University of Science VNUHCM

21CTT5 - fit@hcmus

Group 14

COMPARISON OF SORTING ALGORITHMS

1. Information page

Subject	Data structures and algorithms
Teacher	Lê Đình Ngọc
Class	21CTT5
Topic	Sorting algorithms
Students	21120533: Trần Thái Tân 21120566: Nguyễn Hữu Thuận 21120595: Nguyễn Thành Vinh 21120547: Thạch Thị Sinh
Set	Selection Sort Insertion Sort Bubble Sort Shaker Sort Shell Sort Heap Sort Merge Sort Quick Sort Counting Sort Radix Sort Flash Sort

2. Introduction page

- The purpose of this research is to determine the running time of sorting algorithms and compare them.
- Completed 11/11 algorithms in the selected set and 5/5 commands.
- Rating of completion: 100%.

3. Algorithm presentation

3.1. Selection Sort

3.1.1. Ideas

- Mảng được chia ra thành 2 thành hai phần:
  + *Bên trái: Phần đã sắp xếp (gọi tắt là **L**).*
  + *Bên phải: Phần chưa sắp xếp (gọi tắt là **R**).*
- Do vậy, với một mảng gồm n phần tử, ban đầu **L** sẽ có **1** phần tử và **R** sẽ có $n-1$ phần tử.

- Phần tử nhỏ nhất của **R** sẽ được hoán đổi với phần tử cuối cùng của **L**, sau đó thêm phần tử đầu tiên của **R** vào cuối **L**. Điều này sẽ liên tục lặp lại đến khi **R** rỗng.
- Sau khi kết thúc, ta sẽ nhận được một mảng chỉ bao gồm **L**, đồng nghĩa với việc mảng đã được sắp xếp.

3.1.2. Step-by-step descriptions

- **Cho một mảng 7 phần tử như bên dưới, lúc này L sẽ có 1 phần tử và R có 6 phần tử.**

4	6	1	7	3	5	2

- **Lần hoán vị đầu tiên:**
  + *Xác định hai phần tử cần hoán vị là 4 và 1.*

4	6	1	7	3	5	2

  + *Hoán đổi hai phần tử. Vì 6 lúc này lớn hơn mọi phần tử của **L** nên 6 thuộc về phần đã sắp xếp.*

1	6	4	7	3	5	2

- **Lần hoán vị thứ hai:**
  + *Xác định hai phần tử cần hoán vị là 6 và 2.*

1	6	4	7	3	5	2

  + *Hoán đổi hai phần tử. Vì 4 lúc này lớn hơn mọi phần tử của **L** nên 4 thuộc về phần đã sắp xếp.*

1	2	4	7	3	5	6

- **Lần hoán vị thứ ba:**
  + *Xác định hai phần tử cần hoán vị là*

1	2	4	7	3	5	6

  + *Hoán đổi hai phần tử. Vì 7 lúc này lớn hơn mọi phần tử của **L** nên 7 thuộc về phần đã sắp xếp.*

1	2	3	7	4	5	6

- **Lần hoán vị thứ tư:**
  + *Xác định hai phần tử cần hoán vị*

1	2	3	7	4	5	6

  + *Hoán đổi hai phần tử. Vì 7 lúc này lớn hơn mọi phần tử của **L** nên 7 thuộc về phần đã sắp xếp.*

1	2	3	4	7	5	6

- **Lần hoán vị thứ năm:**
  + *Xác định hai phần tử cần hoán vị*

1	2	3	4	7	5	6

  + *Hoán đổi hai phần tử. Vì 7 lúc này lớn hơn mọi phần tử của **L** nên 7 thuộc về phần đã sắp xếp.*

1	2	3	4	5	7	6

- **Lần hoán vị cuối cùng:**
  + *Xác định hai phần tử cần hoán vị*

1	2	3	4	5	7	6

  + *Hoán đổi hai phần tử. Vì 7 lúc này lớn hơn mọi phần tử của **L** nên 7 thuộc về phần đã sắp xếp.*

1	2	3	4	5	6	7

- **Từ đây, toàn bộ mảng đã trở thành L - phần đã sắp xếp:**

1	2	3	4	5	6	7

3.1.3. Complexity evaluations

- **Time complexity:**

  Average case, best case, worst case: $O(n^2)$

  Giải thích:
  + Trong lần hoán vị đầu tiên, ta thực hiện $n-1$ (số lượng phần tử của **R**) phép so sánh.
  + Trong lần hoán vị thứ hai, ta thực hiện $n-2$ (số lượng phần tử của **R**) phép so sánh.
  + Trong lần hoán vị thứ ba, ta thực hiện $n-3$ (số lượng phần tử của **R**) phép so sánh.
  + Trong lần hoán vị thứ i, ta thực hiện $n-i$ (số lượng phần tử của **R**) phép so sánh.
  + Trong lần hoán vị thứ $n-1$, ta thực hiện 1 (số lượng phần tử của **R**) phép so sánh.
  + Vậy sau $n-1$ lần hoán vị, ta thực hiện tổng cộng:

$$\sum_{i=1}^{n-1}(n-i) = (n-1)+(n-2)+(n-3)+\cdots+1 = \frac{n(n-1)}{2} \sim O(n^2)$$

(với mọi trường hợp, ta đều phải thực hiện một lượng so sánh tương đương như trên)

- **Space complexity:**

  Average case, best case, worst case: $O(1)$

  Giải thích: Trong mỗi lần hoán vị, ta sẽ chỉ cần một "không gian" cho một phần tử để so sánh.

3.2. Insertion Sort

3.2.1. Ideas

Ý tưởng

Ta có mảng ban đầu gồm phần tử A[0] xem như đã sắp xếp, ta sẽ duyệt từ phần tử 1 đến n – 1, tìm cách chèn những phần tử đó vào vị trí thích hợp trong mảng ban đầu đã được sắp xếp.

Thuật toán

- Gán i = 1
- Gán x = A[i] và pos = i – 1
- Nếu pos >= 0 và A[pos] > x

  - A[pos + 1] = A[pos]

  - pos = pos – 1

  - Quay lại bước 3
- A[pos + 1] = x
- Nếu i < n:

  - Đúng thì i = i + 1 và quay lại bước 2

  - Sai thì dừng lại

3.2.2. Step-by-step descriptions

- Tạo mảng

- Giải thuật sắp xếp chèn so sánh hai phần tử đầu tiên:

- Cả 14 và 33 đều đã trong thứ tự tăng dần. Hiện tại, 14 là trong danh sách con đã qua sắp xếp.

- Giải thuật tiếp tục di chuyển tới phần tử kế tiếp và so sánh 33 với 27

- Và thấy 33 không nằm ở vị trí đúng, nên giải thuật sắp xếp chèn tráo đổi vị trí của 33 và 27. Đồng thời cũng kiểm tra tất cả các phần tử trong danh sách con đã sắp xếp. Trong danh sách con này chỉ cso 14 và 27 là lớn hơn 14. Do vậy danh sách con vẫn giữ nguyên sau khi đã tráo đổi.

- Trong danh sách con chúng ta có hai giá trị là 14 và 27. Tiếp tục so sánh 33 với 10.

- Hai giá trị này không theo thứ tự.

- Vì thế chúng ta tráo đổi chúng.

- Việc tráo đổi dẫn đến 27 và 10 không theo thứ tự.

- Vì thế chúng ta cũng tráo đổi chúng.

- Chúng ta lại thấy rằng 14 và 10 không theo thứ tự.

- Và chúng ta tiếp tục tráo đổi hai số này. Cuối cùng, sau vòng lặp thứ 3 chúng ta có 4 phần tử.

- Tiến trình trên sẽ tiếp tục diễn ra cho tới khi tất cả giá trị chưa sắp xếp được sắp xếp hết vào trong danh sách con đã qua sắp xếp.

3.2.3. Complexity evaluations

- **Time complexity:**
- Độ phức tạp của trường hợp xấu nhất: $O(n2)$
  - Giả sử, một mảng có thứ tự tăng dần và ta muốn sắp xếp nó theo thứ tự giảm dần. Trong trường hợp này, trường hợp xấu nhất sẽ xảy ra.
  - Mỗi phần tử phải được so sánh với mỗi phần tử khác, do đó, đối với mỗi phần tử thứ n, (n-1) số phép so sánh sẽ được thực hiện.
  - Do đó, tổng số phép so sánh $= n \times (n-1)n2$
- Độ phức tạp của trường hợp tốt nhất: $O(n)$
  - Khi mảng đã được sắp xếp, vòng lặp bên ngoài chạy trong n số lần trong khi vòng lặp bên trong hoàn toàn không chạy. Vì vậy, chỉ có n số phép so sánh được thực hiện. Do đó, độ phức tạp là tuyến tính.
- Độ phức tạp của trường hợp trung bình: $O(n)$
  - Xảy ra khi các phần tử của một mảng có thứ tự lộn xộn (không tăng dần cũng không giảm dần).

- **Space complexity:**

  Độ phức tạp của không gian là O(1) vì một biến phụ được sử dụng.

3.3. Bubble Sort

3.3.1. Ideas

Ý tưởng:

- Duyệt qua danh sách, làm cho các phần tử lớn nhất hoặc nhỏ nhất dịch chuyển về phía cuối danh sách, tiếp tục lại làm phần tử lớn nhất hoặc nhỏ nhất kế đó dịch chuyển về cuối hay chính là làm cho phần tử nhỏ nhất (hoặc lớn nhất) nổi lên, cứ như vậy cho đến hết danh sách

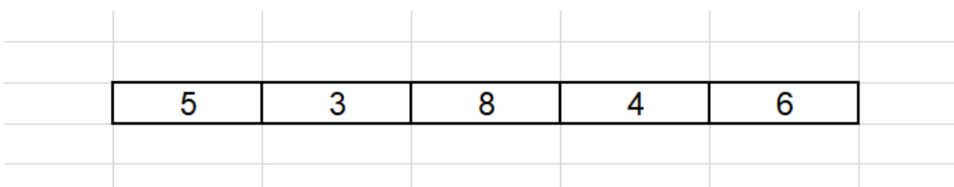
Thuật toán:

Xét một mảng gồm *n* số nguyên: *a*1 ,*a*2 ,*a*3 ,...,*an*

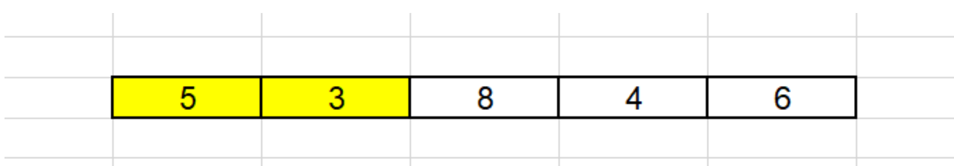
1. Với cách sắp xếp không giảm từ trái qua phải, mục đích của chúng ta là đưa dần các số lớn nhất về cuối dãy (ngoài cùng bên phải).
2. Bắt đầu từ vị trí số 1, xét lần lượt từng cặp 2 phần tử, nếu phần tử bên phải nhỏ hơn phần tử bên trái, ta sẽ thực hiện đổi chỗ 2 phần tử này, nếu không, xét tiếp cặp tiếp theo. Với cách làm như vậy, phần tử nhỏ hơn sẽ "nổi" lên, còn phần tử lớn hơn sẽ "chìm" dần và về bên phải.
3. Khi kết thúc vòng thứ nhất, ta sẽ đưa được phần tử lớn nhất về cuối dãy. Sang vòng thứ hai, ta tiếp tục bắt đầu ở vị trí đầu tiên như vậy và đưa được phần tử lớn thứ hai về vị trí thứ hai ở cuối dãy ...

3.3.2. Step-by-step descriptions

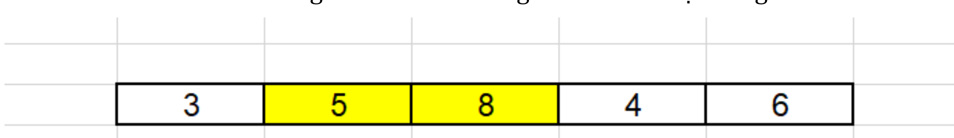
- Tạo mảng



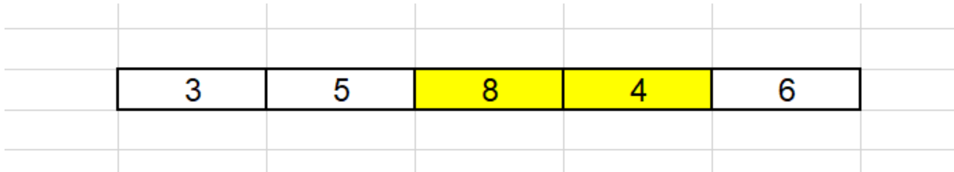
- So sánh 5 và 3 hoán đổi chúng theo thứ tự mong muốn



- Tiếp theo so sánh 5 với 8 và không hoán đổi vì chúng đã theo thứ tự mong muốn



- So sánh 8 với 4 và hoán đổi chúng vì 8 > 4



- Và cứ tiếp tục như vậy đến khi phần tử lớn nhất đặt đúng vị trí của kết quả cuối cùng

3.3.3. Complexity evaluations

- **Time complexity:**

- Độ phức tạp của trường hợp xấu nhất: O(n2).
- Nếu chúng ta muốn sắp xếp theo thứ tự tăng dần và mảng theo thứ tự giảm dần thì trường hợp xấu nhất sẽ xảy ra.
- Độ phức tạp của trường hợp tốt nhất: O(n).
- Nếu mảng đã được sắp xếp, thì không cần sắp xếp.
- Độ phức tạp của trường hợp trung bình: O(n2)
- Nó xảy ra khi các phần tử của mảng có thứ tự lộn xộn (không tăng dần cũng không giảm dần).

- **Space complexity:**

- Độ phức tạp của không gian là O(1) vì một biến trung gian được bổ sung được sử dụng để hoán đổi. Trong thuật toán được tối ưu hóa, biến được hoán đổi làm tăng thêm độ phức tạp của không gian, do đó là O(2).

3.4. Shaker Sort

3.4.1. Ideas

1. Bước 1: Khai báo kiểu biến và giá trị cho biến cần sử dụng trong đoan chương trình, cụ thể chương trình ở đây sử dụng biến left, right, và k. Do chúng ta đang có 1 mảng 1 chiều và nhiệm vụ trong thuật toán shakersort này ta cần phải gán biến left = 0 (chính là biến ở đầu mảng), right = n-1 (biến ở cuối mảng), k=n-1 (định vị trí bắt đầu lắc).

2. Bước 2: Sau đó, ta dùng 1 vòng lặp while với điều kiện lặp là left < right. Trong vòng lặp while này sẽ có 2 vòng lặp for nữa, ta cứ hiểu 2 vòng lặp for này là 1 lượt đi và 1 lượt về.

   + Lượt đi:

   - Ta duyệt vòng lặp for từ cuối mảng tới đầu mảng, nếu gặp cặp nghịch thế(số trước lớn hơn số đầu) thì ta gọi hàm hoán vị (hoặc dùng swap(M[i-1] ,M[i]);). Trong lần lặp này sẽ đưa được giá trị nhỏ nhất trong mảng về vị trí đầu tiên.
   - Dùng biến k đánh dấu để bỏ qua đoạn đã được sắp xếp thứ tự.

   + Lượt về:

   - Lấy k = left .Ta duyệt từ đầu mảng đến cuối mảng bắt đầu từ vị trí k , nếu gặp cặp nghịch thế(số trước lớn hơn số đầu) thì ta gọi hàm hoán vị (hoặc dùng swap(M[i] ,M[i+1]);). Trong lần lặp này sẽ đưa được giá trị lớn nhất trong mảng về vị trí cuối cùng.
   - Dùng biến k đánh dấu để bỏ qua đoạn đã được sắp xếp thứ tự. Sau đó gán k = right

3. Bước 3: Lặp lại bước 2

Kiểm tra chiều dài đoạn cần sắp xếp nếu = 1 thì ngưng, còn nếu > 1 thì lặp lại bước 2.

3.4.2. Step-by-step descriptions

- Tạo mảng

12	2	8	5	1	6	4	15

- L = 1, r = 8

  Lượt đi

						<--------	
12	2	8	5	1	6	4	15

  Lượt về: cập nhật 1 = 2

3.4.3. Complexity evaluations

- Độ phức tạp cho trường hợp tốt nhất là O(n).
- Độ phức tạp cho trường hợp xấu nhất O(n^2).
- Độ phức tạp trong trường hợp trung bình là O(n^2).
- Thuật toán nhận diện được mảng đã sắp xếp.

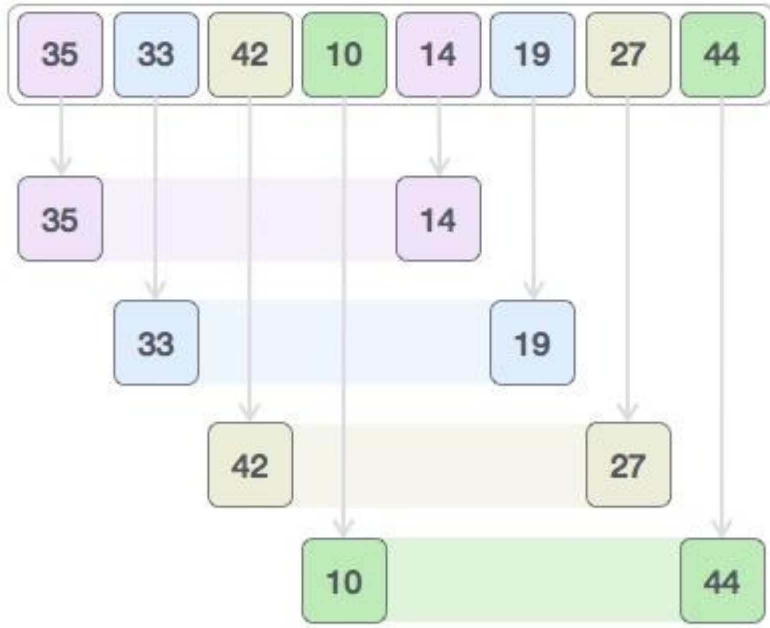
3.5. Shell Sort

3.5.1. Ideas

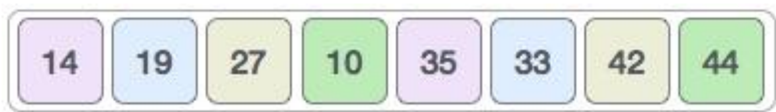
Content

3.5.2. Step-by-step descriptions

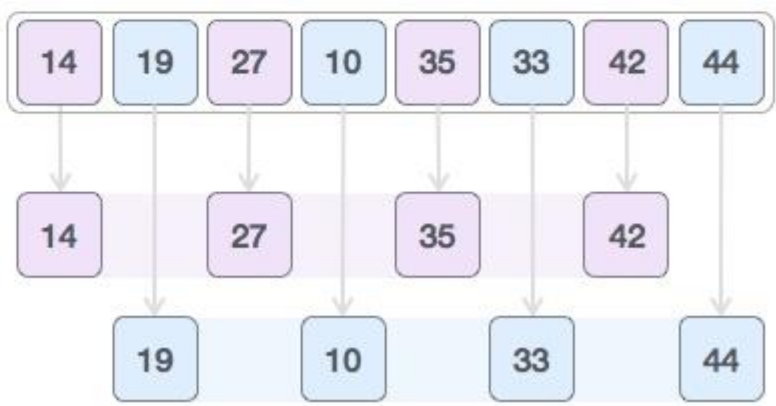
- Chúng ta sử dụng một mảng gồm các giá trị như dưới đây. Giả sử ban đầu giá trị Khoảng (interval) là 4. Ví dụ, với phần tử 35 thì với khoảng là 4 thì phần tử còn lại sẽ là 14. Do đó ta sẽ có các cặp giá trị {35, 14}, {33, 19}, {42, 27}, và {10, 14}.



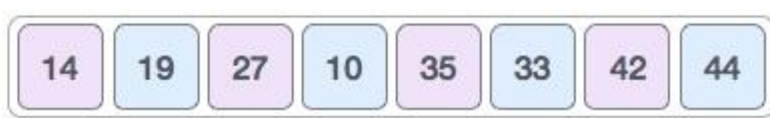
- So sánh các giá trị này với nhau trong các danh sách con và tráo đổi chúng (nếu cần) trong mảng ban đầu. Sau bước này, mảng mới sẽ trống như sau:



- Sau đó, lấy giá trị Khoảng (interval) là 2 và với khoảng cách này sẽ cho hai danh sách con: {14, 27, 35, 42}, {19, 10, 33, 44}.



- Tiếp tục so sánh và tráo đổi các giá trị (nếu cần) trong mảng ban đầu. Sau bước này, mảng sẽ trông như sau:



- Cuối cùng, chúng ta sắp xếp phần mảng còn lại này với Khoảng (interval) bằng 1. Shell Sort sử dụng giải thuật sắp xếp chèn để sắp xếp mảng.

3.5.3. Complexity evaluations

Time complexity:

- Độ phức tạp của trường hợp xấu nhất: Nhỏ hơn hoặc bằng O(n2)
    - Độ phức tạp của trường hợp xấu nhất của sắp xếp Shell Sort luôn nhỏ hơn hoặc bằng O(n2).
    - Theo định lý Poonen, độ phức tạp trong trường hợp xấu nhất cho kiểu sắp xếp này là Θ(NlogN)2/(loglogN)2) hoặc Θ(NlogN)2/loglogN) hoặc Θ(N(logN)2).
- Độ phức tạp của trường hợp tốt nhất: O(n∗logn)
    - Khi mảng đã được sắp xếp, tổng số phép so sánh cho mỗi khoảng (hoặc khoảng tăng) bằng kích thước của mảng.
- Độ phức tạp của trường hợp trung bình: O(n∗logn)
    - Nằm ở nằm trong khoảng xung quanh O(n1,25).
    - Độ phức tạp phụ thuộc vào khoảng được chọn. Các độ phức tạp trên khác nhau đối với các trình tự gia tăng khác nhau được chọn. Trình tự tăng tốt nhất là không xác định.

Space complexity:

Độ phức tạp về không gian cho kiểu sắp xếp Shell Sort là O(1)

3.6. Heap Sort

3.6.1. Ideas

Thuật toán Heap sort lấy ý tưởng giải quyết từ cấu trúc heap, cụ thể:

- Ta coi dãy cần sắp xếp là một cây nhị phân hoàn chỉnh, sau đó hiệu chỉnh cây thành dạng cấu trúc heap (vun đống)
- Dựa vào tính chất của cấu trúc heap, ta có thể lấy được ra phần từ lớn nhất hoặc nhỏ nhất của dãy, phần tử này chính là gốc của heap. Giảm số lượng phần tử của cây nhị phân và tái cấu trúc heap.
- Đưa phần tử đỉnh heap về đúng vị trí của dãy ở cuối mảng, sau đó giảm số lượng phần tử của mảng (không xét tới phần tử cuối nữa)
- Tái cấu trúc heap và lặp lại việc lấy phần tử gốc của cấu trúc heap cho tới khi danh sách ban đầu chỉ còn 1 phần tử. Đưa phần tử này về đúng vị trí và kết thúc thuật toán.

Ta phải thực hiện tái cấu trúc heap, vun lại đống bởi vì sau khi lấy ra phần tử gốc heap, cấu trúc heap không còn nữa.

Thuật toán được chia làm 2 phần:

- Hàm vun đống
- Hàm sắp xếp

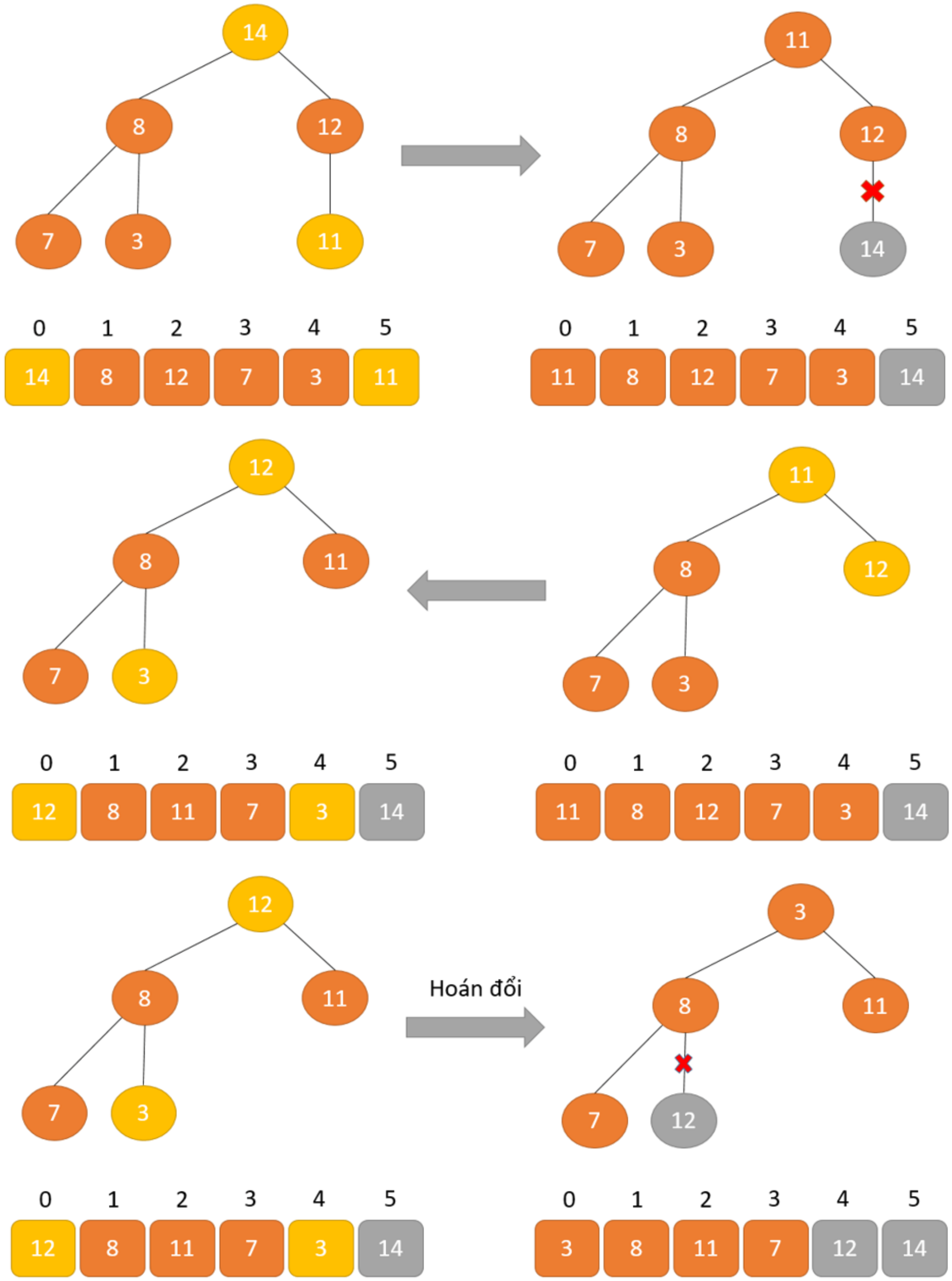
Hàm vun đống một đỉnh: Hàm vun đống là hàm sẽ giúp tạo cấu trúc heap từ mảng ban đầu. Ta sẽ viết hàm vun đống cho 1 đỉnh i, và một số nội dung mình cần chú ý đó là.

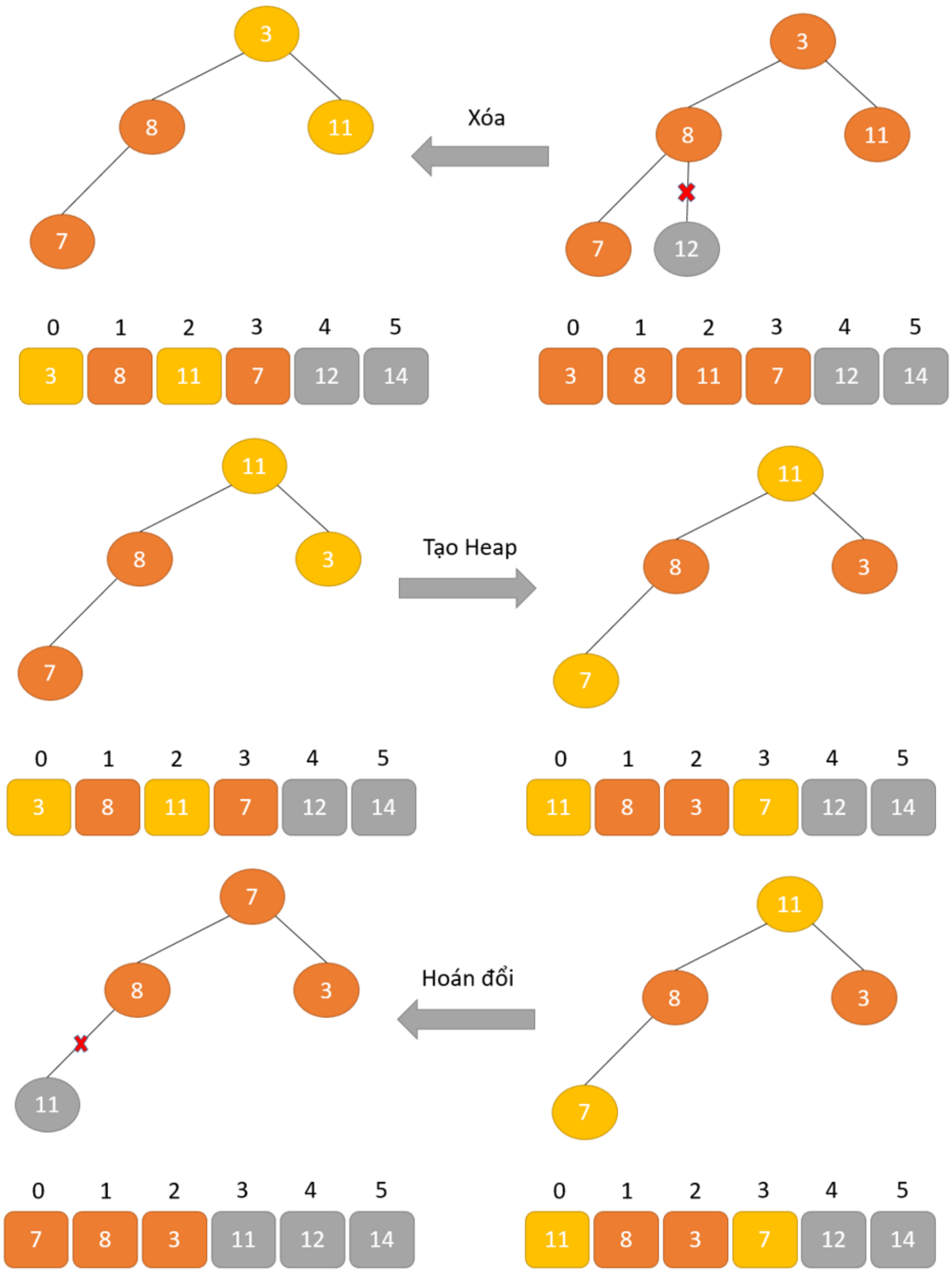
a. Nếu đỉnh cha có chỉ số là **i** thì con trái có chỉ số là: $\mathbf{L = i * 2 + 1}$
   Con phải có chỉ số là: r = i*2 +2
   Trong đó L và **r** phải nhỏ hơn **n** (số phần tử của mảng)
b. Việc cần làm là tìm ra đỉnh lớn nhất trong 3 đỉnh: i, L, r
c. Nếu đỉnh lớn nhất khác đỉnh ban đầu, ta tiến hành đổi chỗ. Đệ quy vun đống tại vị trí vừa đổi chỗ để vun các node tiếp theo.

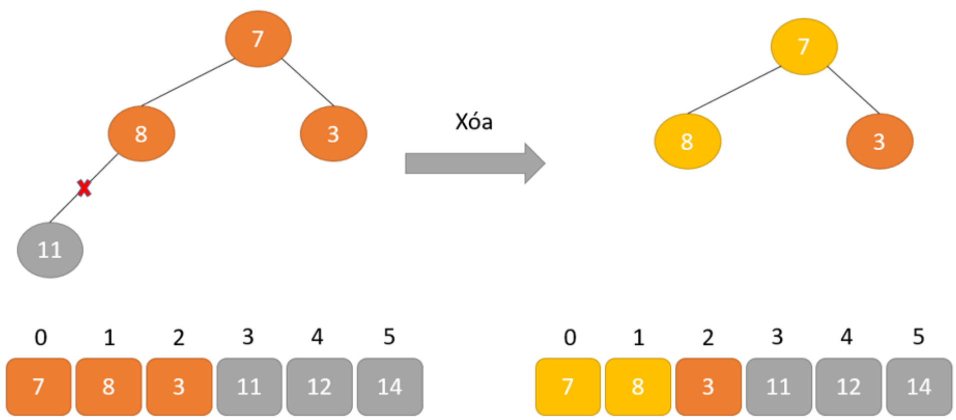
Hàm sắp xếp vun đống: Hàm heapify bên trên giúp vun một đỉnh thành một đống, bây giờ ta chỉ cần vun toàn bộ dãy ban đầu thành đống, sau đó lấy ra phần tử lớn nhất cho về cuối mảng. Lặp lại hành động này cho tới khi thu được dãy sắp xếp.

a. Để vun đống, ta sẽ phải vun từ dưới lên, tức là vun từ đỉnh cha cuối cùng của heap. Một mảng **n** phần tử sẽ có $\mathbf{n/2}$ node cha.
b. Đổi chỗ đỉnh gốc đống với phần tử cuối cùng của mảng. Như vậy phần tử lớn nhất sẽ nằm ở cuối mảng
c. Giảm số lượng phần tử (loại bỏ phần tử cuối cùng đúng vị trí)
d. Lặp lại việc vun đống và lấy phần tử cho tới khi còn 1 phần tử thì dừng
e. Một mảng n phần tử sẽ phải thực hiện vun đống n lần

3.6.2. Step-by-step descriptions




Xóa
Tạo Heap
Hoán đổi



3.6.3. Complexity evaluations

- Heap Sort có độ phức tạp về thời gian là O(nlogn) cho tất cả các trường hợp (trường hợp tốt nhất, trường hợp trung bình và trường hợp xấu nhất).
- Chiều cao của một cây nhị phân hoàn chỉnh chứa n phần tử là logn
- Như chúng ta đã thấy trước đó, để tạo cấu trúc Heap cho một phần tử có các cây con đã là Max Heap, chúng ta cần tiếp tục so sánh phần tử với các phần tử con bên trái và bên phải của nó và đẩy nó xuống dưới cho đến khi nó đạt đến điểm mà cả hai cây con của nó đều nhỏ hơn nó.
- Trong trường hợp xấu nhất, chúng ta sẽ cần phải di chuyển một phần tử từ nút gốc đến nút lá để thực hiện nhiều phép so sánh và hoán đổi log(n).
- Trong giai đoạn xây dựng cấu trúc Max Heap, chúng ta đã thực hiện điều đó cho n2 phần tử nên độ phức tạp trong trường hợp xấu nhất của bước này là n2×log(n)≈nlog(n).
- Trong bước sắp xếp, chúng ta hoán đổi phần tử gốc với phần tử cuối cùng và tạo cấu trúc Heap cho phần tử gốc. Đối với mỗi phần tử, điều này lại làm tốn thời gian là log(n) vì chúng ta có thể phải đưa phần tử đó từ nút gốc đến nút lá. Vì chúng ta lặp lại n lần này nên bước sắp xếp vun đống cũng là nlog(n).
- Cũng vì các bước xây dựng cấu trúc Max Heap và sắp xếp vun đống được thực hiện lần lượt nên độ phức tạp của thuật toán không được nhân lên và nó vẫn theo thứ tự nlog(n).
- Ngoài ra, nó thực hiện sắp xếp theo độ phức tạp về không gian là O(1). So với Sắp xếp nhanh, nó có trường hợp xấu nhất tốt hơn là (O(nlogn)). Sắp xếp nhanh có độ phức tạp O(n2) cho trường hợp xấu nhất. Nhưng trong các trường hợp khác, thuật toán sắp xếp nhanh hay QuickSort sẽ nhanh hơn.

3.7. Merge Sort

3.7.1. Ideas

Ý tưởng:

- Chia đôi mảng thành hai mảng con, sắp xếp hai mảng con đó và trộn lại theo đúng thứ tự, mảng con được sắp xếp bằng cách tương tự.

Thuật toán:

1. Tìm vị trí chính giữa mảng
2. Sắp xếp mảng thứ nhất (từ vị trí left đến mid)
3. Sắp xếp mảng thứ hai (từ vị trí mid + 1 đến right)

4. Trộn hai mảng đã sắp xếp với nhau

3.7.2. Step-by-step descriptions

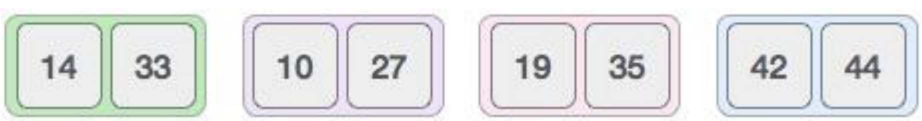
- Tạo 1 mảng

- Đầu tiên, giải thuật sắp xếp trộn chia toàn bộ mảng thành hai nửa. Tiến trình chia này tiếp tục diễn ra cho đến khi không còn chia được nữa và chúng ta thu được các giá trị tương ứng biểu diễn các phần tử trong mảng. Trong hình dưới, đầu tiên chúng ta chia mảng kích cỡ 8 thành hai mảng kích cỡ 4.

- Tiến trình chia này không làm thay đổi thứ tự các phần tử trong mảng ban đầu. Bây giờ chúng ta tiếp tục chia các mảng này thành 2 nửa.

- Tiến hành chia tiếp cho tới khi không còn chia được nữa.

- Bây giờ chúng ta tổ hợp chúng theo như đúng cách thức mà chúng được chia ra.
- Đầu tiên chúng ta so sánh hai phần tử trong mỗi list và sau đó tổ hợp chúng vào trong một list khác theo cách thức đã được sắp xếp. Ví dụ, 14 và 33 là trong các vị trí đã được sắp xếp. Chúng ta so sánh 27 và 10 và trong list khác chúng ta đặt 10 ở đầu và sau đó là 27. Tương tự, chúng ta thay đổi vị trí của 19 và 35. 42 và 44 được đặt tương ứng.



- Vòng lặp tiếp theo là để kết hợp từng cặp list một ở trên. Chúng ta so sánh các giá trị và sau đó hợp nhất chúng lại vào trong một list chứa 4 giá trị, và 4 giá trị này đều đã được sắp thứ tự.

- Sau bước kết hợp cuối cùng, danh sách sẽ trông giống như sau:

3.7.3. Complexity evaluations

- **Time complexity:**

- Độ phức tạp của trường hợp tốt nhất: O(n * log n)
- Độ phức tạp của trường hợp xấu nhất: O(n * log n)
- Độ phức tạp của trường hợp trung bình: O(n * log n)
- **Space complexity:**

Độ phức tạp không gian của sắp xếp hợp nhất là O(n).

3.8. Quick Sort

3.8.1. Ideas

Ý tưởng

- Phân hoạch dãy ban đầu thành 3 phần
  - Mảng 1: a0, a1, …, ai có giá trị nhỏ hơn x.
  - Mảng 2: ak = x.
  - Mảng 3: aj, …, an-1 có giá trị lớn hơn x.
- Về cơ bản, nếu xem dãy 1 và 3 chỉ là một phần tử thì mảng đã được sắp xếp tăng dần, vì vậy ta tiến hành phần hoạch hai dãy con 1 và 3 như dãy ban đầu cho đến khi mảng đã hoàn toàn được sắp xếp.

Thuật toán

- Phân hoạch dãy con có vị trí của phần tử bắt đầu là r, kết thúc là l
  1 Chọn phần tử ak là mốc để phân hoạch (0 <= k <= n-1)
    - Gán x = ak, I = l, j =r
    - Chưa có một chứng minh gì chọn việc chọn vị trí k tối ưu nhất, thông thường hay chọn phần tử ở chính giữa dãy phần hoạch: k = (l + r) /2

  1 Hiệu chỉnh cặp phần tử a[i] và a[j] sai vị trí
    - Nếu a[i] < x, tăng i.
    - Nếu a[j] > x, tăng j.
    - Nếu i <= j thì
      + Hoán vị a[i] và a[j]
      + Tăng i và giảm j.

  1 So sánh i và j
    - Nếu i < j : lặp lại bước 2.
    - Ngược lại: dừng phân hoạch.
- Thuật toán Quick sort cho cả mảng
  1. Phân hoạch dãy ban đầu thành 3 mảng con
    - Mảng 1: a[l], a1, …, aj có giá trị nhỏ hơn x.
    - Mảng 2: a[j+1], …, a[i-1] = x.
    - Mảng 3: ai, …, a[r] có giá trị lớn hơn x.

  2. Sắp xếp
    - Nếu l < j : phân hoạch dã a[l],…, a[j]
    - Nếu i < r: phân hoạch dãy a[i],… ,a[r]

3.8.2. Step-by-step descriptions

- Tạo mảng (giả sử chọn mảng có 8 phần tử như hình dưới), với phần tử pivot ban đầu ở vị trí 3 (pivot = 7)

	i			pivot				j
	5	9	2	7	4	6	8	1
index:	0	1	2	3	4	5	6	7

- *Phân hoạch lần đầu với phần tử pivot = 7:*
  + Chọn i, j lần lượt nằm ở đầu với cuối mảng phân hoạch (i=0, j=7).
  + i = 0, phần tử tại vị trí i có giá trị nhỏ hơn pivot, tăng i (i=1), khi này phần tử tại vị trí i có giá trị lớn hơn pivot, dừng lại.
  + j = 7, phần tử tại vị trí j có giá trị nhỏ hơn pivot, dừng lại.
  + i < j, hoán vị 2 giá trị tại vị trí i và j, tăng i (i=2) và giảm j (j=6).

		i		pivot				j
	5	9	2	7	4	6	8	1
index:	0	1	2	3	4	5	6	7

			i	pivot			j	
	5	1	2	7	4	6	8	9
index:	0	1	2	3	4	5	6	7

  + do i < j nên tiếp tục chạy, i tăng đến pivot thì dừng (i=3).
  + phần tử tại vị trí j có giá trị lớn hơn pivot, giảm j cho đến khi j = 5, phần tử tại vị trí j nhỏ hơn pivot nên dừng lại.
  + i < j, hoán vị 2 giá trị tại tại vị trí i và j, tăng i (i=4) và giảm j (j=4).

				pivot = i		j		
	5	1	2	7	4	6	8	9
index:	0	1	2	3	4	5	6	7

					i = j	pivot		
	5	9	2	6	4	7	8	9
index:	0	1	2	3	4	5	6	7

  + i = j nên kết thúc lần phân hoạch đầu tiên, lúc này mảng đã phân hoạch được 2 mảng con (cách nhau bởi pivot = 7).

- *Phân hoạch lần hai với mảng bên trái, pivot = 2:*

	i		pivot		j			
	5	9	2	6	4	7	8	9
index:	0	1	2	3	4	5	6	7

  + Chọn i, j lần lượt nằm ở đầu với cuối mảng phân hoạch (i=0, j=4).
  + i = 0, phần tử tại vị trí i có giá trị lớn hơn pivot, dừng lại.

+ j = 4, phần tử tại vị trí j có giá trị lớn hơn pivot, giảm j (j=3), tiếp tục thỏa mãn, giảm tiếp cho đến khi j = pivot = 2 thì dừng lại.
+ i < j, hoán vị 2 giá trị tại vị trí i và j, tăng i (i=1), giảm j (j=1).

	i		pivot = j		j			
	5	9	2	6	4	7	8	9
index:	0	1	2	3	4	5	6	7

	pivot	i=j						
	2	9	5	6	4	7	8	9
index:	0	1	2	3	4	5	6	7

+ i = j, kết thúc lần phân hoạch thứ hai

- *Phân hoạch lần ba với pivot = 5:*

		i	pivot		j			
	2	9	5	6	4	7	8	9
index:	0	1	2	3	4	5	6	7

+ Chọn i, j lần lượt nằm ở đầu với cuối mảng phân hoạch (i=1, j=4).
+ i = 1, phần tử tại vị trí i có giá trị lớn hơn pivot, dừng lại.
+ j = 4, phần tử tại vị trí j có giá trị nhỏ hơn pivot, dừng lại.

		i	pivot		j			
	2	9	5	6	4	7	8	9
index:	0	1	2	3	4	5	6	7

+ i < j, hoán vị 2 giá trị tại vị trí i và j, tăng I (i=2), giảm j (j=3).

			pivot =i	j				
	2	4	5	6	9	7	8	9
index:	0	1	2	3	4	5	6	7

+ j=3, phần tử tại vị trí đó có giá trị lớn hơn pivot, giảm j (j=2), giá trị tại j bằng pivot, dừng lại.

			pivot = i = j					
	2	4	5	6	9	7	8	9
index:	0	1	2	3	4	5	6	7

+ i = j, kết thúc lần phân hoạch thứ ba.

- *Phân hoạch lần bốn với pivot = 6:*

				pivot = i	j			
	2	4	5	6	9	7	8	9
index:	0	1	2	3	4	5	6	7

+ Chọn i, j lần lượt nằm ở đầu với cuối mảng phân hoạch (i=3, j=4).
+ j = 4, phần tử tại vị trí đó có giá trị lớn pivot, giảm j (j=3), lúc này vị trí của pivot = i = j, dừng lại.
+ i = j, kết thúc lần phân hoạch thứ bốn.

- *Phân hoạch lần năm với mảng bên phải ở lần phân hoạch một, tương tự ta có :*

						i	pivot	j
	2	4	5	6	9	7	8	9
index:	0	1	2	3	4	5	6	7

							pivot =i = j	
	2	4	5	6	9	7	8	9
index:	0	1	2	3	4	5	6	7

- **Sau khi thực hiện các bước trên, mảng đã được sắp xếp theo thứ tự tăng dần.**

3.8.3. Complexity evaluations

- **Time complexity:**
  1. Trường hợp tốt nhất: $O(n\log(n))$
     - Khi phần tử pivot là phần tử chia mảng thành 2 phần, trong đó chi phí khi tìm pivot và phân hoạch là $\log_2(n)$, số lần phân hoạch là n. Do đó độ phức tạp của nó xấp xỉ là $O(n\log(n))$
  2. Trường hợp xấu nhất: $O(n^2)$
     - Xảy ra khi phần tử pivot là phần tử nhỏ nhất hoặc lớn nhất (hoặc vị trí đặc biệt khác) và mảng đang xét đã được sắp xếp. Một trường hợp nữa là tất cả phần tử của mảng bằng nhau và pivot ở đầu hoặc cuối mảng.
     - Khi đó, ta nhận thấy rằng lần phân hoạch thứ i cần phải tốn chi phí là $O(n-i)$ vì nó phải duyệt qua $n-i$ phần tử, vì vậy chi phí cả thuật toán là $\sum_{i=0}^{n}(n-i)$ và nó xấp xỉ $O(n^2)$
  3. Trường hợp trung bình: $O(n\log(n))$
- **Space complexity:**

1. Nếu trong trường hợp xấu nhất của time complexity, ta phải phân hoạch n lần thì cũng phải tạo n lần pivot, khi đó space complexity là O(n).
2. Trường hợp trung bình và tốt nhất là O(log(n)).

3.8.4. Variants/improvements

- **Phân hoạch Lomuto:**
  1. Mã giả:

```
Partition(a, p, r)
x = a[r]
for j = p to r – 1
        if a[j] <= x
    i = i + 1
            exchange a[i] with a[j]
exchange a[i+1] with a[r]
return i + 1
```

  2. Ý tưởng

- Phân hoạch Lomuto dùng phần tử ngoài cùng của mảng làm pivot, dùng một biến j để chạy từ trái đến trước phần tử pivot, nếu a[j] <= x thì ta tăng i và hoán vị a[i] và a[j], cứ làm thế cho đến khi chạy hết j, cuối cùng ta hoán vị a[i+1] và pivot.
- Ở thuật toán này, mảng được chia ra làm 4 phần: [p,i] bao gồm các phần tử <=x, [i,j] bao gồm các phần tử >x, [j,r] bao gồm các phần tử chưa xác định tính thứ tự và a[r]. Thuật toán trên về cơ bản là ta sẽ đẩy phần >x lùi về phía a[r], cuối cùng hoán vị a[r] với phần tử sau phần tử ngoài cùng của phần <=x, trả về vị trí đó. Mảng được phân ra theo tư tưởng của quicksort.

3. Complexity evaluations
   - Tương tự như thuật toán quicksort ban đầu.

- **Phân hoạch Hoare**

1. Mã giả

```
Partition(a, p, r)
x = a[p]
i = p - 1
j = r + 1
while true
  do
    j = j - 1
  while a[j] > x
  do
    i = i + 1
  while a[i] < x
  if i < j
    exchange a[i] with a[j]
  else
    return j
```

2. Ý tưởng
   - Phân hoạch Hoare dùng phần tử trái cùng làm pivot, ý tưởng gần giống với quicksort ban đầu ta xét, khác là ta sẽ trả về vị trí j sau khi phân hoạch, lúc này mảng sẽ phân hoạch thành 2 mảng nhỏ.
3. Complexity evaluations
   - Tương tự như thuật toán quicksort ban đầu.

3.9. Counting Sort

3.9.1. Ideas

- Đầu tiên, tìm phần tử lớn nhất trong mảng **A** (gọi là max), sau đó tạo một mảng **C** có $max + 1$ phần tử dùng để **lưu số lượng của từng phần tử.**
- Tiếp theo, ta sẽ lưu số lượng của từng phần tử từ **A** vào **C**. Trong đó, với mỗi phần tử thì:
  + *Vị trí trong **C**: Giá trị trong **A**.*

+ *Giá trị trong **C**: Số lượng phần tử có cùng giá trị đó trong **A**.*

- Từ đây, để có thể thực hiện sắp xếp với độ phức tạp nhỏ nhất, ta phải thực hiện thêm một bước nữa gọi là *"dàn mảng"*.
    + *Bước này sẽ chuyển đổi mảng **C** từ "lưu **số-lượng** của từng phần tử" thành "lưu **vị-trí-đã-sắp-xếp** của từng phần tử".*
    + *Để thực hiện việc này, ta cần tăng giá trị của một phần tử trong mảng **C** với một lượng bằng với giá trị của phần tử đứng trước nó trong **C**.*

      ***Giải thích:** Giả sử mảng **A** có 3 phần tử là **4**, **5** và **6**. Thì phần tử **5** phải đứng sau tất cả các phần tử **4**, và phần tử **6** phải đứng sau tất cả các phần tử **5**. Do đó, vị trí đầu tiên của các phần tử **5** sẽ bắt đầu sau một khoảng **{số lượng phần tử 4}**.*
- Cuối cùng, ta sẽ tạo một mảng **B** để lưu các phần tử đã được sắp xếp. Trong đó, với mỗi phần tử thì:
    + *Vị trí trong **B**: Giá trị trong **C** trừ đi **1**.*
    + *Giá trị trong **B**: Giá trị trong **A**.*

3.9.2. Step-by-step descriptions

- **Cho một mảng 7 phần tử như bên dưới, lúc này max sẽ bằng 4:**

4	2	1	2	3	1	2

- **Tạo mảng C có 5 phần tử 0:**

4	2	1	2	3	1	2

0	1	2	3	4
0	0	0	0	0

- **Lưu số lượng của phần tử đầu tiên:**

4	2	1	2	3	1	2

0	1	2	3	4
0	0	0	0	1

- **Lưu số lượng của phần tử đầu thứ hai:**

4	2	1	2	3	1	2

0	1	2	3	4
0	0	1	0	1

- **Lưu số lượng của phần tử thứ ba:**

4	2	1	2	3	1	2

0	1	2	3	4
0	1	1	0	1

- **Lưu số lượng của phần tử thứ tư:**

4	2	1	2	3	1	2

0	1	2	3	4
0	1	2	0	1

- **Lưu số lượng của phần tử thứ năm:**

4	2	1	2	3	1	2

0	1	2	3	4
0	1	2	1	1

- **Lưu số lượng của phần tử thứ sáu:**

4	2	1	2	3	1	2

0	1	2	3	4
0	2	2	1	1

- Lưu số lượng của phần tử thứ bảy:

4	2	1	2	3	1	2

0	1	2	3	4
0	2	3	1	1

- Thực hiện “dàn mảng” C:

4	2	1	2	3	1	2

0	1	2	3	4
0	2	5	6	7

- Tạo mảng B có 7 phần tử:

4	2	1	2	3	1	2

0	1	2	3	4
0	2	5	6	7

0	1	2	3	4	5	6

- Sắp xếp phần tử đầu tiên:

4	2	1	2	3	1	2

0	1	2	3	4
0	2	5	6	6

0	1	2	3	4	5	6
						4

- Sắp xếp phần tử thứ hai:

4	2	1	2	3	1	2

0	1	2	3	4
0	2	4	6	6

0	1	2	3	4	5	6
				2		4

- Sắp xếp phần tử thứ ba:

4	2	1	2	3	1	2

0	1	2	3	4
0	1	4	6	6

0	1	2	3	4	5	6
	1			2		4

- Sắp xếp phần tử thứ tư:

4	2	1	2	3	1	2

0	1	2	3	4
0	1	3	6	6

0	1	2	3	4	5	6
	1		2	2		4

- Sắp xếp phần tử thứ năm:

4	2	1	2	3	1	2

0	1	2	3	4
0	1	3	5	6

0	1	2	3	4	5	6
	1		2	2	3	4

- **Sắp xếp phần tử thứ sáu:**

4	2	1	2	3	1	2

0	1	2	3	4
0	0	3	5	6

0	1	2	3	4	5	6
1	1		2	2	3	4

- **Sắp xếp phần tử cuối cùng:**

4	2	1	2	3	1	2

0	1	2	3	4
0	0	2	6	6

0	1	2	3	4	5	6
1	1	2	2	2	3	4

- **Từ đây, ta nhận được mảng B hoàn chỉnh - mảng đã được sắp xếp:**

1	1	2	2	2	3	4

3.9.3. Complexity evaluations

- **Time complexity:**

Average case, best case, worst case: $\boldsymbol{O(n+k)}$

Giải thích:

+ Trong vòng lặp đầu tiên, ta thực hiện $\boldsymbol{n}$ (số lượng phần tử của **A**) phép so sánh.
+ Trong vòng lặp thứ hai, ta thực hiện $\boldsymbol{k}$ (số lượng phần tử của **C**) phép so sánh.
+ Trong vòng lặp cuối cùng, ta thực hiện $\boldsymbol{n}$ (số lượng phần tử của **A**) phép so sánh.
+ Vậy, sau **3** lần hoán vị, ta thực hiện tổng cộng:

$$\boldsymbol{O(n)+O(k)+O(n)\ \sim\ O(n+k)}$$

(với mọi trường hợp, ta đều phải thực hiện một lượng so sánh tương đương như trên)

- **Space complexity:**

Average case, best case, worst case: $\boldsymbol{O(n+k)}$

Giải thích: Ta cần tạo tổng cộng 2 mảng **C** và **B**:

+ Mảng **C** có $\boldsymbol{k}$ phần tử, dẫn đến tiêu thụ $\boldsymbol{k}$ "không gian".
+ Mảng **B** có $\boldsymbol{n}$ phần tử, dẫn đến tiêu thụ $\boldsymbol{n}$ "không gian".
+ Vây, ta cần tổng cộng số "không gian" là:

$$\boldsymbol{O(k)+O(n)\ \sim\ O(n+k)}$$

(với mọi trường hợp, ta đều phải thực hiện một lượng "không gian" tương đương như trên)

3.10. Radix Sort

3.10.1. Ideas

Ý tưởng

- Tìm số lớn nhất của mảng (mục đích là tìm số chữ số nhiều nhất của một số trong mảng).
- Phân loại các số này theo chữ số hàng đơn vị, hàng chục, hàng trăm,… (cho đến vị trí đầu tiên của phần tử lớn nhất tìm được ở trên), lấy các phần tử ra và sắp chúng theo thứ tự từ cơ số 0 đến 9 . Lặp lại bước sắp xếp này cho đến khi duyệt đến hết vị trí đầu tiên của phần tử lớn nhất.

Thuật toán

- Tìm số lớn nhất của mảng (mảng a[n]).
- Tạo biến đếm exp = 1 và mảng đếm (output[n]) lưu giá trị của mảng sau khi lấy ra theo thứ tự cơ số.
- Lặp khi (số lớn nhất / exp > 0)
  1. exp *= 10. (mỗi vòng lặp xét tiếp 1 chữ số hàng tiếp theo)
  2. Tạo mảng vector có kích thước là 10.
  3. Duyệt mảng, lưu phần tử của mảng phù hợp với vị trí cơ số của hàng đang xét vào vector.
  4. Duyệt mảng vector từ 0 đến 9 và lưu giá trị theo thứ tự vào mảng ban đầu.
- Sau khi hoành thành vòng lặp, mảng đã được sắp xếp tăng dần

3.10.2. Step-by-step descriptions

- Tạo mảng gồm 7 phần tử như hình dưới, với số lớn nhất là số có 4 chữ số, do đó ta sẽ duyệt 4 lần theo cơ số của từng hàng: đơn vị, chục, trăm, nghìn.
- Vòng lặp thứ nhất, chữ số hàng đơn vị:
  + Tạo mảng vector như hình dưới, lần lượt đưa mỗi số của mảng vào vị trí thích hợp.

120	312	1193	2614	70	782	1109

cơ số		
0	120	70
1		
2	312	782
3	1193	
4	2614	
5		
6		
7		
8		
9	1109	

+ Lần lượt lấy các số ra theo thứ tự từ trên xuống

	120	312	1193	2614	70	782	1109
Lần 1	120	70	312	782	1193	2614	1109

cơ số		
0	120	70
1		
2	312	782
3	1193	
4	2614	
5		
6		
7		
8		
9	1109	

- Vòng lặp thứ hai, chữ số hàng chục:
  + Tiếp tục tạo mảng vector và lưu vào vị trí thích hợp.

	120	312	1193	2614	70	782	1109
Lần 1	120	70	312	782	1193	2614	1109
Lần 2							

cơ số		
0	1109	
1	312	2614
2	120	
3		
4		
5		
6		
7	70	
8	782	
9	1193	

  + Lần lượt lấy các số ra theo thứ tự từ trên xuống.

	120	312	1193	2614	70	782	1109
Lần 1	120	70	312	782	1193	2614	1109
Lần 2	1109	312	2614	120	70	782	1193

cơ số		
0	1109	
1	312	2614
2	120	
3		
4		
5		
6		
7	70	
8	782	
9	1193	

- Vòng lặp thứ 3, chữ số hàng trăm:
  + Tiếp tục tạo mảng vector và lưu vào vị trí thích hợp.

	120	312	1193	2614	70	782	1109
Lần 1	120	70	312	782	1193	2614	1109
Lần 2	1109	312	2614	120	70	782	1193
Lần 3							

cơ số			
0	70		
1	1109	120	1193
2			
3	312		
4			
5			
6	2614		
7	782		
8			
9			

+ Lần lượt lấy các số ra theo thứ tự từ trên xuống.

	120	312	1193	2614	70	782	1109
Lần 1	120	70	312	782	1193	2614	1109
Lần 2	1109	312	2614	120	70	782	1193
Lần 3	70	1109	120	1193	312	2614	782
Lần 4							

cơ số			
0	70		
1	1109	120	1193
2			
3	312		
4			
5			
6	2614		
7	782		
8			
9			

+ Nhận xét: Lúc này số có 2 chữ số đã xếp đúng thứ tự.

- Vòng lặp cuối, chữ số hàng nghìn:
  + Tiếp tục tạo mảng vector và lưu vào vị trí thích hợp.

	120	312	1193	2614	70	782	1109
Lần 1	120	70	312	782	1193	2614	1109
Lần 2	1109	312	2614	120	70	782	1193
Lần 3	70	1109	120	1193	312	2614	782
Lần 4							
cơ số							
0	70	120	312	782			
1	1109	1193					
2	2614						
3							
4							
5							
6							
7							
8							
9							

+ Lần lượt lấy các số ra theo thứ tự từ trên xuống.

	120	312	1193	2614	70	782	1109
Lần 1	120	70	312	782	1193	2614	1109
Lần 2	1109	312	2614	120	70	782	1193
Lần 3	70	1109	120	1193	312	2614	782
Lần 4	70	120	312	782	1109	1193	2614
cơ số							
0	70	120	312	782			
1	1109	1193					
2	2614						
3							
4							
5							
6							
7							
8							
9							

- Mảng được sắp xếp tăng dần sau 4 lần lặp

	120	312	1193	2614	70	782	1109
Lần 1	120	70	312	782	1193	2614	1109
Lần 2	1109	312	2614	120	70	782	1193
Lần 3	70	1109	120	1193	312	2614	782
Lần 4	70	120	312	782	1109	1193	2614

3.10.3. Complexity evaluations

- **Time complexity:**
  1. Trường hợp trung bình: **O(x*(n+y))**
     - Với x là số lượng chữ số lớn nhất của phần tử lớn nhất trong mảng, n là số phần tử, y là hệ đếm đang dùng (thường y = 10).
  2. Trường hợp tốt nhất: **O(x*n)**
     - Khi b = O(n).
  3. Trường hợp xấu nhất: **O(n^2)**
     - Khi x >= n ⇔ số lượng chữ số của phần tử lớn nhất lớn hơn số phần tử của mảng. Lấy trung bình là O(n^2).

- **Space complexity: O(x*n)**
  - Vì mỗi lần lần lặp ta phải tạo 1 mảng vector chiếm bộ nhớ bằng số phần tử của mảng cần xét, nên chi phí tạo mảng này mỗi lần lặp là O(n), đồng thời ta phải lặp qua x lần (x là số lượng chữ số lớn nhất của phần tử lớn nhất trong mảng), do đó tổng chi phí không gian của thuật toán là O(x*n)

3.10.4. Variants/improvements

- **Radixsort không dùng vector:** Thay vì tạo mảng vector để lưu các giá trị mỗi lần thuật hiện radix sort, ta có thể dùng một mảng con count[10] (count[i] đại diện cho cơ số i trong vòng lặp đang xét). Cộng dồn mảng count -> count[i] đại diện cho toàn bộ phần tử trước i. Kết hợp tạo thêm 1 mảng output để bắt đầu thực hiện thuật toán radix. Cuối cùng gán lại output cho mảng ban đầu. (xem code trong file src). Về time complexity thì không tăng lên, nhưng về space complexity tăng lên vì phải dùng 2 mảng con (cụ thể là O(x*(10+n))).
- **Radixsort cho mảng chuỗi với độ dài như nhau:** Ta có thể dùng radix sort cho 1 mảng chuỗi nhưng với điều kiện là độ dại của các chuỗi như nhau, ý tưởng như radix sort cho số, xem code ở file src. (Lưu ý: tùy vào người sử dụng, nếu các mảng có độ dài không bằng nhau thì có thể cộng chuỗi cho kí tự 0 hay kí tự nào đó để cho các chuỗi bằng nhau)

3.11. Flash Sort

3.11.1. Ideas

Ý tưởng:

- Tạo những "thùng chứa" để chứa những phần tử vào vị trí phù hợp của các mảng con. Sau đó dùng phép hoán vị để đưa chúng vào đúng vị trí sắp xếp của mình.

Giải thuật:

- Tìm số lượng "thùng chứa" để chia các phần tử của mảng, dùng một mảng con với giá trị của nó là số giá trị mà "thùng" chứa (số lượng mảng con = m = 0.43 * số phần tử của mảng). Đánh dấu các mảng con từ 0 -> m – 1.

- Quyết định xem phần tử nào của mảng thuộc "thùng chứa" nào bằng công thức (k = (m-1)*(a[i] – min)/(max – min)).

- Cộng dồn giá trị của mảng, khi đó "thùng chứa" thứ l mang giá trị đại diện cho tất cả giá trị của mảng 0 đến l.

- Tạo biến nmove để xác định khi nào ngừng lặp, nmove lặp n-1 lần với n là số phần tử của mảng., biến k để đại diện cho "thùng chứa" thứ k, biến j là vị trí của phần tử dùng để xét xem nó đang xét ở "thùng" nào, flash là phần tử đầu tiên.

- Hoán vị phần tử lớn nhất để đưa nó về đầu mảng và bắt đầu quá trình sắp xếp của thuật toán flash:

1. Xét vòng while, khi j > l[k] – 1 thì ta tăng j cho đến khi nào nó gặp phần tử thuộc bucket tiếp theo.

2. Khi j khác l[k], đưa flash về đúng chổ, đồng thời flash sẽ giữ phần tử cho lần hoán vị tiếp theo của nó. Tăng biến nmove.

- Sau khi hoàn thành, có thể sẽ còn một số ít các phân tử chưa sắp xếp đúng vị trí, do đó cần dùng một thuật toán có chi phí thấp để xử lý trường hợp này, ở đây ta chọn insertion sort.

3.11.2. Step-by-step descriptions

- Đầu tiên, ta tạo mảng 5 phần tử và hoàn thành các bước trước khi xét vòng lặp:

-1	13	-18	3	2
13	-1	-18	3	2

minVal = -18
maxIdx = 2

nmove = 0
j = 0
k = 1
flash = 13

index	0	1
1	4	5

- Bắt đầu vòng lặp:

+ Vòng lặp đầu tiên, k đang chỉ thùng thứ nhất (l[1]) ta hoán vì flash và a[--l[k]] , khi đó flash sẽ giữ giá trị bị hoán vị, 13 đã về đúng thùng của nó.

-1	13	-18	3	2	minVal = -18
13	-1	-18	3	2	maxIdx = 2
13	-1	-18	3	13	

nmove = 1
j = 0
k = 1
flash = 2

index	0	1
1	4	4

+ Tiếp tục chạy vòng lặp vì j khác l[k], lúc này ta tính được k = 0 (thùng 1 chứa 1 phần tử và đã được sắp xếp đúng chỗ)

-1	13	-18	3	2	minVal = -18
13	-1	-18	3	2	maxIdx = 2
13	-1	-18	3	13	
13	-1	-18	2	13	

nmove = 2
j = 0
k = 0
flash = 3

index	0	1
1	3	4

+ Chạy tiếp các vòng lặp tiếp theo

	-1	13	-18	3	2	minVal = -18
	13	-1	-18	3	2	maxIdx = 2
	13	-1	-18	3	13	
	13	-1	-18	2	13	
nmove = 3	13	-1	3	2	13	
j = 0						
k = 0						
flash = -18						

index	0	1
1	2	4

	-1	13	-18	3	2	minVal = -18
	13	-1	-18	3	2	maxIdx = 2
	13	-1	-18	3	13	
	13	-1	-18	2	13	
nmove = 4	13	-1	3	2	13	
j = 0	13	-18	3	2	13	
k = 0						
flash = -1						

index	0	1
1	1	4

	-1	13	-18	3	2	minVal = -18
	13	-1	-18	3	2	maxIdx = 2
	13	-1	-18	3	13	
	13	-1	-18	2	13	
nmove = 5	13	-1	3	2	13	
j = 0	-1	-18	3	2	13	
k = 0						
flash = 13						

index	0	1
1	0	4

- Lúc này, các phần tử của thùng 1 đã đúng vị trí, ta cần dùng 1 thuật toán đơn giản (là insertion sort) để sắp xếp lại thùng 0. Thuật toán kết thúc, mảng đã được sắp xếp.

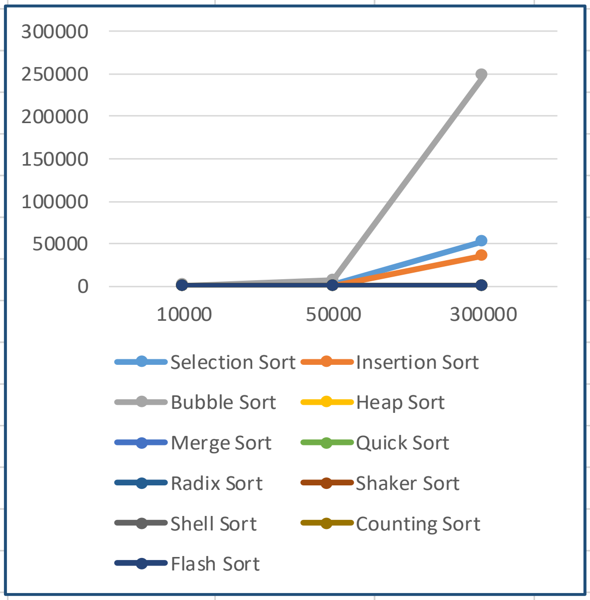
3.11.3. Complexity evaluations

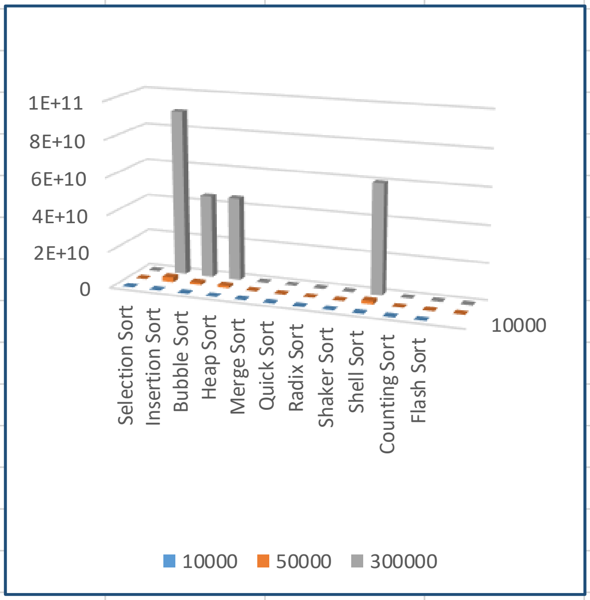
- **Time complexity:** O(n^2/m), người ta chứng minh được rằng m = 0.43*n là trường hợp tốt nhất cho thuật toán.

- **Space complexity**: O(m)

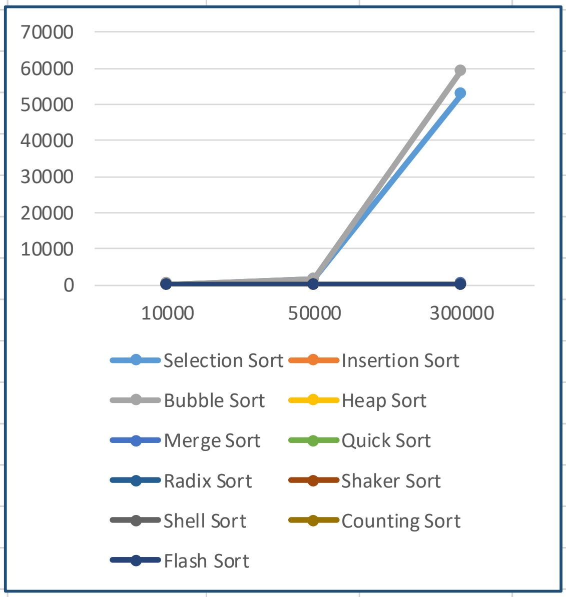
4. Experimental results and comments

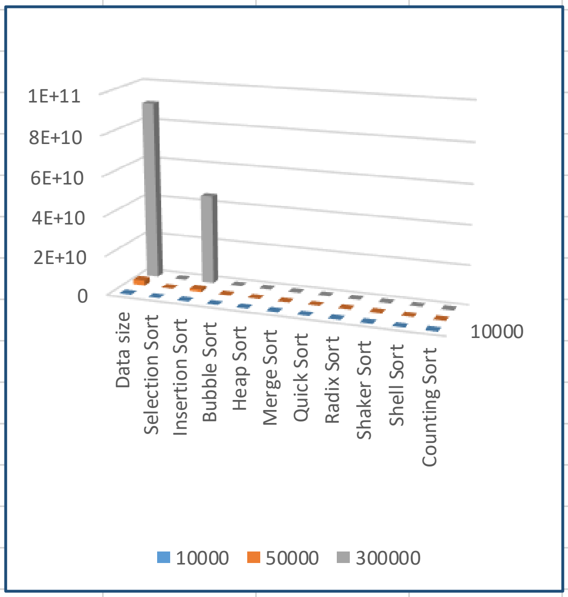
Data order: Random						
Data size	**10000**	**10000**	**50000**	**50000**	**300000**	**300000**
Resulting statics	Running time	Comparision	Running time	Comparision	Running time	Comparision
Selection Sort	56	100009999	1600	2500049999	51911	90000299999
Insertion Sort	38	50408307	920	1250358109	35350	44937924304
Bubble Sort	209	50005000	6295	1250025000	248036	45000150000
Heap Sort	1	637584	10	3771780	74	26489690
Merge Sort	3	701503	11	4092319	67	28401744
Quick Sort	1	282786	4	1711602	28	11502730
Radix Sort	1	100092	4	600115	26	3600115
Shaker Sort	1	67055288	1	1657621815	1	60056944571
Shell Sort	1	643055	1	4417925	1	34533711
Counting Sort	0	70002	0	298306	3	1298306
Flash Sort	0	146331	3	1127731	18	11324250



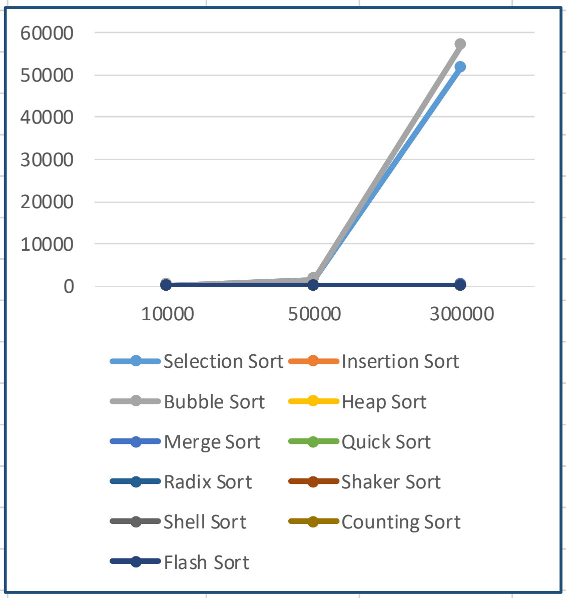


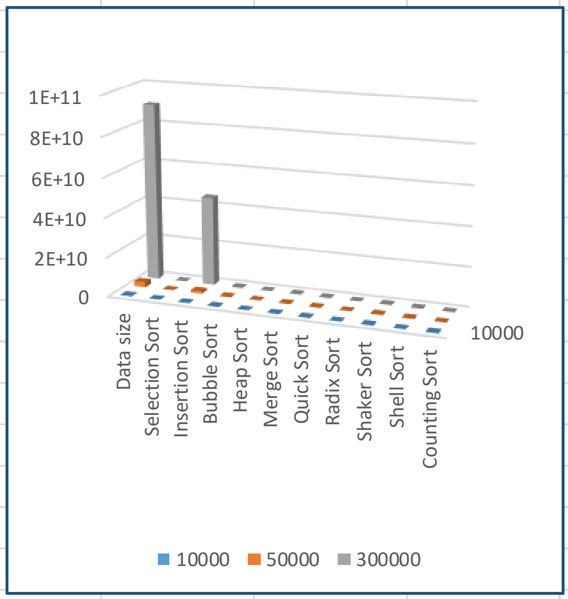
Data order: Nearly Sorted						
Data size	**10000**	**10000**	**50000**	**50000**	**300000**	**300000**
Resulting statics	Running time	Comparision	Running time	Comparision	Running time	Comparision
Selection Sort	56	100009999	1570	2500049999	52621	90000299999
Insertion Sort	1	202122	0	563290	1	1320342
Bubble Sort	65	50005000	1497	1250025000	59140	45000150000
Heap Sort	1	670000	9	3925535	60	27413277
Merge Sort	1	475242	9	2722826	42	18867274
Quick Sort	1	193610	1	1084458	8	7275750
Radix Sort	0	100092	4	600115	27	4200138
Shaker Sort	1	181509	1	629147	1	1054552
Shell Sort	1	419558	1	2262095	1	15433395
Counting Sort	0	70002	0	350002	3	2100002
Flash Sort	0	118532	2	592935	8	3557935



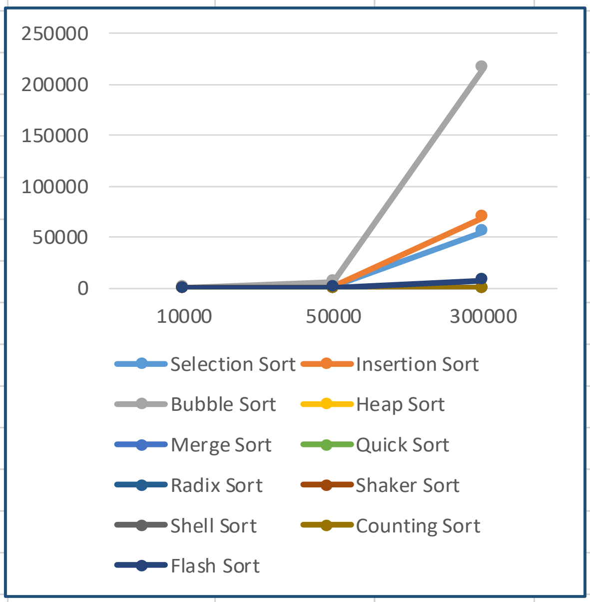


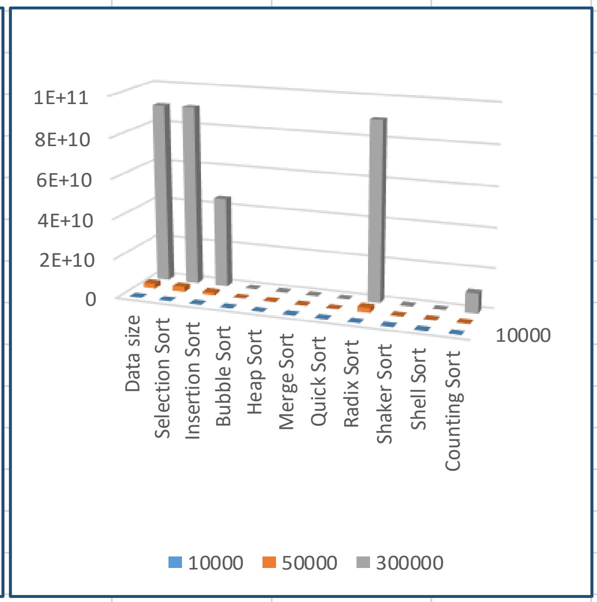
Data order: Sorted						
Data size	**10000**	**10000**	**50000**	**50000**	**300000**	**300000**
Resulting statics	Running time	Comparision	Running time	Comparision	Running time	Comparision
Selection Sort	55	100009999	1416	2500049999	51793	90000299999
Insertion Sort	0	29998	0	149998	1	899998
Bubble Sort	63	50005000	1538	1250025000	56954	45000150000
Heap Sort	2	670329	9	3925351	59	27413230
Merge Sort	2	475242	13	2722826	44	18645946
Quick Sort	0	193610	1	1084458	7	7275706
Radix Sort	1	100092	5	600115	25	4200138
Shaker Sort	1	20002	1	100002	1	600002
Shell Sort	1	360042	1	2100049	1	15300061
Counting Sort	0	70002	1	350002	2	2100002
Flash Sort	0	118597	2	592997	9	3557997





Data order: Reversed						
Data size	**10000**	**10000**	**50000**	**50000**	**300000**	**300000**
Resulting statics	Running time	Comparision	Running time	Comparision	Running time	Comparision
Selection Sort	61	100009999	1483	2500049999	55198	90000299999
Insertion Sort	75	100009999	1884	2500049999	69764	90000299999
Bubble Sort	245	50005000	5788	1250025000	215996	45000150000
Heap Sort	1	606771	10	3612724	56	25569379
Merge Sort	2	475242	19	2722826	43	24065209
Quick Sort	1	203607	1	1134455	9	7275706
Radix Sort	1	100092	5	600115	26	4200138
Shaker Sort	1	100005001	1	2500025001	1	90000150001
Shell Sort	1	475175	1	2844628	1	20001852
Counting Sort	0	70002	1	350002	2	2100002
Flash Sort	9	11184622	201	278234058	7538	10002870997



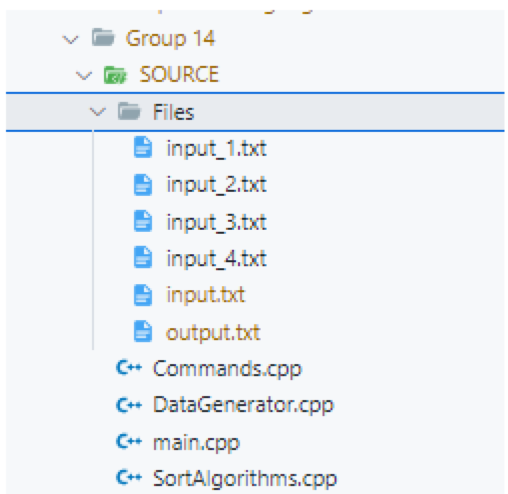


- **Comments:**
  1. Chưa thể chắc chắn được thuật toán nào chạy nhanh nhất được vì mỗi thuật toán lại có thể chạy chậm vào tùy dạng kiểu dữ liệu ban đầu, tuy nhiên có một điều chắc chắn rằng bubble sort là thuật toán kém hiệu quả nhất (về mọi mặt), dựa vào đồ thị và bảng số liệu ta có thể chắc chắn đều này.
  2. Nếu chỉ xét riêng về phép so sánh (comparisions) thì nhóm thuật toán: counting sort, radix sort (có thể tính thêm flash sort) là sử dụng ít phép so sánh hơn các thuật toán còn lại.

3. Chia thuật toán thành 2 dạng "ổn định" và "không ổn định" dựa vào runtime và comparision trong bảng số liệu ở phần trên (đã được thể hiện bằng màu sắc), ngoài ra có thể tham khảo thêm ở phần chi tiết của những thuật toán này.
    a. Nhóm thuật toán ổn định:
        i. Selection sort
        ii. Shell sort
        iii. Heap sort
        iv. Merge sort
        v. Radix sort
        vi. Flash sort
    b. Nhóm thuật toán "không ổn định"
        i. Insertion sort
        ii. Bubble sort
        iii. Shaker sort
        iv. Quick sort
        v. Counting sort

5. Project organization and Programming notes

5.1 Project organization



- Toàn bộ file code nằm trong folder SOURCE.
- Folder Files bao gồm 6 file txt, trong đó file input.txt là dữ liệu dùng để chạy cmd 1 và cmd 4 đồng thời để lưu dữ liệu khi chạy cmd 2 và cmd 5 và output.txt là dữ liệu dùng để chứa những dữ liệu được dùng trong cmd 1 và cmd 2. File input_1.txt, input_2.txt, input_3.txt, input_4.txt dùng để lưu dữ liệu tương ứng với 4 generated input : random, nearly sorted, sorted, reversed.
- Commands.cpp là file code chứ code 5 cmd đề yêu cầu và một số hàm phụ trợ khác.
- DataGenerator.cpp là file chứa những kiểu dữ liệu để chạy, một số hàm phụ trợ cần thiết khác.

- SortAlgorithms.cpp là file chứa những thuật toán sort, trong đó mỗi thuật toán có 2 hàm thuật toán như nhau nhưng một trong chúng để đếm phép so sánh, còn lại là một hàm bình thường dùng để đo thời gian (cách đo thì được trình bày bằng code trng hàm Commands.cpp).
- main.cpp là file chứa hàm main chính của toàn bộ chương trình.

5.2 Programming notes

- Nhằm để thuận tiện đo thời gian, nhóm em quyết định viết hàm đo thời gian vào trong một hàm phụ trợ trong file Commands.cpp mà không viết hàm đo riêng mỗi thuật toán sort. Đồng thời đơn vị đo thời gian sử dụng là ms (có làm tròn), do đó một số hàm chạy nhanh có thể làm tròn thành 0ms.
- Ngoài ra nhóm em còn sử dụng nhiều thư viện phụ trợ khác như fstream, cstring,…

6. References and citations

1. Bài giảng lý thuyết – thầy Cao Xuân Nam.
2. Lab 2 & 3 – tài liệu thực hành – thầy Lê Đình Ngọc.
3. Slide bài giảng - Văn Chí Nam, Nguyễn Thị Hồng Nhung, Đặng Nguyễn Đức Tiến. (https://drive.google.com/drive/folders/1LQGkI_0pg9Fsoun0y4JluunTehBgC9wk)
4. QuickSort - GeeksforGeeks
5. www.geeksforgeeks.org/radix-sort/
6. Thuật Toán Sắp Xếp Nhanh QuickSort | Phân Hoạch Lomuto Và Phân Hoạch Hoare - YouTube
7. Flash Sort thuật toán sắp xếp với độ phức tạp O(n) ? - YouTube
8. https://viettuts.vn/cau-truc-du-lieu-va-giai-thuat/giai-thuat-sap-xep-shell-sort
9. https://tek4.vn/khoa-hoc/cau-truc-du-lieu-va-giai-thuat/thuat-toan-sap-xep-vun-dong-heap-sort
10. Selection Sort Algorithm - GeeksforGeeks
11. Bubble Sort Algorithm - GeeksforGeeks
12. Insertion Sort - GeeksforGeeks
13. Merge Sort Algorithm - GeeksforGeeks
14. Heap Sort - GeeksforGeeks
15. Counting Sort - GeeksforGeeks
16. Radix Sort - GeeksforGeekss