---	---	---

	0	0	1	2			4
--	---	---	---	---	--	--	---

-----PASS 6-----

Duyệt ngược a[]	3	4	2	1	0	0	4
-----------------	---	---	---	---	---	---	---

count[]	0	1	2	3	4		
	0	2	3	5	6 → 5		

result[]	0	1	2	3	4	5	6
	0	0	1	2		4	4

-----PASS 7-----

Duyệt ngược a[]	3	4	2	1	0	0	4
-----------------	---	---	---	---	---	---	---

count[]	0	1	2	3	4		
	0	2	3	5 → 4	5		

result[]	0	1	2	3	4	5	6
	0	0	1	2	3	4	4

⇒ Mảng ban đầu đã được sắp xếp

3. Đánh giá thuật toán

- Độ phức tạp thời gian: $O(n)$ trong mọi trường hợp
- Độ phức tạp không gian: $O(n)$

4. Ưu, nhược điểm

- Ưu điểm:
 - + Chạy nhanh hơn các thuật toán sắp xếp có sử dụng so sánh như quick sort hay merge sort
 - + Dễ hiểu & dễ cài đặt hơn các thuật toán $O(n \log n)$
 - + Là 1 stable sort → thích hợp khi ta không muốn mất thứ tự của 2 giá trị bằng nhau trong mảng đầu vào
- Nhược điểm:
 - + Không xử lý được khi mảng có số âm
 - + Các phần tử không được cách nhau quá lớn (VD: [0, 999, 1111, 10000] → kích thước mảng count[] là 10000 phần tử) vì sẽ rất lãng phí bộ nhớ do kích thước mảng count[] được xác định bằng với phần tử lớn nhất trong a[]

XI. Radix sort

1. Ý tưởng

- Radix sort (sort theo cơ số), đúng như tên gọi, thuật toán sẽ áp dụng counting sort cho mỗi cơ số trong $a[i]$ (với $a[]$ là mảng đầu vào), chính vì thế mà radix sort cũng được coi là sắp xếp không dựa vào phép so sánh. Cụ thể, với mỗi cơ số, sẽ có 10 xô (10 buckets) được đánh số từ $0 \rightarrow 9$ (nghĩa là $count[]$ trong counting sort sẽ luôn có kích thước cố định). Tại mỗi cơ số của mỗi số, ta sẽ đẩy nó vào bucket có số tương ứng. Sau khi duyệt hết mảng với 1 cơ số bất kỳ, duyệt toàn bộ các bucket, và đẩy nó ra lần lượt theo trình tự

2. Thuật toán từng bước

- Bước 1: với mỗi hàng 10^x , duyệt từng phần tử với hàng tương ứng, đẩy phần tử đó vào bucket với chỉ số tương ứng, nếu trong bucket đã tồn tại (vài) phần tử thì phần tử mới thêm ngay sau đó
- Bước 2: mảng mới được tạo bằng việc rút lần lượt từng phần tử trong bucket $0 \rightarrow 9$.
- Bước 3: lặp lại 3 bước trên với hàng 10^{x+1} . Kết thúc lặp khi xét tới hàng cao nhất của số lớn nhất trong mảng

Xét ví dụ: Cho mảng gồm 7 phần tử như sau:

[0381, 0894, 0535, 1158, 0659, 0419, 0549]

Chữ số hàng đơn vị										
bucket	0	1	2	3	4	5	6	7	8	9
		381			894	535			1158	659
										419
										549
Mảng sau khi rút lần lượt từng phần tử trong bucket: [381, 894, 535, 1158, 659, 419, 549]										
Chữ số hàng chục										
bucket	0	1	2	3	4	5	6	7	8	9
		419		535	549	1158			381	894
						659				
Mảng sau khi rút lần lượt từng phần tử trong bucket: [419, 535, 549, 1158, 659, 381, 894]										
Chữ số hàng trăm										
bucket	0	1	2	3	4	5	6	7	8	9
		1158		381	419	535	659		894	
						549				
Mảng sau khi rút lần lượt từng phần tử trong bucket: [1158, 381, 419, 535, 549, 659, 894]										
Chữ số hàng ngàn										
bucket	0	1	2	3	4	5	6	7	8	9
	0381	1158								
	0419									
	0535									
	0549									
	0659									
	0894									

Mảng sau khi rút lần lượt từng phần tử trong bucket:
 [381, 419, 535, 549, 659, 984, 1158]
 → Mảng đã được sắp xếp

3. Đánh giá thuật toán

- Độ phức tạp thời gian: $O(d(n+b))$. Trong đó
 - + d: số lượng chữ số tối đa mà 1 số có thể có
 - + n: kích thước mảng đầu vào
 - + b: cơ số (trong ví dụ trên cơ số $b = 10$)
- Độ phức tạp không gian: $O(n)$

4. Ưu, nhược điểm

- Ưu điểm: chạy nhanh khi mảng đầu vào có các phần tử có giá trị gần (VD: $0 \leq a[i] \leq 100$)
- Nhược điểm:
 - + Chỉ thích hợp cho kiểu số nguyên (int)
 - + Khi các mảng đầu vào gần như có thứ tự tăng dần thì radix sort sẽ tiêu tốn nhiều thời gian hơn các thuật toán dùng so sánh
 - + Về mặt không gian, radix sort tốn nhiều không gian hơn quick sort

XII. Flash sort

1. Ý tưởng

- Dựa trên việc phân các phần tử về các lớp (classes) trong mảng. FlashSort bao gồm ba khối logic: Phân loại các phần tử (Elements Classification); Hoán vị các phần tử (Elements Permutation); sắp xếp các phần tử trong từng phân lớp theo đúng thứ tự (Elements Ordering)
- + Phân loại phần tử: để biết 1 phần tử thuộc phân lớp nào, sử dụng công thức

$$k[i] = 1 + \text{floor} \left[\frac{(m-1) * (a[i] - \text{minVal})}{\text{maxValue} - \text{minValue}} \right] (*)$$

- $k[i]$: cho biết phân lớp chứa phần tử i của mảng A , $k[i]$ nằm trong khoảng $[1, m]$
- $m = 0.45 * n$: số phân lớp cho 1 mảng

Lập 1 mảng $L[]$ lưu cận trên của mỗi class, cận trên được tìm bằng việc cộng dồn $L[i]$ với class trước đó, $L[i-1]$

+ Hoán vị các phần tử: để tìm chính xác vị trí của 1 phần tử trong 1 class, ta tính xem nó phải nằm ở phân lớp nào, rồi chen nó lần lượt từ phải qua trái của class, và vì ta bỏ vào phân lớp đó 1 phần tử nên phải lùi vị trí của phân lớp lại 1 đơn vị. Đối với những phần tử chưa ở đúng chỗ, ta tiến hành 1 chu trình khác để tìm 1 index thỏa $i < L[k[a[i]]]$

+ Sắp xếp các phần tử trong từng phân lớp theo đúng thứ tự: việc bây giờ là sắp xếp mỗi phân lớp thì sẽ sắp xếp được mảng → sử dụng insertion sort cho mỗi class

2. Thuật toán từng bước

- Bước 1: khởi tạo 2 mảng
 + class[n]: sử dụng công thức (*) để tính toán phân lớp cho phần tử a[i] và thêm kết quả vừa tính vào mảng class[]
 + sizeClass[m=0.42n]: đếm số lần xuất hiện của mỗi class[i] và chèn vào vị trí index = class[i] trong sizeClass[]. Sau đó, cập nhật lại mảng sizeClass[] để nó giữ cận trên (upper bound) của mỗi class[i]
- Bước 2: trong a[], thứ tự các class vẫn còn lộn xộn (do chưa lộn các phần tử về đúng vị trí) nên:

```
//duyet từng ptu trong a
for (int i = 0; i < n; i++) {
    //while (index của ptu đó chưa vượt quá upper bound trong class của nó)
    while (i < lenClass[classOf[i]]) {
        int temp = lenClass[classOf[i]];

        //lần lượt chèn a[i] vào cuối class của nó
        swap(a[i], a[temp - 1]);
        //2 ptu swap ở trên trong mảng class[] bị thay đổi do thứ tự của chúng trong a thay đổi
        swap(classOf[i], classOf[temp - 1]);
        //sau khi insert vào cuối mỗi class, upper bound của class đó giảm đi 1
        lenClass[classOf[i]]--;
    }
}
```

- Bước 3: htai thứ tự class trong mảng a không còn lộn xộn nữa & đã về đúng thứ tự, ngoài ra ta sẽ thấy 1 tính chất: các phần tử thuộc class[i] luôn lớn hơn các phần tử thuộc class[i-1]. Chính vì thế chỉ cần sắp xếp lại mỗi class là mảng sẽ được sắp xếp → áp dụng insertion sort cho mỗi class

Xét ví dụ: Cho mảng gồm 7 phần tử như sau: [n=7, m=int(0.45n)=3]. Chú thích

a,b	2 phần tử đã swap	class[1]
	Phần tử đang chèn trong class[i]	class[2]
	Cập nhật mảng sizeClass[]	class[3]

---BƯỚC 1---

Index	0	1	2	3	4	5	6		Index	1	2	3
a[]	10	9	7	5	-4	0	1		sizeClass[]	3	3	1
class[]	3	2	2	2	1	1	1					

Cập nhật sizeClass[]

Index	1	2	3
sizeClass[]	3	6	7

---BƯỚC 2---

- Với i=0, sizeClass[class[0]] = 7 → swap(a[0],a[6]) → cũng có nghĩa là chèn a[6] theo thứ tự ngược trong class[3]

Index	0	1	2	3	4	5	6
a[]	1	9	7	5	-4	0	10
class[]	1	2	2	2	1	1	3

class[1] (chưa đúng)

class[2] (chưa đúng)

class[3] (đúng)

Index	1	2	3
sizeClass[]	3	6	6

- Tiếp tục với $i=0$, $\text{sizeClass}[\text{class}[0]] = 3 \rightarrow \text{swap}(a[0], a[2]) \rightarrow$ cũng có nghĩa là chèn $a[0]$ theo thứ tự ngược trong $\text{class}[1]$

Index	0	1	2	3	4	5	6
a[]	7	9	1	5	-4	0	10
class[]	2	2	1	2	1	1	3

class[1] (chưa đúng)

class[2] (chưa đúng)

class[3] (đúng)

Index	1	2	3
sizeClass[]	2	6	6

- Tiếp tục với $i=0$, $\text{sizeClass}[\text{class}[0]] = 6 \rightarrow \text{swap}(a[0], a[5]) \rightarrow$ cũng có nghĩa là chèn $a[0]$ theo thứ tự ngược trong $\text{class}[2]$

Index	0	1	2	3	4	5	6
a[]	0	9	1	5	-4	7	10
class[]	1	2	1	2	1	2	3

class[1] (chưa đúng)

class[2] (chưa đúng)

class[3] (đúng)

Index	1	2	3
sizeClass[]	2	5	6

- Tiếp tục với $i=0$, $\text{sizeClass}[\text{class}[0]] = 2 \rightarrow \text{swap}(a[0], a[1])$

Index	0	1	2	3	4	5	6
a[]	9	0	1	5	-4	7	10
class[]	2	1	1	2	1	2	3

class[1] (chưa đúng)

class[2] (chưa đúng)

class[3] (đúng)

Index	1	2	3
sizeClass[]	1	5	6

- Tiếp tục với $i=0$, $\text{sizeClass}[\text{class}[0]] = 5 \rightarrow \text{swap}(a[0], a[4])$

Index	0	1	2	3	4	5	6
a[]	-4	0	1	5	9	7	10
class[]	1	1	1	2	2	2	3

class[1]

class[2]

class[3] (đúng)

Index	1	2	3
sizeClass[]	1	4	6

- Tiếp tục với $i=0$, $\text{sizeClass}[\text{class}[0]] = 1 \rightarrow \text{swap}(a[0], a[0])$

Index	0	1	2	3	4	5	6
a[]	-4	0	1	5	9	7	10
class[]	1	1	1	2	2	2	3

class[1]

class[2]

class[3] (đúng)

Index	1	2	3
sizeClass[]	0	4	6

- Tiếp tục với $i=0$, $\text{sizeClass}[\text{class}[0]] = 0 \rightarrow \text{break while}$ do không thỏa mãn điều kiện
- Với $i=1$; $\text{sizeClass}[\text{class}[1]] = 0 \rightarrow \text{break while}$
- Với $i=2$, $\text{sizeClass}[\text{class}[2]] = 0 \rightarrow \text{break while}$
- Với $i=3$, $\text{sizeClass}[\text{class}[3]] = 4 \rightarrow \text{swap}(a[3], a[3])$

Index	0	1	2	3	4	5	6
a[]	-4	0	1	5	9	7	10
class[]	1	1	1	2	2	2	3

class[1] (đúng)

class[2]

class[3] (đúng)

Index	1	2	3
sizeClass[]	0	3	6

- Với $i = 4, 5, 6 \rightarrow \text{break while}$ do không thỏa điều kiện. Mảng sau cùng là:

Index	0	1	2	3	4	5	6
a[]	-4	0	1	5	9	7	10
class[]	1	1	1	2	2	2	3

class[1] (đúng)

class[2] (đúng)

class[3] (đúng)

---BƯỚC 3---

- Áp dụng insertion sort cho lần lượt từng class trong mảng class[], kết quả cuối cùng thu được là:

Index	0	1	2	3	4	5	6
a[]	-4	0	1	5	7	9	10
class[]	1	1	1	2	2	2	3

class[1] (đúng)

class[2] (đúng)

class[3] (đúng)

\Rightarrow Vậy mảng ban đầu đã được sắp xếp

3. Đánh giá thuật toán

- Độ phức tạp thời gian:
 - + Trường hợp tốt nhất: $O(n)$
 - + Trường hợp trung bình: $O(n)$
 - + Trường hợp tệ nhất: $O(n^2)$. Xảy ra khi 1 mảng chỉ chia được rất ít/ chỉ có 1 class
- Độ phức tạp không gian: $O(n)$

4. Ưu, nhược điểm

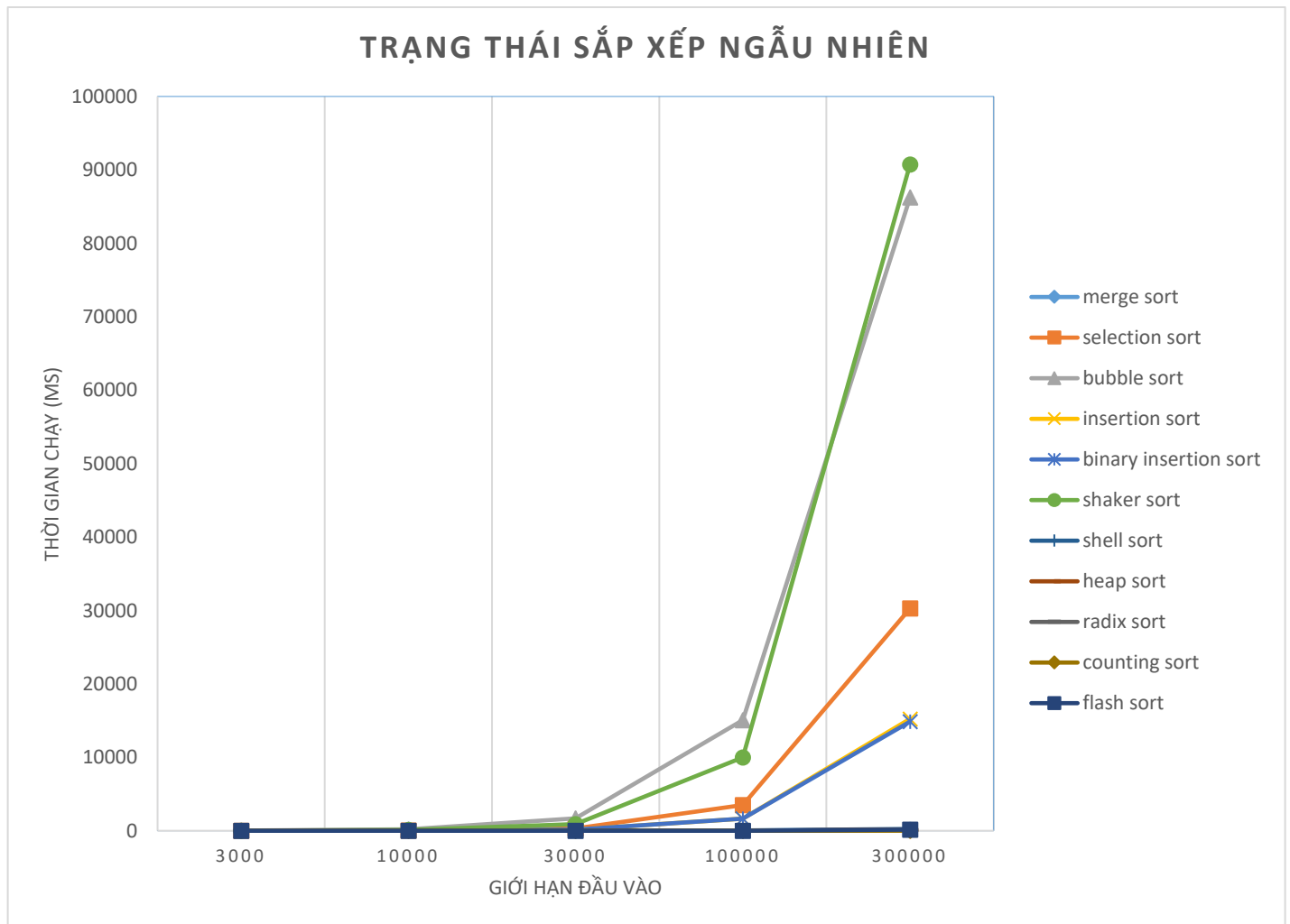
- Ưu điểm:
 - + Chạy tốt khi mảng đầu vào có kích thước tương đối lớn
- Nhược điểm:
 - + Ở trường hợp tệ nhất, flash sort chạy chậm hơn quicksort dù cho có cùng độ phức tạp thời gian $O(n^2)$

B. KẾT QUẢ THỰC NGHIỆM

Thông tin máy tính sinh viên	
Hệ điều hành	Window 10, 64 bit
RAM	8 GB
CPU	Intel Core i5

I. Trạng thái sắp xếp của dữ liệu đầu vào: **ngẫu nhiên**

data size	merge sort	selection sort	bubble sort	insertion sort	binary insertion sort	shaker sort	shell sort	heap sort	radix sort	counting sort	flash sort
3000	0.933	4.459	17.094	1.522	1.741	7.638	0.291	0.249	0.142	0.036	0.108
10000	4.157	53.782	195.243	17.011	18.197	97.355	1.168	0.958	0.536	0.11	0.388
30000	9.059	328.032	1680.81	157.842	163.36	915.075	4.142	3.435	2.053	0.399	1.513
100000	31.745	3501.35	15058.9	1632.13	1666.4	9989.55	15.234	13.29	6.549	1.344	5.167
300000	86.828	30273.7	86234.9	15233.3	14860.2	90746.3	239.149	146.932	75.199	11.922	148.508

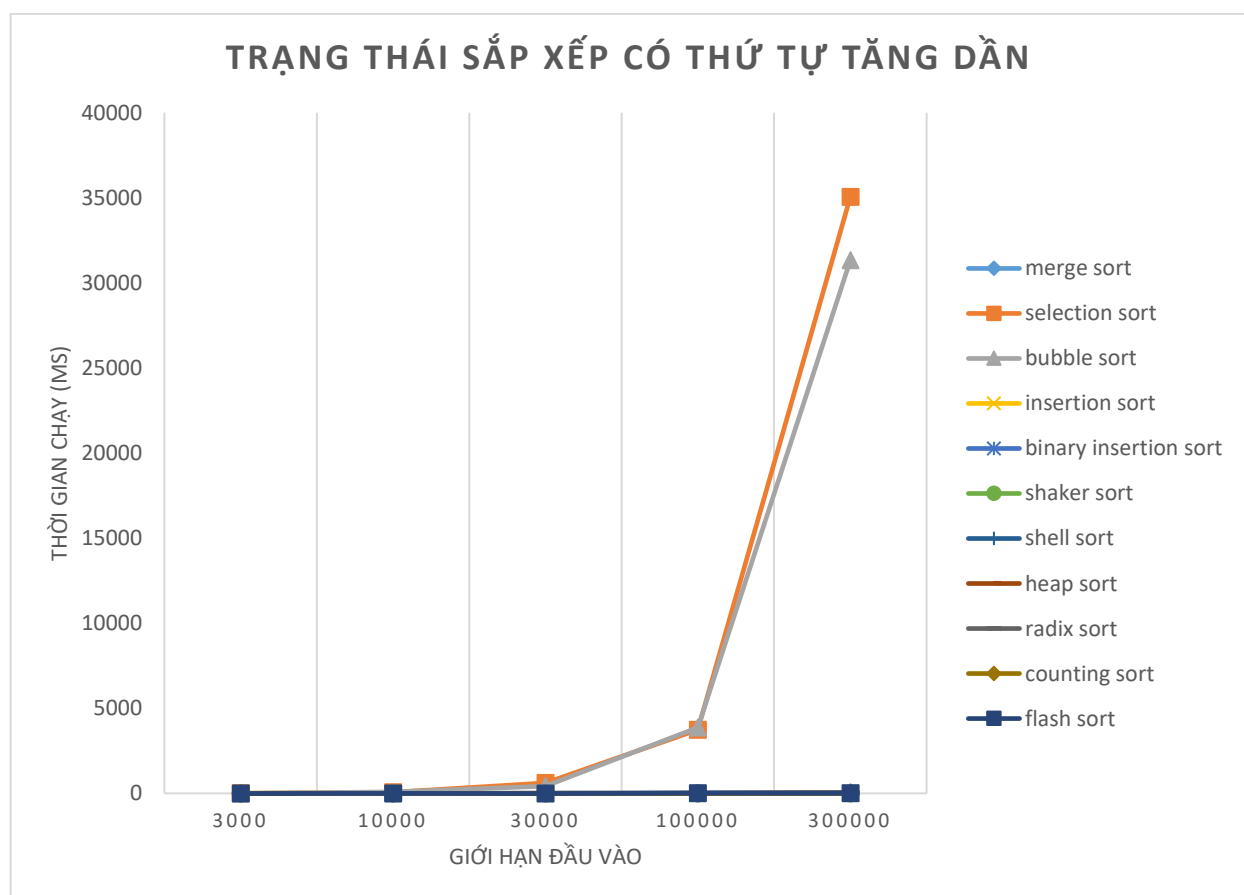


- Dưới 30.000 phần tử, chưa có sự khác biệt quá rõ rệt giữa thời gian chạy các thuật toán. Sự phân hóa bắt đầu khi giới hạn phần tử >30.000:

- + Dựa vào bảng, với mọi giới hạn đầu vào qui định (3000, 10000, ..., 300000), counting sort là thuật toán chạy nhanh nhất, theo sát nút sau đó là flash sort. Tuy nhiên 2 thuật toán này chạy chậm hơn trong trường hợp thứ tự đầu vào tăng dần/gần tăng dần
- + Ngay sau đó là binary insertion sort & insertion sort – với thời gian gần gấp đôi so với 2 thuật toán dẫn đầu & selection sort – với thời gian chạy gấp 3 lần so với 2 thuật toán dẫn đầu
- + Vị trí cuối cùng là bubble sort & shaker sort. Với dưới 10.000 phần tử thì shaker sort chạy tối ưu hơn bubble sort một chút, còn trên 10.000 phần tử thì thời gian của 2 thuật toán này không quá khác biệt

II. Trạng thái sắp xếp của dữ liệu đầu vào: có thứ tự tăng dần

data size	merge sort	selection sort	bubble sort	insertion sort	binary insertion sort	shaker sort	shell sort	heap sort	radix sort	counting sort	flash sort
3000	1.019	6.052	6.504	0.008	0.114	0.03	0.084	0.454	0.461	0.076	0.491
10000	14.769	70.297	87.629	0.028	0.547	0.054	0.547	1.069	0.892	0.162	0.424
30000	10.832	607.257	430.908	0.082	1.573	0.031	1.067	3.993	2.413	0.396	0.867
100000	24.075	3741	3875.07	0.313	3.657	0.069	3.66	11.652	6.041	0.705	2.525
300000	63.088	35054.1	31334.4	0.718	12.058	0.228	11.858	39.795	25.502	2.851	8.469



- Với dưới 30000 phần tử, chưa có gì thật sự khác biệt giữa các thuật toán. Trên 30000 phần tử, sự khác biệt rõ rệt nhất thể hiện qua 2 thuật toán selection sort & bubble sort. Đây là 2

thuật toán chạy chậm nhất (bubble sort chậm hơn selection sort) trong khi 10 thuật toán còn lại gần như chạy ngang hàng nhau ($< 5s$)

+ Dựa vào bảng, với mọi giới hạn đầu vào qui định (3000, 10000, ..., 300000), shaker sort là thuật toán chạy trong thời gian rất ngắn nhất do chỉ cần đúng 1 lần duyệt. Đây cũng là trường hợp tốt nhất của shaker sort (300.000 phần tử chưa tới 3 microsecond !). còn các trường hợp thứ tự sắp xếp khác shaker sort chạy chậm hơn, khi thứ tự sắp xếp giảm dần thì shaker sort chậm nhất trong tất cả thuật toán

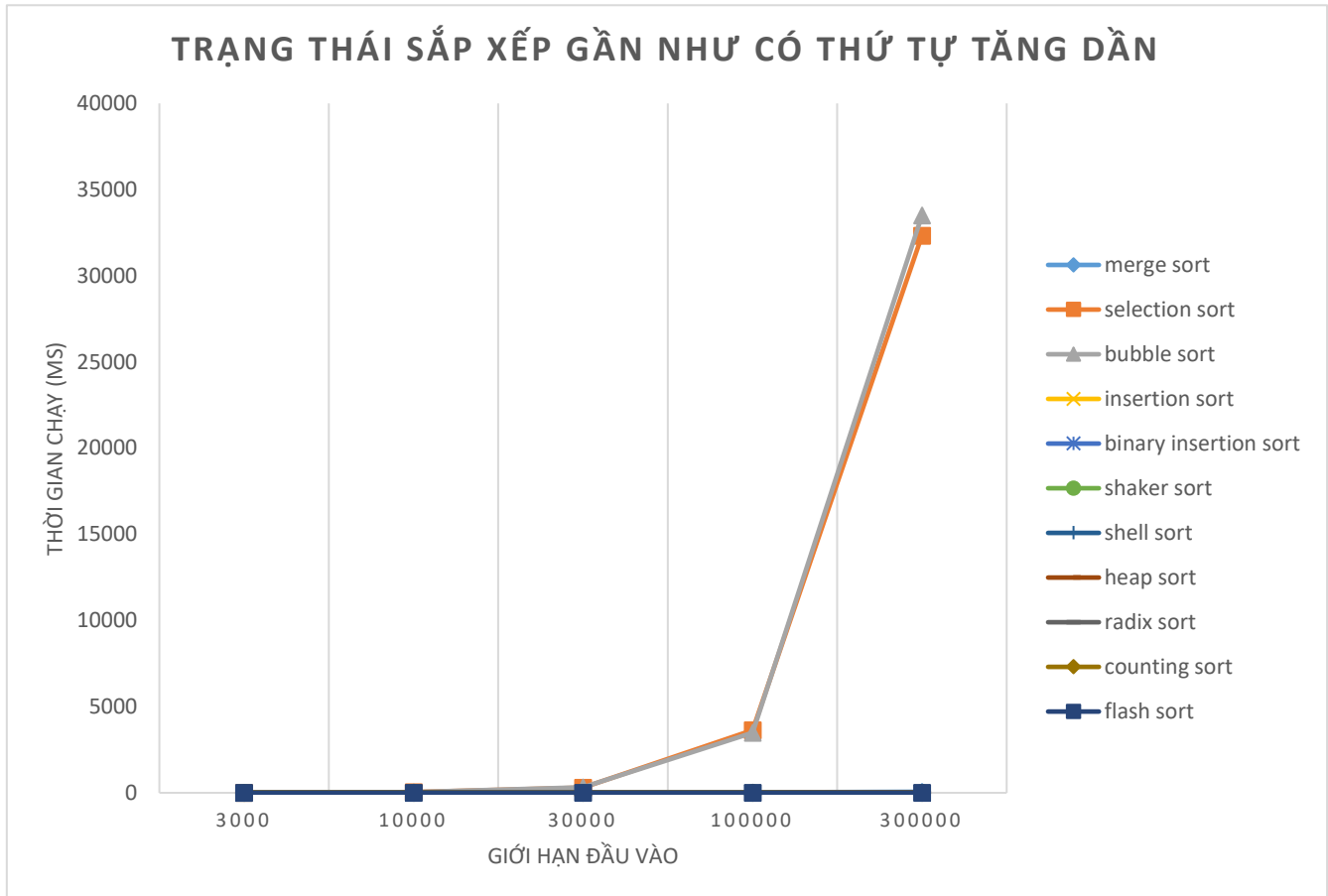
+ Thuật toán chạy tốt thứ 2 theo sát sau shaker sort là insertion sort – với thời gian chạy ở mỗi giới hạn đầu vào gấp 3 tới 4 lần insertion sort

- Selection sort do luôn phải duyệt phần tử tới cuối mảng để tìm min cho mỗi lần duyệt chính nên tốn chi phí $O(n^2)$, làm nó trở nên chậm chạp hơn nhiều khi kích thước đầu vào bắt đầu lớn. Tuy nhiên khi giới hạn đầu vào > 10.000 , selection sort vẫn cho thời gian chạy nhanh hơn bubble sort

Các thuật toán có thời gian chạy ổn định	Các thuật toán có thời gian chạy không ổn định
Flash sort, counting sort, shaker sort, radix sort, heap sort, shell sort, shaker sort, binary insertion sort, insertion sort, merge sort	Bubble sort, selection sort

III. Trạng thái sắp xếp của dữ liệu đầu vào: gần như có thứ tự tăng dần

data size	merge sort	selection sort	bubble sort	insertion sort	binary insertion sort	shaker sort	shell sort	heap sort	radix sort	counting sort	flash sort
3000	0.663	3.219	3.468	0.037	0.102	0.038	0.146	0.229	0.16	0.021	0.104
10000	1.915	33.044	33.923	0.123	0.505	0.168	0.541	1.001	0.776	0.09	0.368
30000	6.15	305.907	319.817	0.327	1.013	0.437	1.423	3.095	1.814	0.235	0.87
100000	19.696	3639.4	3481.67	0.391	3.752	0.487	4.117	11.747	7.172	0.665	2.628
300000	59.83	32326.1	33496.6	0.898	12.063	0.678	12.692	38.687	23.695	2.868	8.413

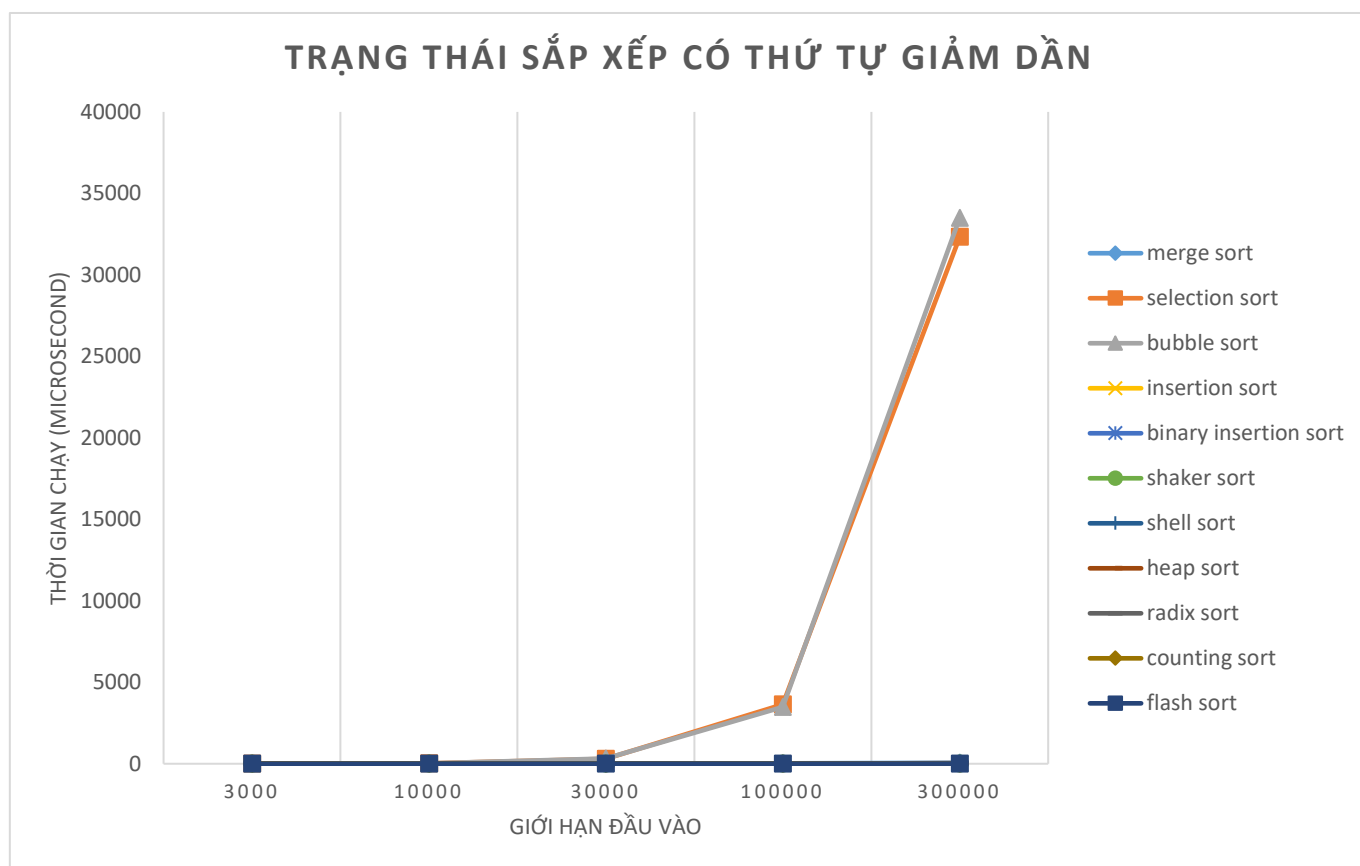


- Dưới 30.000 phần tử chưa có gì quá khác biệt giữa các thuật toán với nhau. Trên 30.000 phần tử, Bubble sort & selection sort là 2 thuật toán chạy chậm nhất (thời gian chạy của chúng xấp xỉ nhau), 10 thuật toán còn lại chạy nhanh tương đương nhau (<5s)
 - + Với selection sort, trường hợp tệ nhất là khi mảng gần được xếp tăng dần và các phần tử cận cuối của mảng lại là rất bé thì việc duyệt mảng tìm min tốn khá nhiều chi phí
- Dựa vào bảng, ta thấy được rằng với mỗi kích thước đầu vào thì sẽ có 1 thuật toán tối ưu để sắp xếp chứ không giới hạn về đúng 1 kiểu sắp xếp như 2 trạng thái trên. Nhưng xét về mặt trung bình, shaker sort có thời gian sắp xếp ổn nhất vì tại mỗi kích thước, thời gian không chênh lệch nhau quá nhiều (~ 100) như insertion sort hay counting sort

Các thuật toán có thời gian chạy ổn định	Các thuật toán có thời gian chạy không ổn định
Flash sort, counting sort, shaker sort, radix sort, heap sort, shell sort, shaker sort, binary insertion sort, insertion sort, merge sort	Bubble sort, selection sort

IV. Trạng thái sắp xếp của dữ liệu đầu vào: có thứ tự giảm dần

data size	merge sort	selection sort	bubble sort	insertion sort	binary insertion sort	shaker sort	shell sort	heap sort	radix sort	counting sort	flash sort
3000	0.907	4.323	5.876	3.335	3.079	7.335	0.086	0.213	0.147	0.026	0.095
10000	1.913	41.588	59.528	32.411	33.04	79.83	0.436	0.845	0.486	0.076	0.303
30000	5.722	381.794	527.109	318.983	312.693	763.577	1.128	2.894	1.781	0.214	0.82
100000	19.161	4346.08	6133	3805.31	3499.86	8569.97	4.23	10.819	6.146	0.699	2.855
300000	58.711	40046.4	55469.3	32161	31349.1	77968.8	13.526	36.48	24.027	2.457	10.577



- Dưới 30.000 phần tử chưa có sự khác biệt rõ rệt giữa các thuật toán. Nhưng từ 30.000 phần tử trở lên, mỗi đồ thị thuật toán đều có hướng đi riêng. Cụ thể:
 - + Counting sort vẫn giữ vị trí ưu thế, đó là lợi dụng được vùng nhớ phụ để lưu trữ, không có so sánh thừa như các thuật toán dựa trên so sánh khác → giảm thời gian tới mức thấp nhất. Theo ngay sau là flash sort với thời gian nhỉnh hơn không nhiều
 - + Gấp 3 lần thời gian chạy của 2 thuật toán đầu tiên, đồng thời giữ vị trí thứ 2 là shell sort & insertion sort, tiếp theo là selection sort. Cuối cùng là bubble sort & shaker sort (với shaker sort chạy chậm nhất). Có thể thấy tuy cùng chạy trong thời gian $O(n^2)$, tuy selection sort rơi vào trường hợp tệ nhất nhưng vẫn chạy nhanh hơn bubble sort & shaker sort do bản chất 2

thuật toán này là. Shaker sort tuy được đánh giá là cải tiến của bubble sort, nhưng trong trường hợp này vẫn chạy chậm hơn bubble sort

Các thuật toán có thời gian chạy ổn định	Các thuật toán có thời gian chạy không ổn định
Flash sort, counting sort, shaker sort, radix sort, heap sort, shell sort, insertion sort, merge sort	Bubble sort, selection sort, binary insertion sort, shaker sort