



TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG



Lập trình C nâng cao

Nguyễn Khánh Phương

Bộ môn Khoa học máy tính
Viện Công nghệ thông tin và truyền thông
E-mail: phuongnk@soict.hust.edu.vn

Chủ đề môn học

- Lập trình C trên môi trường UNIX
- Các chủ đề lập trình liên quan đến cấu trúc dữ liệu và giải thuật về sắp xếp và tìm kiếm nâng cao.
- Trình biên dịch (Compiler): gcc
<http://www.network-theory.co.uk/docs/gccintro/>
- Trình soạn thảo (Editor): Emacs, K-Developer.

gcc syntax

- Các thông số:
 - Wall : turn on all alerts
 - c: make object file
 - o: name of output file
 - g: debug information
 - l: library

- Ví dụ:

```
gcc -Wall hello.c
```

→ Mặc định tạo ra executable file là **a.out** → chạy chương trình: **./a**

```
gcc -Wall hello.c -o runhello
```

→ Sử dụng tùy chọn **-o** → tạo executable file khác với tên mặc định, và đặt tên là runhello (→ executable file là **runhello.out**)

→ Chạy chương trình:

```
./runhello
```

Nội dung

Phần 1: Sắp xếp (Sorting)

Phần 2: Cây (Tree)

Phần 3: Đồ thị (Graph)

Nội dung

Phần 1: Sắp xếp (Sorting)

Phần 2: Cây (Tree)

Phần 3: Đồ thị (Graph)

Bài toán sắp xếp

- Sắp xếp (Sorting)
 - Là quá trình tổ chức lại họ các dữ liệu theo thứ tự giảm dần hoặc tăng dần (ascending or descending order)
- Dữ liệu cần sắp xếp có thể là
 - Số nguyên (Integers)
 - Xâu ký tự (Character strings)
 - Đối tượng (Objects)
- Khoá sắp xếp (Sort key)
 - Là bộ phận của bản ghi xác định thứ tự sắp xếp của bản ghi trong họ các bản ghi.
 - Ta cần sắp xếp các bản ghi theo thứ tự của các khoá.

```
typedef struct thisinh {  
    char hoten[30];  
    float Cintro, Cbasic, Cadvanced, tong;  
}thisinh;  
thisinh VNK63[100];
```

Ứng dụng của sắp xếp

D. Knuth: 40% thời gian hoạt động của các máy tính là dành cho sắp xếp!

- Quản lý và tìm kiếm thông tin: dữ liệu đã sắp xếp sẽ cho phép quản lý và tìm kiếm thông tin hiệu quả và nhanh chóng hơn

Ví dụ: Điện hình nhất trong thực tế là các ứng dụng quản lý danh bạ điện thoại thì có sắp xếp theo số, theo tên. Quản lý học sinh thì có sắp xếp theo điểm, theo lớp, theo trường, ...

- Nén dữ liệu: mã hóa Huffman. Hiệu quả phụ thuộc vào việc sắp xếp thứ tự xuất hiện các mục
- Giải thuật xử lý xâu: tìm tiền tố dài nhất giữa một tập các xâu, tìm chuỗi chung giữa các xâu
→ cần giải thuật sắp xếp xâu
- Các thuật toán lập lịch (Scheduling algorithms),
 - ví dụ thiết kế chương trình dịch, truyền thông, ...
- Đồ họa máy tính
- Sinh tin học
- Cân bằng tải trong máy tính song song
- Hiển thị kết quả xếp hạng của google
- Tổ chức thư viện MP3
- Máy tìm kiếm web (Web search engine);
và nhiều ứng dụng khác...

Ứng dụng của sắp xếp

How Obama got the programmer vote in the 2008 United States Presidential Campaign



Barack Obama got asked a computer science question by Google CEO Eric Schmidt

Các loại thuật toán sắp xếp

Sắp xếp trong (internal sort):

- Đòi hỏi họ dữ liệu được đưa toàn bộ vào bộ nhớ trong của máy tính
- Ví dụ:
 - insertion sort (sắp xếp chèn), selection sort (sắp xếp lựa chọn), bubble sort (sắp xếp nổi bọt)
 - quick sort (sắp xếp nhanh), merge sort (sắp xếp trộn), heap sort (sắp xếp vun đống), sample sort (sắp xếp dựa mẫu), shell sort (vỏ sò)

Sắp xếp ngoài (external sort):

- Họ dữ liệu không thể cùng lúc đưa toàn bộ vào bộ nhớ trong, nhưng có thể đọc vào từng bộ phận từ bộ nhớ ngoài
- Ví dụ: Poly-phase mergesort (trộn nhiều đoạn), cascade-merge (thác nước), oscillating sort (dao động)

Sắp xếp song song (Parallel sort):

- Bitonic sort, Batcher even-odd sort.
- Smooth sort, cube sort, column sort.
- GPU sort.

Bài toán sắp xếp

- **Các đặc trưng của một thuật toán sắp xếp:**

- **Tại chỗ (in place):** nếu **không gian nhớ phụ** mà thuật toán đòi hỏi là $O(1)$, nghĩa là bị chặn bởi hằng số không phụ thuộc vào độ dài của dãy cần sắp xếp.
- **Ổn định (stable):** Nếu các phần tử có cùng giá trị vẫn giữ nguyên thứ tự tương đối của chúng như trước khi sắp xếp.

“sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted.”

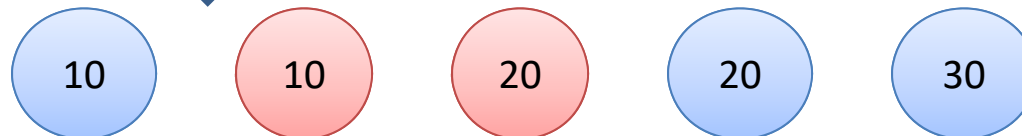
Trước sắp xếp



Sắp xếp này ổn định vì thứ tự của các quả bóng có giá trị bằng nhau là không thay đổi trước và sau khi sắp xếp:

- Quả bóng màu xanh với giá trị 10 đứng trước quả bóng màu cam giá trị 10.
- Tương tự với 2 quả bóng xanh và cam cùng giá trị 20

Sau sắp xếp



Bài toán sắp xếp

- Có hai phép toán cơ bản mà thuật toán sắp xếp thường phải sử dụng:

- **Đổi chỗ** (Swap): Thời gian thực hiện là $O(1)$

```
void swap( datatype &a, datatype &b){  
    datatype temp = a; //datatype-kiểu dữ liệu của phần tử  
    a = b;  
    b = temp;  
}
```

- **So sánh**: Compare(a, b) trả lại true nếu a đi trước b trong thứ tự cần sắp xếp và false nếu trái lại.

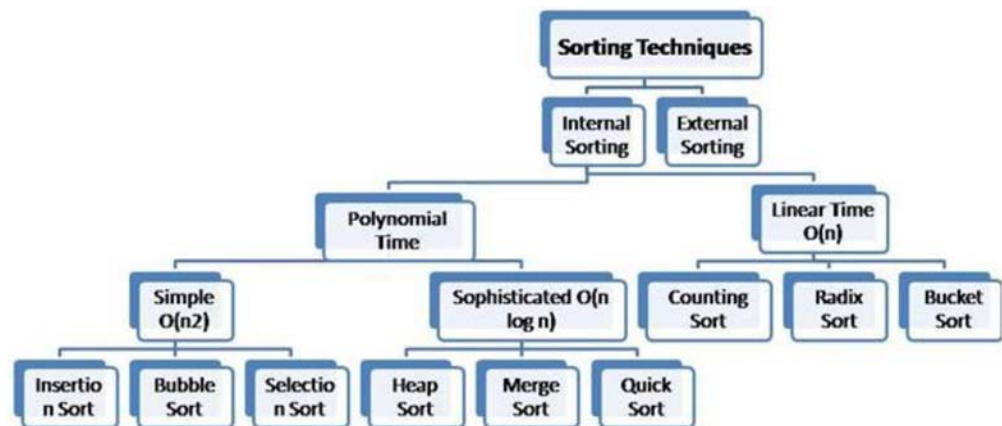
- Các thuật toán chỉ sử dụng phép toán so sánh để xác định thứ tự giữa hai phần tử được gọi là thuật toán sử dụng phép so sánh (*Comparison-based sorting algorithm*).

- Nếu có những thông tin bổ sung về dữ liệu đầu vào, ví dụ như:

- Các số nguyên nằm trong khoảng $[0..k]$ trong đó $k = O(n)$
- Các số thực phân bố đều trong khoảng $[0, 1)$

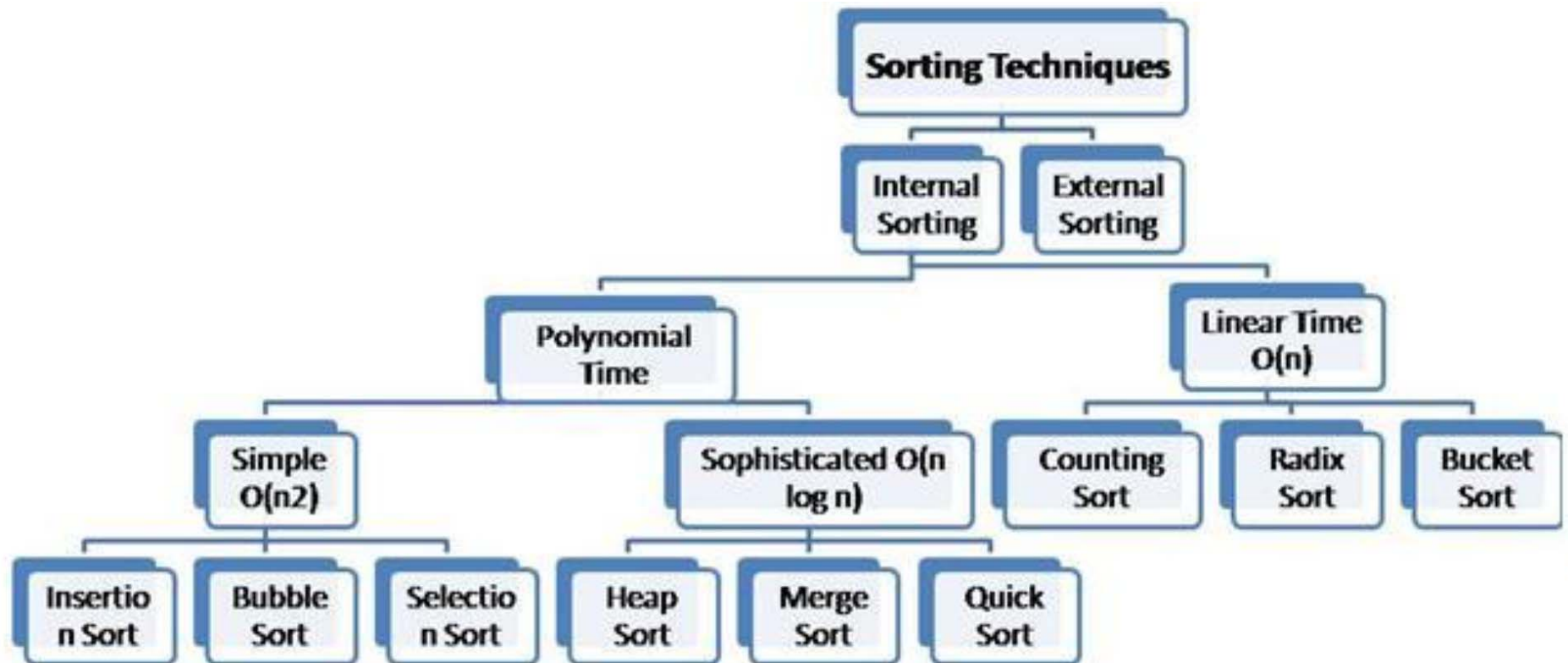
ta sẽ có thuật toán tốt hơn thuật toán sắp xếp chỉ dựa vào phép so sánh.

(Thuật toán thời gian tuyến tính: sắp xếp đếm (counting-sort), sắp xếp theo cơ số (radix-sort), sắp xếp đóng gói (bucket-sort))



1

Bài toán sắp xếp



1

Các thuật toán dựa trên phép so sánh

| Name | Average | Worst | Memory | Stable | Method |
|---------------------|---------------|-----------------|-------------|--------|--------------|
| Bubble sort | — | $O(n^2)$ | $O(1)$ | Yes | Exchanging |
| Cocktail sort | — | $O(n^2)$ | $O(1)$ | Yes | Exchanging |
| Comb sort | $O(n \log n)$ | $O(n \log n)$ | $O(1)$ | No | Exchanging |
| Gnome sort | — | $O(n^2)$ | $O(1)$ | Yes | Exchanging |
| Selection sort | $O(n^2)$ | $O(n^2)$ | $O(1)$ | No | Selection |
| Insertion sort | $O(n + d)$ | $O(n^2)$ | $O(1)$ | Yes | Insertion |
| Shell sort | — | $O(n \log^2 n)$ | $O(1)$ | No | Insertion |
| Binary tree sort | $O(n \log n)$ | $O(n \log n)$ | $O(n)$ | Yes | Insertion |
| Library sort | $O(n \log n)$ | $O(n^2)$ | $O(n)$ | Yes | Insertion |
| Merge sort | $O(n \log n)$ | $O(n \log n)$ | $O(n)$ | Yes | Merging |
| In-place merge sort | $O(n \log n)$ | $O(n \log n)$ | $O(1)$ | Yes | Merging |
| Heapsort | $O(n \log n)$ | $O(n \log n)$ | $O(1)$ | No | Selection |
| Smoothsort | — | $O(n \log n)$ | $O(1)$ | No | Selection |
| Quicksort | $O(n \log n)$ | $O(n^2)$ | $O(\log n)$ | No | Partitioning |
| Introsort | $O(n \log n)$ | $O(n \log n)$ | $O(\log n)$ | No | Hybrid |
| Patience sorting | — | $O(n^2)$ | $O(n)$ | No | Insertion |

Các thuật toán không dựa trên phép so sánh

| Name | Average | Worst | Memory | Stable | $n \ll 2^k$? |
|-----------------|------------------------|----------------------------------|----------------------|--------|---------------|
| Pigeonhole sort | $O(n+2^k)$ | $O(n+2^k)$ | $O(2^k)$ | Yes | Yes |
| Bucket sort | $O(n \cdot k)$ | $O(n^2 \cdot k)$ | $O(n \cdot k)$ | Yes | No |
| Counting sort | $O(n+2^k)$ | $O(n+2^k)$ | $O(n+2^k)$ | Yes | Yes |
| LSD Radix sort | $O(n \cdot k/s)$ | $O(n \cdot k/s)$ | $O(n)$ | Yes | No |
| MSD Radix sort | $O(n \cdot k/s)$ | $O(n \cdot (k/s) \cdot 2^s)$ | $O((k/s) \cdot 2^s)$ | No | No |
| Spreadsort | $O(n \cdot k/\log(n))$ | $O(n \cdot (k - \log(n))^{0.5})$ | $O(n)$ | No | No |

Các thông số trong bảng

n - số phần tử cần sắp xếp

k - kích thước mỗi phần tử

s - kích thước bộ phận được sử dụng khi cài đặt.

Nhiều thuật toán được xây dựng dựa trên giả thiết là $n \ll 2^k$.

Các thuật toán sắp xếp

Khi so sánh các thuật toán, thông thường quan tâm đến:

- **Thời gian** chạy. Đối với các dữ liệu rất lớn, các thuật toán không hiệu quả sẽ chạy rất chậm và không thể ứng dụng trong thực tế.
- **Bộ nhớ**. Các thuật toán nhanh đòi hỏi độ quy sẽ có thể phải tạo ra các bản copy của dữ liệu đang xử lý. Với những hệ thống mà bộ nhớ có giới hạn (ví dụ embedded system), một vài thuật toán sẽ không thể chạy được.
- **Độ ổn định (stability)**. Độ ổn định được định nghĩa dựa trên điều gì sẽ xảy ra với các phần tử có giá trị giống nhau.
 - Đối với thuật toán sắp xếp ổn định, các phần tử với giá trị bằng nhau sẽ giữ nguyên thứ tự trong mảng trước và sau khi sắp xếp.
 - Đối với thuật toán sắp xếp không ổn định, các phần tử có giá trị bằng nhau sẽ có thể có thứ tự bất kỳ.

Tiêu chí lựa chọn giải thuật

Nhiều yếu tố ảnh hưởng:

- Ổn định
- Danh sách liên kết hay mảng
- Đặc trưng của dữ liệu cần sắp xếp:
 - Nhiều khóa ?
 - Các khóa là phân biệt ?
 - Nhiều dạng khóa ?
 - Kích thước bản ghi lớn hay nhỏ ?
 - Dữ liệu được sắp xếp ngẫu nhiên?

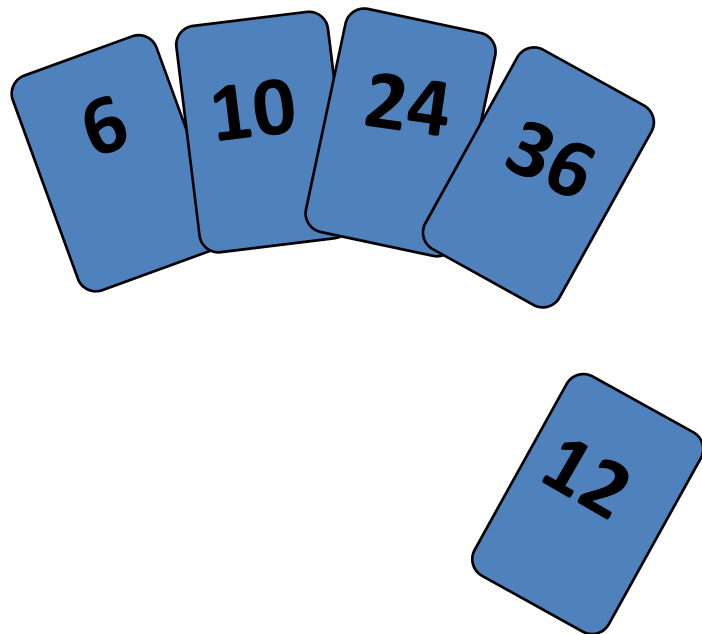
| | | attributes | | | | | | | | |
|-----------|---|------------|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | . | . | . | . | M |
| algorithm | A | • | | | • | | | | | |
| | B | | | • | | • | | | | • |
| | C | | • | | • | | | | | |
| | D | | | | | | • | | | |
| | E | | | • | | | | | | |
| | F | | • | | | • | | | • | |
| | G | • | | | | | | | | • |
| | . | | | • | | • | | | • | |
| | . | | • | • | | | | | • | |
| | . | | | | | | • | | | • |
| | K | • | | | | • | | | | |

Không thể bao phủ tất cả các yếu tố

Các thuật toán sắp xếp

- 1. Sắp xếp chèn (Insertion sort)**
2. Sắp xếp chọn (Selection sort)
3. Sắp xếp trộn (Merge sort)
4. Sắp xếp nhanh (Quick sort)

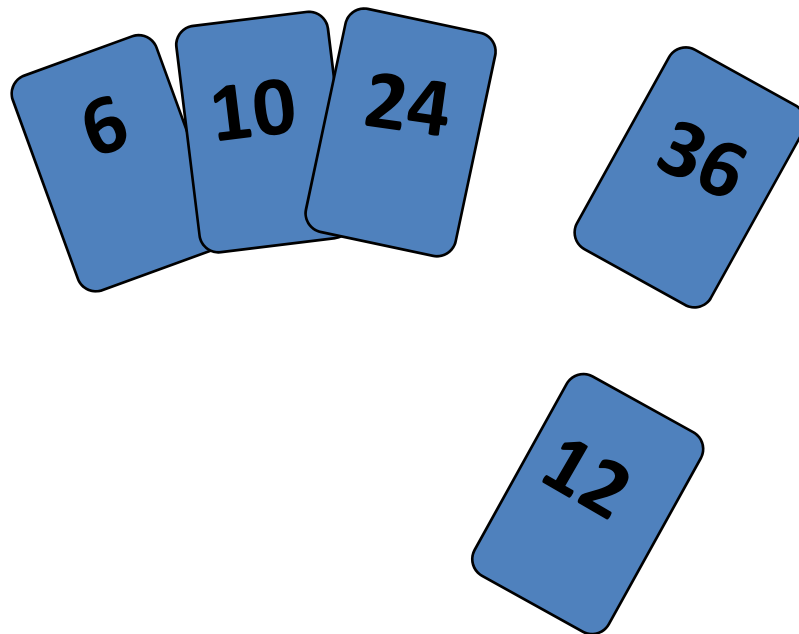
1. Sắp xếp chèn (Insertion sort)



Phỏng theo cách làm của người chơi bài khi cần "chèn" thêm một con bài vào bộ bài đã được sắp xếp trên tay.

Để chèn 12, ta cần tạo chỗ cho nó bởi việc dịch chuyển đầu tiên là 36 và sau đó là 24.

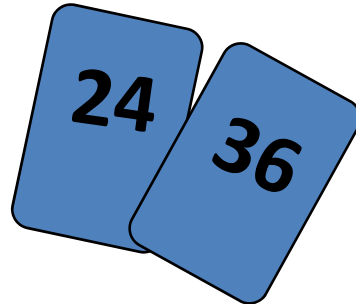
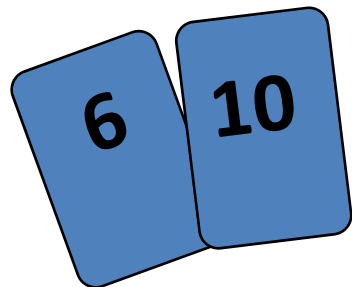
1. Sắp xếp chèn (Insertion sort)



Phỏng theo cách làm của người chơi bài khi cần "chèn" thêm một con bài vào bộ bài đã được sắp xếp trên tay.

Để chèn 12, ta cần tạo chỗ cho nó bởi việc dịch chuyển đầu tiên là 36 và sau đó là 24.

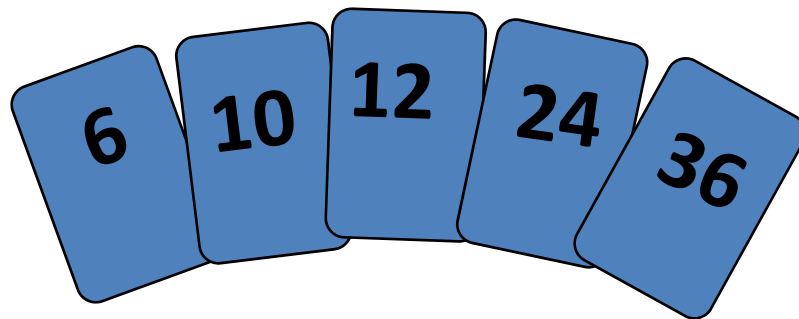
1. Sắp xếp chèn (Insertion sort)



Phỏng theo cách làm của người chơi bài khi cần "chèn" thêm một con bài vào bộ bài đã được sắp xếp trên tay.

Để chèn 12, ta cần tạo chỗ cho nó bởi việc dịch chuyển đầu tiên là 36 và sau đó là 24.

1. Sắp xếp chèn (Insertion sort)



Phỏng theo cách làm của người chơi bài khi cần "chèn" thêm một con bài vào bộ bài đã được sắp xếp trên tay.

Để chèn 12, ta cần tạo chỗ cho nó bởi việc dịch chuyển đầu tiên là 36 và sau đó là 24.

1. Sắp xếp chèn (Insertion sort)

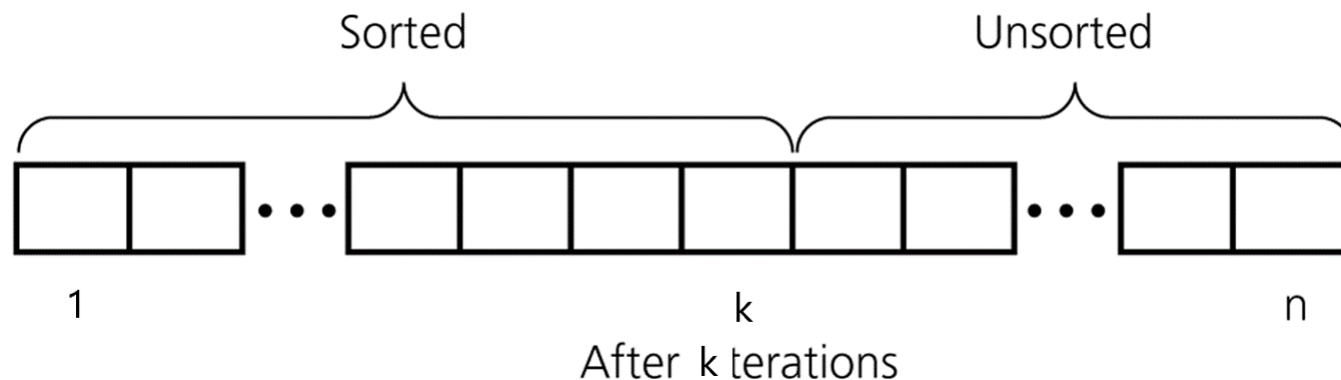
- Thuật toán:

- Tại bước $k = 1, 2, \dots, n$:

- đưa phần tử thứ k trong mảng đã cho vào đúng vị trí trong dãy gồm k phần tử đầu tiên.

Tại mỗi bước lặp k , có thể cần nhiều hơn một lần hoán đổi vị trí các phần tử để có thể đưa phần tử thứ k về đúng vị trí của nó trong dãy cần sắp xếp

Bước lặp k : liên tục đổi chỗ phần tử thứ k với phần tử kề bên trái nó (phần tử ngay trước) chừng nào phần tử thứ k còn nhỏ hơn phần tử đó



Tính chất: Sau bước lặp k , k phần tử đầu tiên $a[1], a[2], \dots, a[k]$ đã được sắp thứ tự.

Ví dụ 1: Insertion Sort

- **Bước lặp k :** liên tục đổi chỗ phần tử thứ k với phần tử kề bên trái nó (phần tử ngay trước) chừng nào phần tử thứ k còn nhỏ hơn phần tử đó
- **Tính chất.** Sau bước lặp thứ k , k phần tử đầu tiên từ $a[1]$ đến $a[k]$ đã được sắp xếp đúng thứ tự.

| Array index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|-------------|------|------|------|------|------|------|------|------|------|------|
| Value | 2.78 | 7.42 | 0.56 | 1.12 | 1.17 | 0.32 | 6.21 | 4.42 | 3.14 | 7.71 |

Iteration 1: step 1.

Ví dụ 1: Insertion Sort

- **Bước lặp k :** liên tục đổi chỗ phần tử thứ k với phần tử kề bên trái nó (phần tử ngay trước) chừng nào phần tử thứ k còn nhỏ hơn phần tử đó
- **Tính chất.** Sau bước lặp thứ k , k phần tử đầu tiên từ $a[1]$ đến $a[k]$ đã được sắp xếp đúng thứ tự.

| Array index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|-------------|------|------|------|------|------|------|------|------|------|------|
| Value | 2.78 | 7.42 | 0.56 | 1.12 | 1.17 | 0.32 | 6.21 | 4.42 | 3.14 | 7.71 |

Iteration 2: step 1.

Ví dụ 1: Insertion Sort

- **Bước lặp k :** liên tục đổi chỗ phần tử thứ k với phần tử kề bên trái nó (phần tử ngay trước) chừng nào phần tử thứ k còn nhỏ hơn phần tử đó
- **Tính chất.** Sau bước lặp thứ k , k phần tử đầu tiên từ $a[1]$ đến $a[k]$ đã được sắp xếp đúng thứ tự.

| Array index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|-------------|------|------|------|------|------|------|------|------|------|------|
| Value | 2.78 | 0.56 | 7.42 | 1.12 | 1.17 | 0.32 | 6.21 | 4.42 | 3.14 | 7.71 |

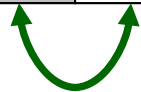


Iteration 3: step 1.

Ví dụ 1: Insertion Sort

- **Bước lặp k :** liên tục đổi chỗ phần tử thứ k với phần tử kề bên trái nó (phần tử ngay trước) chừng nào phần tử thứ k còn nhỏ hơn phần tử đó
- **Tính chất.** Sau bước lặp thứ k , k phần tử đầu tiên từ $a[1]$ đến $a[k]$ đã được sắp xếp đúng thứ tự.

| Array index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|-------------|------|------|------|------|------|------|------|------|------|------|
| Value | 0.56 | 2.78 | 7.42 | 1.12 | 1.17 | 0.32 | 6.21 | 4.42 | 3.14 | 7.71 |



Iteration 3: step 2.

Ví dụ 1: Insertion Sort

- **Bước lặp k :** liên tục đổi chỗ phần tử thứ k với phần tử kề bên trái nó (phần tử ngay trước) chừng nào phần tử thứ k còn nhỏ hơn phần tử đó
- **Tính chất.** Sau bước lặp thứ k , k phần tử đầu tiên từ $a[1]$ đến $a[k]$ đã được sắp xếp đúng thứ tự.

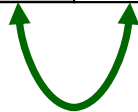
| Array index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|-------------|------|------|------|------|------|------|------|------|------|------|
| Value | 0.56 | 2.78 | 7.42 | 1.12 | 1.17 | 0.32 | 6.21 | 4.42 | 3.14 | 7.71 |

Iteration 3: step 3.

Ví dụ 1: Insertion Sort

- **Bước lặp k :** liên tục đổi chỗ phần tử thứ k với phần tử kề bên trái nó (phần tử ngay trước) chừng nào phần tử thứ k còn nhỏ hơn phần tử đó
- **Tính chất.** Sau bước lặp thứ k , k phần tử đầu tiên từ $a[1]$ đến $a[k]$ đã được sắp xếp đúng thứ tự.

| Array index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|-------------|------|------|------|------|------|------|------|------|------|------|
| Value | 0.56 | 2.78 | 1.12 | 7.42 | 1.17 | 0.32 | 6.21 | 4.42 | 3.14 | 7.71 |



Iteration 4: step 1.

Insertion Sort

- **Bước lặp k :** liên tục đổi chỗ phần tử thứ k với phần tử kề bên trái nó (phần tử ngay trước) chừng nào phần tử thứ k còn nhỏ hơn phần tử đó
- **Tính chất.** Sau bước lặp thứ k , k phần tử đầu tiên từ $a[1]$ đến $a[k]$ đã được sắp xếp đúng thứ tự.

| Array index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|-------------|------|------|------|------|------|------|------|------|------|------|
| Value | 0.56 | 1.12 | 2.78 | 7.42 | 1.17 | 0.32 | 6.21 | 4.42 | 3.14 | 7.71 |



Iteration 4: step 2.

Ví dụ 1: Insertion Sort

- **Bước lặp k :** liên tục đổi chỗ phần tử thứ k với phần tử kề bên trái nó (phần tử ngay trước) chừng nào phần tử thứ k còn nhỏ hơn phần tử đó
- **Tính chất.** Sau bước lặp thứ k , k phần tử đầu tiên từ $a[1]$ đến $a[k]$ đã được sắp xếp đúng thứ tự.

| Array index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|-------------|------|------|------|------|------|------|------|------|------|------|
| Value | 0.56 | 1.12 | 2.78 | 7.42 | 1.17 | 0.32 | 6.21 | 4.42 | 3.14 | 7.71 |

Iteration 4: step 3.

Ví dụ 1: Insertion Sort

- **Bước lặp k :** liên tục đổi chỗ phần tử thứ k với phần tử kề bên trái nó (phần tử ngay trước) chừng nào phần tử thứ k còn nhỏ hơn phần tử đó
- **Tính chất.** Sau bước lặp thứ k , k phần tử đầu tiên từ $a[1]$ đến $a[k]$ đã được sắp xếp đúng thứ tự.

| Array index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|-------------|------|------|------|------|------|------|------|------|------|------|
| Value | 0.56 | 1.12 | 2.78 | 1.17 | 7.42 | 0.32 | 6.21 | 4.42 | 3.14 | 7.71 |

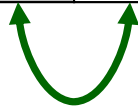


Iteration 5: step 1.

Ví dụ 1: Insertion Sort

- **Bước lặp k :** liên tục đổi chỗ phần tử thứ k với phần tử kề bên trái nó (phần tử ngay trước) chừng nào phần tử thứ k còn nhỏ hơn phần tử đó
- **Tính chất.** Sau bước lặp thứ k , k phần tử đầu tiên từ $a[1]$ đến $a[k]$ đã được sắp xếp đúng thứ tự.

| Array index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|-------------|------|------|------|------|------|------|------|------|------|------|
| Value | 0.56 | 1.12 | 1.17 | 2.78 | 7.42 | 0.32 | 6.21 | 4.42 | 3.14 | 7.71 |



Iteration 5: step 2.

Ví dụ 1: Insertion Sort

- **Bước lặp k :** liên tục đổi chỗ phần tử thứ k với phần tử kề bên trái nó (phần tử ngay trước) chừng nào phần tử thứ k còn nhỏ hơn phần tử đó
- **Tính chất.** Sau bước lặp thứ k , k phần tử đầu tiên từ $a[1]$ đến $a[k]$ đã được sắp xếp đúng thứ tự.

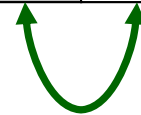
| Array index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|-------------|------|------|------|------|------|------|------|------|------|------|
| Value | 0.56 | 1.12 | 1.17 | 2.78 | 7.42 | 0.32 | 6.21 | 4.42 | 3.14 | 7.71 |

Iteration 5: step 3.

Ví dụ 1: Insertion Sort

- **Bước lặp k :** liên tục đổi chỗ phần tử thứ k với phần tử kề bên trái nó (phần tử ngay trước) chừng nào phần tử thứ k còn nhỏ hơn phần tử đó
- **Tính chất.** Sau bước lặp thứ k , k phần tử đầu tiên từ $a[1]$ đến $a[k]$ đã được sắp xếp đúng thứ tự.

| Array index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|-------------|------|------|------|------|------|------|------|------|------|------|
| Value | 0.56 | 1.12 | 1.17 | 2.78 | 0.32 | 7.42 | 6.21 | 4.42 | 3.14 | 7.71 |

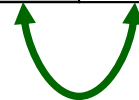


Iteration 6: step 1.

Ví dụ 1: Insertion Sort

- **Bước lặp k :** liên tục đổi chỗ phần tử thứ k với phần tử kề bên trái nó (phần tử ngay trước) chừng nào phần tử thứ k còn nhỏ hơn phần tử đó
- **Tính chất.** Sau bước lặp thứ k , k phần tử đầu tiên từ $a[1]$ đến $a[k]$ đã được sắp xếp đúng thứ tự.

| Array index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|-------------|------|------|------|------|------|------|------|------|------|------|
| Value | 0.56 | 1.12 | 1.17 | 0.32 | 2.78 | 7.42 | 6.21 | 4.42 | 3.14 | 7.71 |



Iteration 6: step 2.

Ví dụ 1: Insertion Sort

- **Bước lặp k :** liên tục đổi chỗ phần tử thứ k với phần tử kề bên trái nó (phần tử ngay trước) chừng nào phần tử thứ k còn nhỏ hơn phần tử đó
- **Tính chất.** Sau bước lặp thứ k , k phần tử đầu tiên từ $a[1]$ đến $a[k]$ đã được sắp xếp đúng thứ tự.

| Array index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|-------------|------|------|------|------|------|------|------|------|------|------|
| Value | 0.56 | 1.12 | 0.32 | 1.17 | 2.78 | 7.42 | 6.21 | 4.42 | 3.14 | 7.71 |



Iteration 6: step 3.

Ví dụ 1: Insertion Sort

- **Bước lặp k :** liên tục đổi chỗ phần tử thứ k với phần tử kề bên trái nó (phần tử ngay trước) chừng nào phần tử thứ k còn nhỏ hơn phần tử đó
- **Tính chất.** Sau bước lặp thứ k , k phần tử đầu tiên từ $a[1]$ đến $a[k]$ đã được sắp xếp đúng thứ tự.

| Array index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|-------------|------|------|------|------|------|------|------|------|------|------|
| Value | 0.56 | 0.32 | 1.12 | 1.17 | 2.78 | 7.42 | 6.21 | 4.42 | 3.14 | 7.71 |

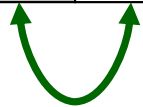


Iteration 6: step 4.

Ví dụ 1: Insertion Sort

- **Bước lặp k :** liên tục đổi chỗ phần tử thứ k với phần tử kề bên trái nó (phần tử ngay trước) chừng nào phần tử thứ k còn nhỏ hơn phần tử đó
- **Tính chất.** Sau bước lặp thứ k , k phần tử đầu tiên từ $a[1]$ đến $a[k]$ đã được sắp xếp đúng thứ tự.

| Array index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|-------------|------|------|------|------|------|------|------|------|------|------|
| Value | 0.32 | 0.56 | 1.12 | 1.17 | 2.78 | 7.42 | 6.21 | 4.42 | 3.14 | 7.71 |



Iteration 6: step 5.

Ví dụ 1: Insertion Sort

- **Bước lặp k :** liên tục đổi chỗ phần tử thứ k với phần tử kề bên trái nó (phần tử ngay trước) chừng nào phần tử thứ k còn nhỏ hơn phần tử đó
- **Tính chất.** Sau bước lặp thứ k , k phần tử đầu tiên từ $a[1]$ đến $a[k]$ đã được sắp xếp đúng thứ tự.

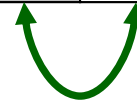
| Array index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|-------------|------|------|------|------|------|------|------|------|------|------|
| Value | 0.32 | 0.56 | 1.12 | 1.17 | 2.78 | 7.42 | 6.21 | 4.42 | 3.14 | 7.71 |

Iteration 6: step 6.

Ví dụ 1: Insertion Sort

- **Bước lặp k :** liên tục đổi chỗ phần tử thứ k với phần tử kề bên trái nó (phần tử ngay trước) chừng nào phần tử thứ k còn nhỏ hơn phần tử đó
- **Tính chất.** Sau bước lặp thứ k , k phần tử đầu tiên từ $a[1]$ đến $a[k]$ đã được sắp xếp đúng thứ tự.

| Array index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|-------------|------|------|------|------|------|------|------|------|------|------|
| Value | 0.32 | 0.56 | 1.12 | 1.17 | 2.78 | 6.21 | 7.42 | 4.42 | 3.14 | 7.71 |



Iteration 7: step 1.

Ví dụ 1: Insertion Sort

- **Bước lặp k :** liên tục đổi chỗ phần tử thứ k với phần tử kề bên trái nó (phần tử ngay trước) chừng nào phần tử thứ k còn nhỏ hơn phần tử đó
- **Tính chất.** Sau bước lặp thứ k , k phần tử đầu tiên từ $a[1]$ đến $a[k]$ đã được sắp xếp đúng thứ tự.

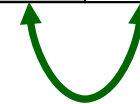
| Array index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|-------------|------|------|------|------|------|------|------|------|------|------|
| Value | 0.32 | 0.56 | 1.12 | 1.17 | 2.78 | 6.21 | 7.42 | 4.42 | 3.14 | 7.71 |

Iteration 7: step 2.

Ví dụ 1: Insertion Sort

- **Bước lặp k :** liên tục đổi chỗ phần tử thứ k với phần tử kề bên trái nó (phần tử ngay trước) chừng nào phần tử thứ k còn nhỏ hơn phần tử đó
- **Tính chất.** Sau bước lặp thứ k , k phần tử đầu tiên từ $a[1]$ đến $a[k]$ đã được sắp xếp đúng thứ tự.

| Array index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|-------------|------|------|------|------|------|------|------|------|------|------|
| Value | 0.32 | 0.56 | 1.12 | 1.17 | 2.78 | 6.21 | 4.42 | 7.42 | 3.14 | 7.71 |

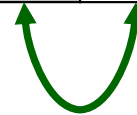


Iteration 8: step 1.

Ví dụ 1: Insertion Sort

- **Bước lặp k :** liên tục đổi chỗ phần tử thứ k với phần tử kề bên trái nó (phần tử ngay trước) chừng nào phần tử thứ k còn nhỏ hơn phần tử đó
- **Tính chất.** Sau bước lặp thứ k , k phần tử đầu tiên từ $a[1]$ đến $a[k]$ đã được sắp xếp đúng thứ tự.

| Array index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|-------------|------|------|------|------|------|------|------|------|------|------|
| Value | 0.32 | 0.56 | 1.12 | 1.17 | 2.78 | 4.42 | 6.21 | 7.42 | 3.14 | 7.71 |



Iteration 8: step 2.

Ví dụ 1: Insertion Sort

- **Bước lặp k :** liên tục đổi chỗ phần tử thứ k với phần tử kề bên trái nó (phần tử ngay trước) chừng nào phần tử thứ k còn nhỏ hơn phần tử đó
- **Tính chất.** Sau bước lặp thứ k , k phần tử đầu tiên từ $a[1]$ đến $a[k]$ đã được sắp xếp đúng thứ tự.

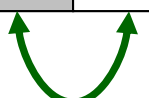
| Array index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|-------------|------|------|------|------|------|------|------|------|------|------|
| Value | 0.32 | 0.56 | 1.12 | 1.17 | 2.78 | 4.42 | 6.21 | 7.42 | 3.14 | 7.71 |

Iteration 8: step 3.

Ví dụ 1: Insertion Sort

- **Bước lặp k :** liên tục đổi chỗ phần tử thứ k với phần tử kề bên trái nó (phần tử ngay trước) chừng nào phần tử thứ k còn nhỏ hơn phần tử đó
- **Tính chất.** Sau bước lặp thứ k , k phần tử đầu tiên từ $a[1]$ đến $a[k]$ đã được sắp xếp đúng thứ tự.

| Array index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|-------------|------|------|------|------|------|------|------|------|------|------|
| Value | 0.32 | 0.56 | 1.12 | 1.17 | 2.78 | 4.42 | 6.21 | 3.14 | 7.42 | 7.71 |

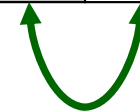


Iteration 9: step 1.

Ví dụ 1: Insertion Sort

- **Bước lặp k :** liên tục đổi chỗ phần tử thứ k với phần tử kề bên trái nó (phần tử ngay trước) chừng nào phần tử thứ k còn nhỏ hơn phần tử đó
- **Tính chất.** Sau bước lặp thứ k , k phần tử đầu tiên từ $a[1]$ đến $a[k]$ đã được sắp xếp đúng thứ tự.

| Array index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|-------------|------|------|------|------|------|------|------|------|------|------|
| Value | 0.32 | 0.56 | 1.12 | 1.17 | 2.78 | 4.42 | 3.14 | 6.21 | 7.42 | 7.71 |

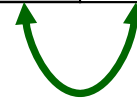


Iteration 9: step 2.

Ví dụ 1: Insertion Sort

- **Bước lặp k :** liên tục đổi chỗ phần tử thứ k với phần tử kề bên trái nó (phần tử ngay trước) chừng nào phần tử thứ k còn nhỏ hơn phần tử đó
- **Tính chất.** Sau bước lặp thứ k , k phần tử đầu tiên từ $a[1]$ đến $a[k]$ đã được sắp xếp đúng thứ tự.

| Array index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|-------------|------|------|------|------|------|------|------|------|------|------|
| Value | 0.32 | 0.56 | 1.12 | 1.17 | 2.78 | 3.14 | 4.42 | 6.21 | 7.42 | 7.71 |



Iteration 9: step 3.

Ví dụ 1: Insertion Sort

- **Bước lặp k :** liên tục đổi chỗ phần tử thứ k với phần tử kề bên trái nó (phần tử ngay trước) chừng nào phần tử thứ k còn nhỏ hơn phần tử đó
- **Tính chất.** Sau bước lặp thứ k , k phần tử đầu tiên từ $a[1]$ đến $a[k]$ đã được sắp xếp đúng thứ tự.

| Array index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|-------------|------|------|------|------|------|------|------|------|------|------|
| Value | 0.32 | 0.56 | 1.12 | 1.17 | 2.78 | 3.14 | 4.42 | 6.21 | 7.42 | 7.71 |

Iteration 9: step 4.

Ví dụ 1: Insertion Sort

- **Bước lặp k :** liên tục đổi chỗ phần tử thứ k với phần tử kề bên trái nó (phần tử ngay trước) chừng nào phần tử thứ k còn nhỏ hơn phần tử đó
- **Tính chất.** Sau bước lặp thứ k , k phần tử đầu tiên từ $a[1]$ đến $a[k]$ đã được sắp xếp đúng thứ tự.

| Array index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|-------------|------|------|------|------|------|------|------|------|------|------|
| Value | 0.32 | 0.56 | 1.12 | 1.17 | 2.78 | 3.14 | 4.42 | 6.21 | 7.42 | 7.71 |

Iteration 10: step 1.

Ví dụ 1: Insertion Sort

- **Bước lặp k :** liên tục đổi chỗ phần tử thứ k với phần tử kề bên trái nó (phần tử ngay trước) chừng nào phần tử thứ k còn nhỏ hơn phần tử đó
- **Tính chất.** Sau bước lặp thứ k , k phần tử đầu tiên từ $a[1]$ đến $a[k]$ đã được sắp xếp đúng thứ tự.

| Array index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|-------------|------|------|------|------|------|------|------|------|------|------|
| Value | 0.32 | 0.56 | 1.12 | 1.17 | 2.78 | 3.14 | 4.42 | 6.21 | 7.42 | 7.71 |

Iteration 10: DONE.

Ví dụ 2: Insertion Sort

- Bước lặp $k = 1, 2, \dots, n$: đưa phần tử thứ k trong mảng đã cho vào đúng vị trí trong dãy gồm k phần tử đầu tiên.

(Bước lặp k : liên tục đổi chỗ phần tử thứ k với phần tử kề bên trái nó (phần tử ngay trước) chừng nào phần tử thứ k còn nhỏ hơn phần tử đó)

- Tính chất.** Sau bước lặp thứ k , k phần tử đầu tiên từ $a[1]$ đến $a[k]$ đã được sắp xếp đúng thứ tự.

| | k=2 | 3 | 4 | 5 | 6 | 7 | 8 |
|----|-----|----|----|----|----|----|----|
| 42 | 20 | 17 | 13 | 13 | 13 | 13 | 13 |
| 20 | 42 | 20 | 17 | 17 | 14 | 14 | 14 |
| 17 | 17 | 42 | 20 | 20 | 17 | 17 | 15 |
| 13 | 13 | 13 | 42 | 28 | 20 | 20 | 17 |
| 28 | 28 | 28 | 28 | 42 | 28 | 23 | 20 |
| 14 | 14 | 14 | 14 | 14 | 42 | 28 | 23 |
| 23 | 23 | 23 | 23 | 23 | 23 | 42 | 28 |
| 15 | 15 | 15 | 15 | 15 | 15 | 15 | 42 |

Ví dụ 3: Insertion Sort

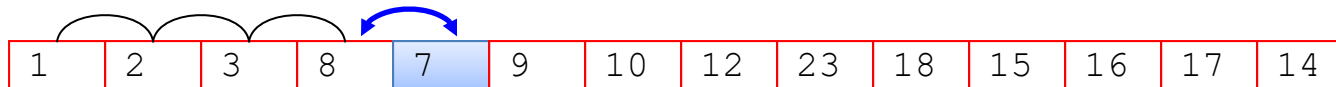
- Bước lặp $k = 1, 2, \dots, n$: đưa phần tử thứ k trong mảng đã cho vào đúng vị trí trong dãy gồm k phần tử đầu tiên.

(Bước lặp k : liên tục đổi chỗ phần tử thứ k với phần tử kề bên trái nó (phần tử ngay trước) chừng nào phần tử thứ k còn nhỏ hơn phần tử đó)

- **Tính chất.** Sau bước lặp thứ k , k phần tử đầu tiên từ $a[1]$ đến $a[k]$ đã được sắp xếp đúng thứ tự.

Ví dụ 3: Cho dãy số gồm 14 phần tử như hình vẽ, hãy áp dụng thuật toán sắp xếp chèn để sắp xếp dãy số đã cho theo thứ tự tăng dần.

Hỏi kết thúc thuật toán, tổng cộng có bao nhiêu phép so sánh và phép đổi chỗ cần thực hiện ?

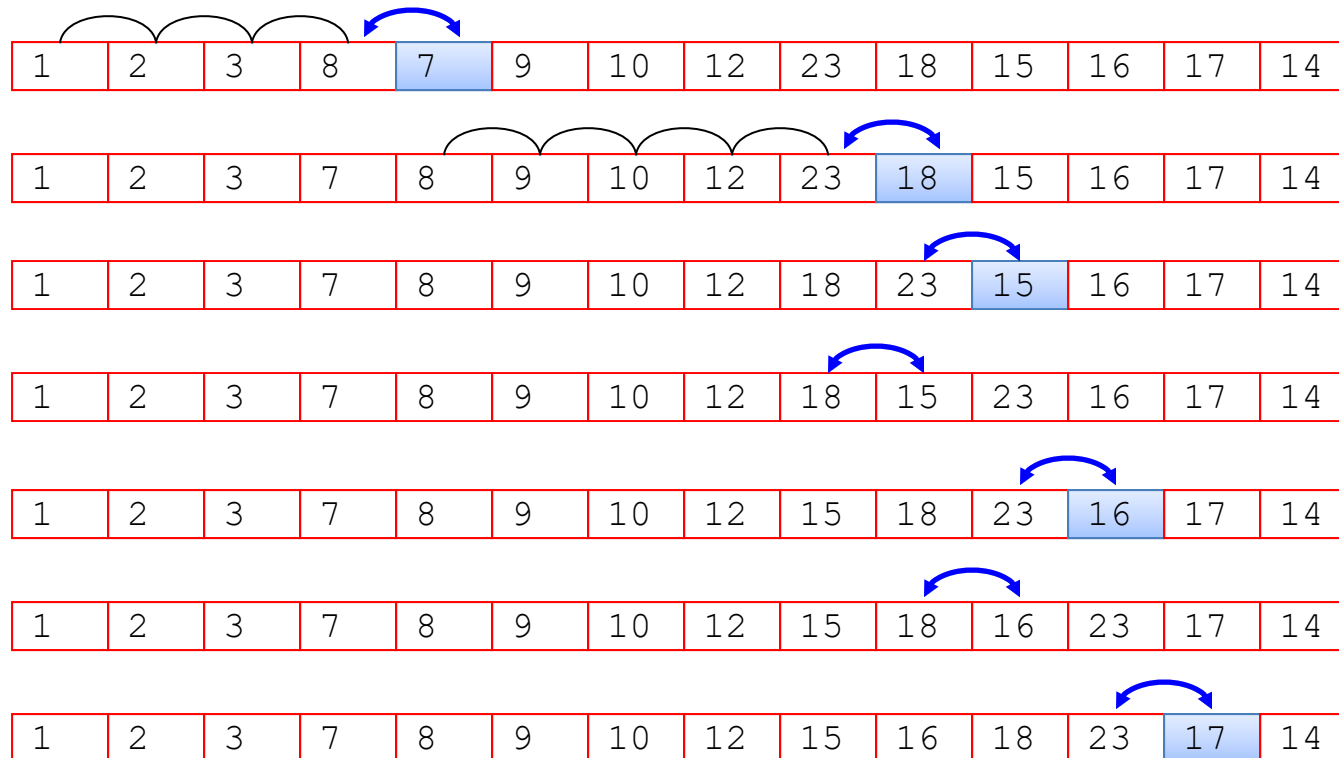


Ví dụ 3: Insertion Sort

- Bước lặp $k = 1, 2, \dots, n$: đưa phần tử thứ k trong mảng đã cho vào đúng vị trí trong dãy gồm k phần tử đầu tiên.

(Bước lặp k : liên tục đổi chỗ phần tử thứ k với phần tử kề bên trái nó (phần tử ngay trước) chừng nào phần tử thứ k còn nhỏ hơn phần tử đó)

- **Tính chất.** Sau bước lặp thứ k , k phần tử đầu tiên từ $a[1]$ đến $a[k]$ đã được sắp xếp đúng thứ tự.

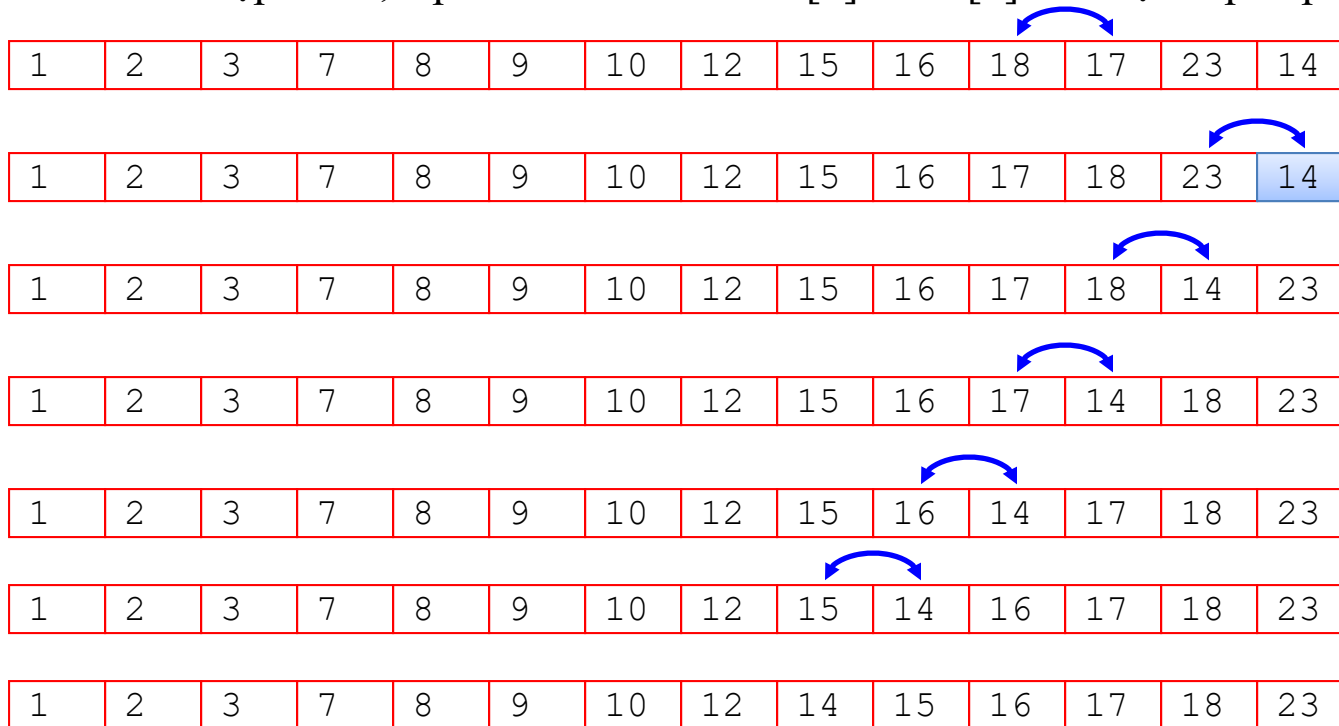




Ví dụ 3: Insertion Sort

- Bước lặp $k = 1, 2, \dots, n$: đưa phần tử thứ k trong mảng đã cho vào đúng vị trí trong dãy gồm k phần tử đầu tiên.

(Bước lặp k : liên tục đổi chỗ phần tử thứ k với phần tử kề bên trái nó (phần tử ngay trước) chừng nào phần tử thứ k còn nhỏ hơn phần tử đó)

- Tính chất.** Sau bước lặp thứ k , k phần tử đầu tiên từ $a[1]$ đến $a[k]$ đã được sắp xếp đúng thứ tự.



54 13 phép đổi chỗ: 
20 phép so sánh: 

Cài đặt: Insertion Sort Algorithm

$k=5$: tìm vị trí đúng cho $a[5]=14$

```
void insertionSort(int a[], int size);
```

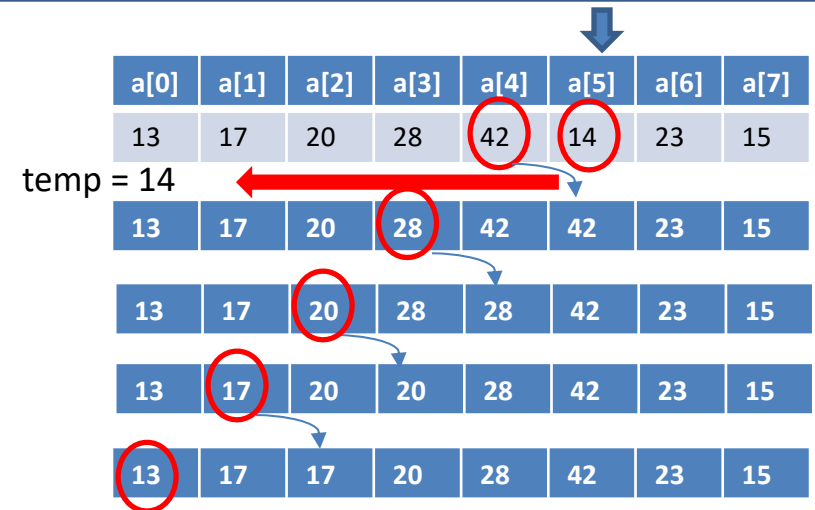
- Thuật toán:

- Tại bước $k = 1, 2, \dots, n$:

- đưa phần tử thứ k trong mảng đã cho vào đúng vị trí trong dãy gồm k phần tử đầu tiên.

Tại mỗi bước lặp k , có thể cần nhiều hơn một lần hoán đổi vị trí các phần tử để có thể đưa phần tử thứ k về đúng vị trí của nó trong dãy cần sắp xếp

Bước lặp k : liên tục đổi chỗ phần tử thứ k với phần tử kề bên trái nó (phần tử ngay trước) chừng nào phần tử thứ k còn nhỏ hơn phần tử đó



```
for (int k = 1; k < size; k++) {  
    int temp = a[k];  
    int pos = k;  
    /* bước lặp k: liên tục đổi chỗ phần tử thứ k với phần tử kề bên  
       trái nó chừng nào phần tử thứ k còn nhỏ hơn phần tử đó */  
    while (pos > 0 && a[pos-1] > temp) {  
        a[pos] = a[pos-1];  
        pos--;  
    } // end while  
}
```

Cài đặt: Insertion Sort Algorithm

$k=5$: tìm vị trí đúng cho $a[5]=14$

```
void insertionSort(int a[], int size);
```

- Thuật toán:

- Tại bước $k = 1, 2, \dots, n$:

- đưa phần tử thứ k trong mảng đã cho vào đúng vị trí trong dãy gồm k phần tử đầu tiên.

Tại mỗi bước lặp k , có thể cần nhiều hơn một lần hoán đổi vị trí các phần tử để có thể đưa phần tử thứ k về đúng vị trí của nó trong dãy cần sắp xếp

Bước lặp k : liên tục đổi chỗ phần tử thứ k với phần tử kề bên trái nó (phần tử ngay trước) chừng nào phần tử thứ k còn nhỏ hơn phần tử đó

temp = 14

| a[0] | a[1] | a[2] | a[3] | a[4] | a[5] | a[6] | a[7] |
|------|------|------|------|------|------|------|------|
| 13 | 17 | 20 | 28 | 42 | 14 | 23 | 15 |
| 13 | 17 | 20 | 28 | 42 | 42 | 23 | 15 |
| 13 | 17 | 20 | 28 | 28 | 42 | 23 | 15 |
| 13 | 17 | 20 | 20 | 28 | 42 | 23 | 15 |
| 13 | 17 | 17 | 20 | 28 | 42 | 23 | 15 |
| 13 | 14 | 17 | 20 | 28 | 42 | 23 | 15 |

```
for (int k = 1; k < size; k++) {  
    int temp = a[k];  
    int pos = k;  
    /* bước lặp k: liên tục đổi chỗ phần tử thứ k với phần tử kề bên  
       trái nó chừng nào phần tử thứ k còn nhỏ hơn phần tử đó */  
    while (pos > 0 && a[pos-1] > temp) {  
        a[pos] = a[pos-1];  
        pos--;  
    } // end while  
    // Chèn giá trị temp (=a[k]) vào đúng vị trí  
    a[pos] = temp;  
}
```


Cài đặt: Insertion Sort Algorithm

```
void insertionSort(int a[], int size) {
    int k, pos, temp;
    for (k=1; k < size; k++) {
        temp = a[k];
        pos = k;
        while ((pos > 0) && (a[pos-1] > temp)) {
            a[pos] = a[pos-1];
            pos = pos - 1;
        }
        a[pos] = temp;
    }
}
```

```
void main()
{
    int a[5] = {8,4,3,2,1};
    insertionSort(a,5);
    for (int i = 0; i<5; i++)
        printf("%d \n",a[i]);
}
```

Đánh giá độ phức tạp tính toán của Insertion sort: $O(n^2)$

- Sắp xếp chèn là tại chỗ và ổn định (In place and Stable)
- Phân tích thời gian tính của thuật toán
 - **Best Case:** 0 hoán đổi, $n-1$ so sánh (khi dãy đầu vào là đã được sắp)
 - **Worst Case:** $n^2/2$ hoán đổi và so sánh (khi dãy đầu vào có thứ tự ngược lại với thứ tự cần sắp xếp)
 - **Average Case:** $n^2/4$ hoán đổi và so sánh
- Là thuật toán sắp xếp tốt đối với dãy đã gần được sắp xếp
 - Nghĩa là mỗi phần tử đã đứng ở vị trí rất gần vị trí trong thứ tự cần sắp xếp

| # of Sorted elements | Best case | Worst case |
|----------------------|-------------------------|------------------------------|
| 0 | 0 | 0 |
| 1 | 1 | 1 |
| 2 | 1 | 2 |
| ... | ... | ... |
| $n-1$ | 1 | $n-1$ |
| Số phép so sánh | $n-1$ | $n(n-1)/2$ |

Đánh giá độ phức tạp tính toán của Insertion sort

Vòng lặp ngoài for:

- thực hiện $n-1$ lần
 - Gồm 5 câu lệnh (bao gồm cả câu lệnh gán và so sánh ở vòng lặp for)
- Tổng chi phí cho vòng lặp ngoài: $5(n-1)$

Số lần vòng lặp while thực hiện phụ thuộc vào trạng thái sắp xếp các phần tử trong mảng:

- Best case: mảng đầu vào đã được sắp xếp → do đó phép hoán đổi vị trí các phần tử trong mảng không được thực hiện lần nào.
 - Với mỗi giá trị k ở vòng lặp for, ta chỉ test điều kiện thực hiện vòng lặp while đúng 1 lần (2 thao tác = 1 phép so sánh $pos > 0$ và 1 phép so sánh phần tử $a[pos-1] > temp$), các câu lệnh trong thân vòng lặp while không bao giờ được thực hiện
 - Do đó, tổng chi phí cho vòng lặp while trong toàn bộ chương trình là $2(n-1)$ thao tác.
- Worst case: mảng đầu vào có thứ tự ngược với thứ tự cần sắp xếp
 - Vòng lặp for thứ k : vòng lặp while thực hiện tổng cộng $4k+1$ thao tác
 - Do đó, tổng chi phí cho vòng lặp while trong toàn bộ chương trình là $2n(n-1)+n-1$

→ Time cost:

- Best case: $7(n-1)$
- Worst case: $5(n-1)+2n(n-1)+n-1$

```
void insertionSort(int a[], int n) {  
    int k, pos, temp;  
    for (k=1; k < n; k++) {  
        temp = a[k];  
        pos = k;  
        while ((pos > 0) && (a[pos-1] > temp))  
        {  
            a[pos] = a[pos-1];  
            pos = pos - 1;  
        }  
        a[pos] = temp;  
    }  
}
```

Các thuật toán sắp xếp

1. Sắp xếp chèn (Insertion sort)
- 2. Sắp xếp chọn (Selection sort)**
3. Sắp xếp trộn (Merge sort)
4. Sắp xếp nhanh (Quick sort)

2. Sắp xếp chọn (Selection sort)

- Thuật toán
 - Tìm phần tử nhỏ nhất đưa vào vị trí 1
 - Tìm phần tử nhỏ tiếp theo đưa vào vị trí 2
 - Tìm phần tử nhỏ tiếp theo đưa vào vị trí 3
 - ...

| | i=0 | 1 | 2 | 3 | 4 | 5 | 6 |
|----|-----|----|----|----|----|----|----|
| 42 | 13 | 13 | 13 | 13 | 13 | 13 | 13 |
| 20 | 20 | 14 | 14 | 14 | 14 | 14 | 14 |
| 17 | 17 | 17 | 15 | 15 | 15 | 15 | 15 |
| 13 | 42 | 42 | 42 | 17 | 17 | 17 | 17 |
| 28 | 28 | 28 | 28 | 28 | 20 | 20 | 20 |
| 14 | 14 | 20 | 20 | 20 | 28 | 23 | 23 |
| 23 | 23 | 23 | 23 | 23 | 23 | 28 | 28 |
| 15 | 15 | 15 | 17 | 42 | 42 | 42 | 42 |

```
void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

```
void selectionSort(int a[], int n){
    int i, j, index_min;
    for (i = 0; i < n-1; i++) {
        index_min = i;
        //Tìm phần tử nhỏ nhất từ a[i+1] đến phần tử cuối cùng trong mảng
        for (j = i+1; j < n; j++)
            if (a[j] < a[index_min]) index_min = j;
        //đưa phần tử a[index_min] vào vị trí thứ i:
        swap(&a[i], &a[index_min]);
    }
}
```

Đánh giá độ phức tạp tính toán của Selection sort: $O(n^2)$

- Thuật toán
 - Tìm phần tử nhỏ nhất đưa vào vị trí 1
 - Tìm phần tử nhỏ tiếp theo đưa vào vị trí 2
 - Tìm phần tử nhỏ tiếp theo đưa vào vị trí 3
 - ...

| | i=0 | 1 | 2 | 3 | 4 | 5 | 6 |
|----|-----|----|----|----|----|----|----|
| 42 | 13 | 13 | 13 | 13 | 13 | 13 | 13 |
| 20 | 20 | 14 | 14 | 14 | 14 | 14 | 14 |
| 17 | 17 | 17 | 15 | 15 | 15 | 15 | 15 |
| 13 | 42 | 42 | 42 | 17 | 17 | 17 | 17 |
| 28 | 28 | 28 | 28 | 28 | 20 | 20 | 20 |
| 14 | 14 | 20 | 20 | 20 | 28 | 23 | 23 |
| 23 | 23 | 23 | 23 | 23 | 23 | 28 | 28 |
| 15 | 15 | 15 | 17 | 42 | 42 | 42 | 42 |

Đánh giá độ phức tạp tính toán:

- Best case:** 0 phép đổi chỗ, $n^2/2$ phép so sánh.
- Worst case:** $n - 1$ phép đổi chỗ, $n^2/2$ phép so sánh.
- Average case:** $O(n)$ phép đổi chỗ, $n^2/2$ phép so sánh.
- Ưu điểm nổi bật của sắp xếp chọn là số phép đổi chỗ là ít. Điều này là có ý nghĩa nếu như việc đổi chỗ là tốn kém.

Các thuật toán sắp xếp

1. Sắp xếp chèn (Insertion sort)
 2. Sắp xếp chọn (Selection sort)
 3. Sắp xếp trộn (Merge sort)
 4. Sắp xếp nhanh (Quick sort)
- } Devide and conquer

Chia để trị (Divide and Conquer)

3 bước thực hiện:

- Divide: Phân rã bài toán đã cho thành **bài toán cùng dạng với kích thước nhỏ hơn (gọi là bài toán con)**
- Conquer: Giải các bài toán con một cách độc lập
- Combine: Tổng hợp lời giải của các bài toán con để thu được lời giải của bài toán ban đầu

Idea 1: Divide array into two halves, *recursively* sort left and right halves, then *merge* two halves → **Mergesort**

Idea 2 : Partition array into items that are “small” and items that are “large”, then *recursively* sort the two sets → **Quicksort**

Các thuật toán sắp xếp

1. Sắp xếp chèn (Insertion sort)
 2. Sắp xếp chọn (Selection sort)
 - 3. Sắp xếp trộn (Merge sort)**
 4. Sắp xếp nhanh (Quick sort)
- } Devide and conquer

3. Sắp xếp trộn (Merge sort)

Bài toán: Cần sắp xếp mảng $A[1 .. n]$:

- **Chia (Divide)**
 - Chia dãy gồm n phần tử cần sắp xếp ra thành 2 dãy, mỗi dãy có $n/2$ phần tử
- **Trị (Conquer)**
 - Sắp xếp mỗi dãy con một cách đệ qui sử dụng *sắp xếp trộn*
 - Khi dãy chỉ còn một phần tử thì trả lại phần tử này
- **Tổ hợp (Combine)**
 - Trộn (Merge) hai dãy con được sắp xếp để thu được dãy được sắp xếp gồm tất cả các phần tử của cả hai dãy con

3. Sắp xếp trộn (Merge Sort)

MERGE-SORT(A, p, r)

if $p < r$

then $q \leftarrow \lfloor (p + r)/2 \rfloor$

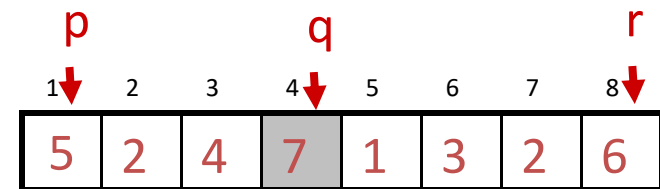
MERGE-SORT(A, p, q)

MERGE-SORT($A, q + 1, r$)

MERGE(A, p, q, r)

endif

- Lệnh gọi thực hiện thuật toán: MERGE-SORT($A, 1, n$)



▷ Kiểm tra điều kiện neo

▷ Chia (Divide)

▷ Trị (Conquer)

▷ Trị (Conquer)

▷ Tổ hợp (Combine)

MERGE-SORT(A, p, r)

if $p < r$

then $q \leftarrow \lfloor (p + r) / 2 \rfloor$

MERGE-SORT(A, p, q)

MERGE-SORT(A, q + 1, r)

MERGE(A, p, q, r)

endif



▷ Kiểm tra điều kiện neo

▷ Chia (Divide)

▷ Trị (Conquer)

▷ Trị (Conquer)

▷ Tổ hợp (Combine)

Ví dụ: Sắp xếp trộn

Divide

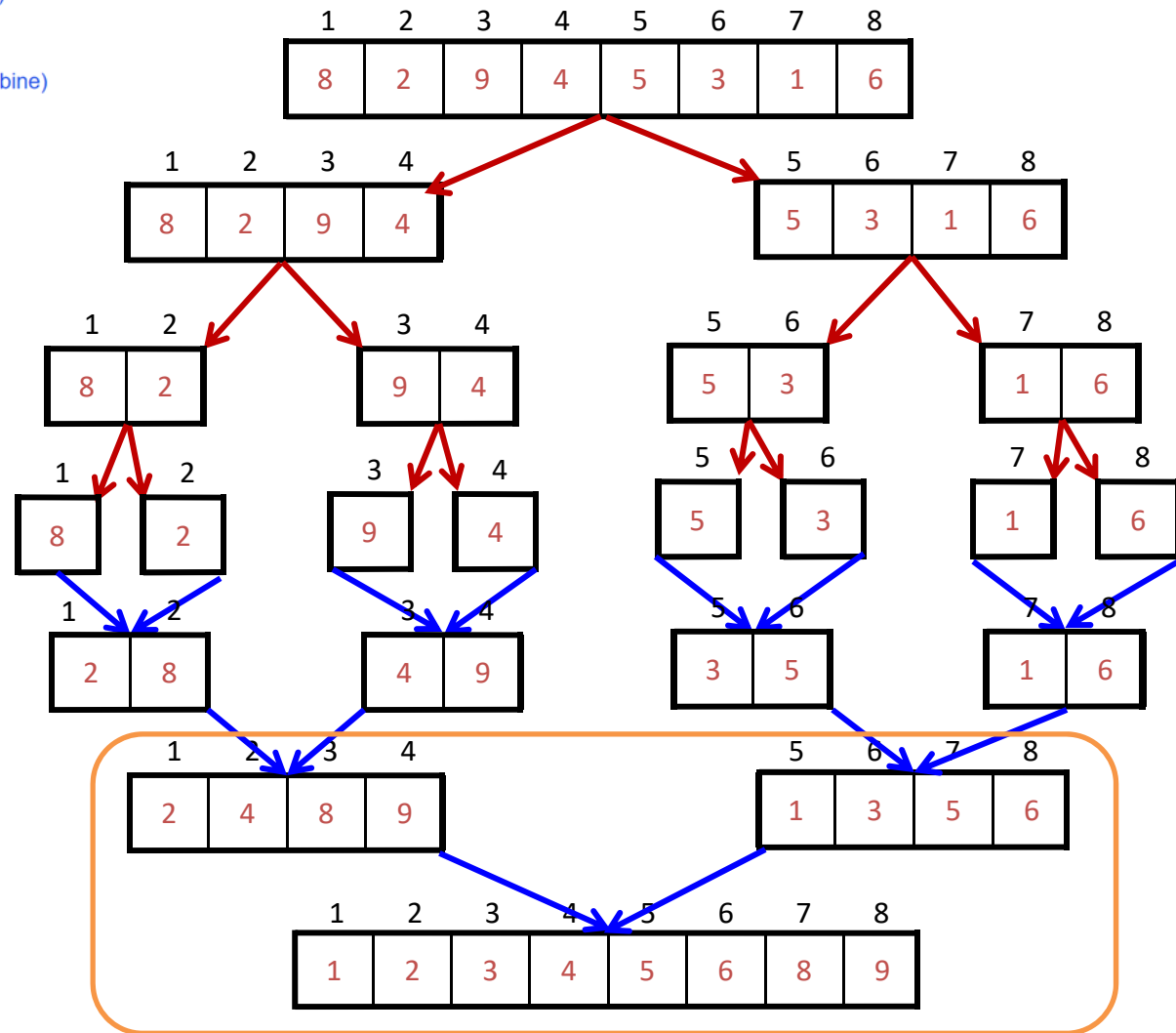
Divide

1 phần tử

Merge

Merge

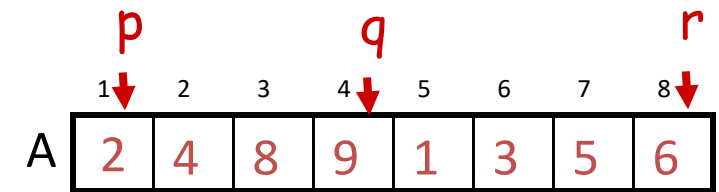
Kết quả:



Ý tưởng trộn 2 dãy đã sắp xếp theo thứ tự tăng dần thành 1 dãy sắp xếp cũng theo thứ tự tăng dần

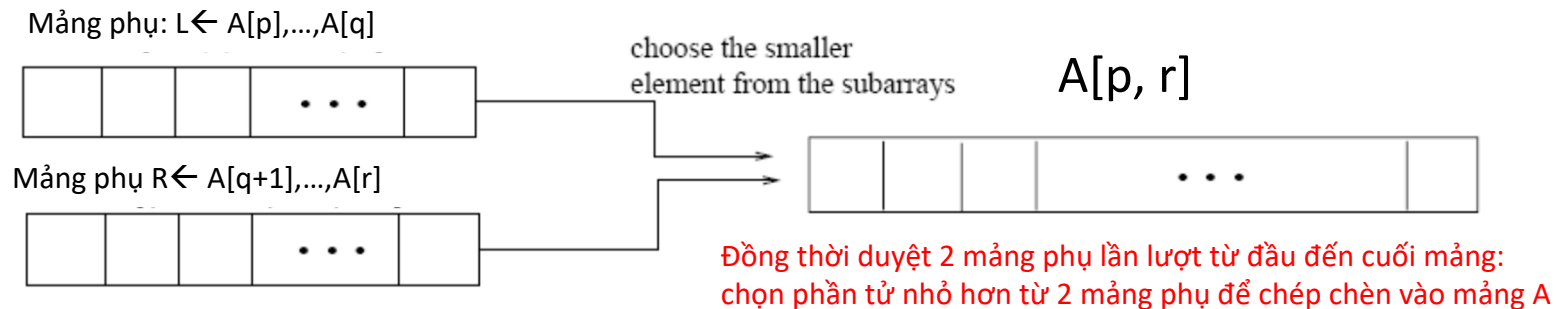
Mảng A gồm 2 nửa đã sắp xếp:

- Dãy 1: gồm các số từ $A[p] \dots A[q]$
- Dãy 2: gồm các số từ $A[q+1] \dots A[r]$



Cần trộn 2 dãy này với nhau để thu được 1 dãy gồm các số được sắp xếp theo thứ tự tăng dần.

Idea 1: dùng 2 mảng phụ L, R có kích thước bằng $\frac{1}{2}$ mảng A



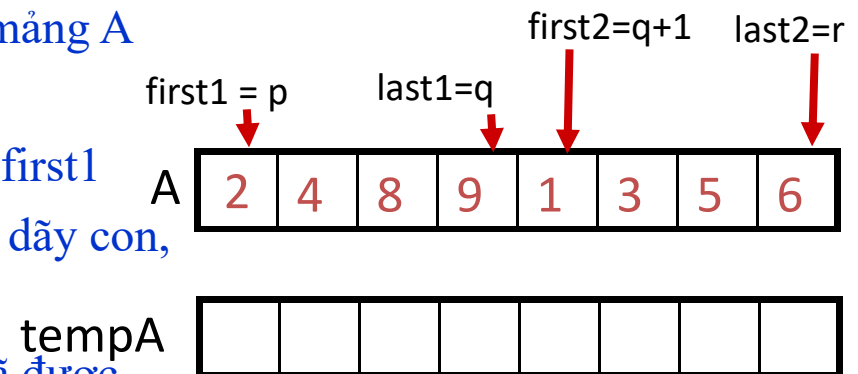
Idea 2: dùng 1 mảng phụ tempA có kích thước bằng mảng A

Đồng thời duyệt qua lần lượt từng phần tử của

2 dãy con $A[p..q]$ và $A[q+1..r]$ bằng cách dùng biến first1 và first2: so sánh từng cặp 2 phần tử tương ứng của 2 dãy con, chọn phần tử nhỏ hơn để chép vào mảng phụ tempA.

Kết thúc vòng lặp, tất cả các phần tử của 2 dãy con đã được

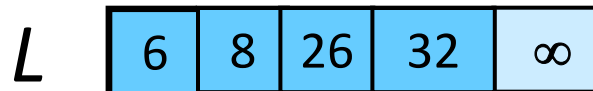
duyet qua; khi đó mảng tempA chứa tất cả các phần tử của 2 dãy con, nhưng các phần tử đã được sắp xếp. Copy mảng tempA chèn vào mảng A



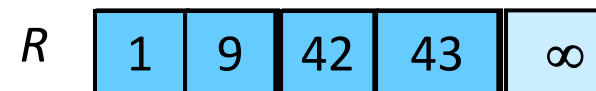
Trộn (Merge) – Minh họa Idea1



k



i

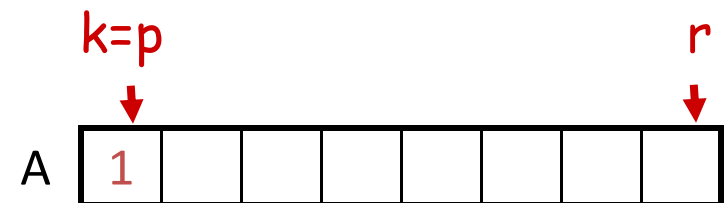
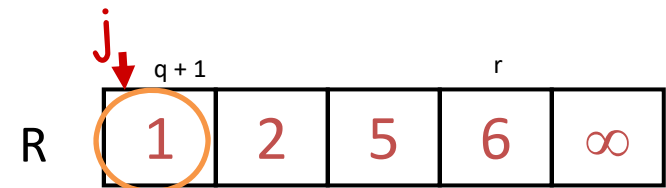
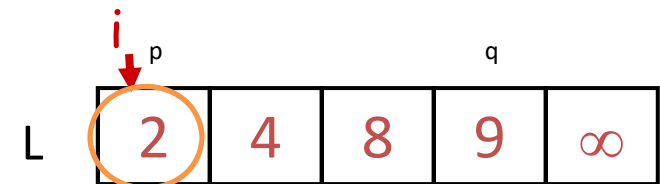
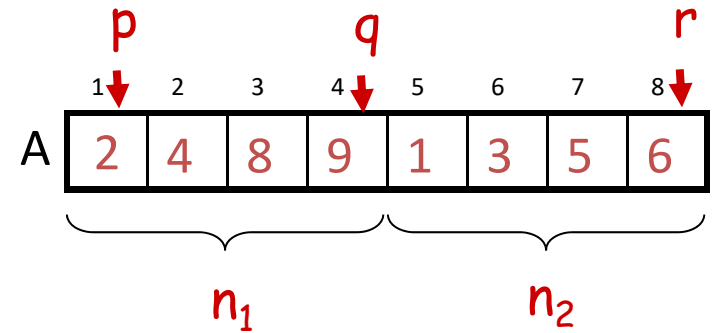


j

Idea 1: Trộn (Merge) - Pseudocode

MERGE(A, p, q, r)

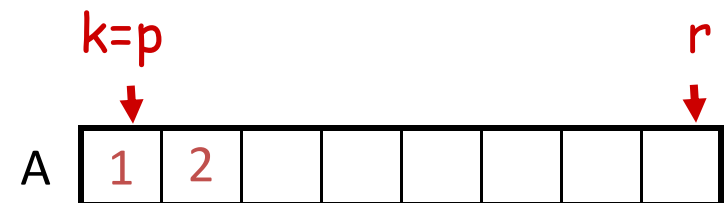
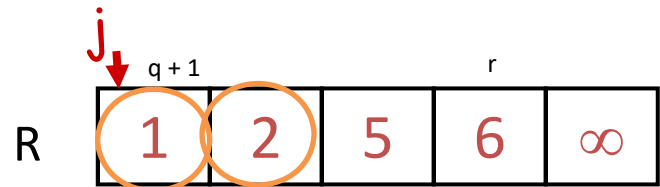
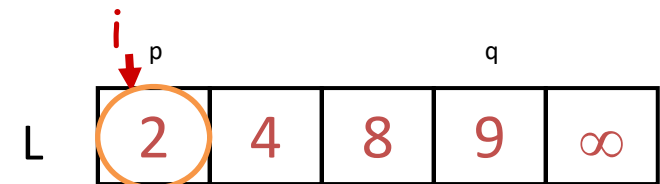
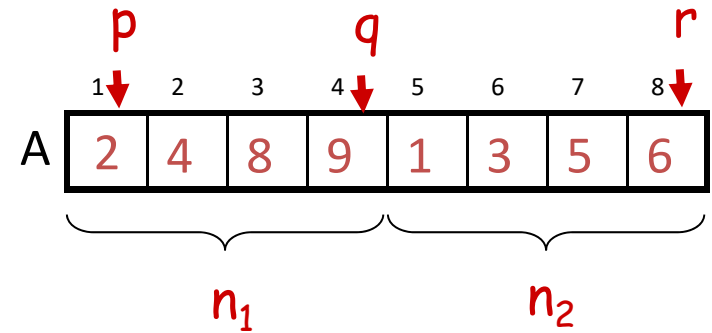
1. Tính n_1 và n_2
2. Sao n_1 phần tử đầu tiên vào $L[1 \dots n_1]$ và n_2 phần tử tiếp theo vào $R[1 \dots n_2]$
3. $L[n_1 + 1] \leftarrow \infty$; $R[n_2 + 1] \leftarrow \infty$
4. $i \leftarrow 1$; $j \leftarrow 1$
5. **for** $k \leftarrow p$ **to** r **do**
6. **if** $L[i] \leq R[j]$
7. **then** $A[k] \leftarrow L[i]$
8. $i \leftarrow i + 1$
9. **else** $A[k] \leftarrow R[j]$
10. $j \leftarrow j + 1$



Idea 1: Trộn (Merge) - Pseudocode

MERGE(A, p, q, r)

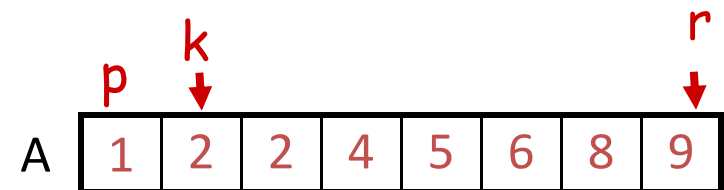
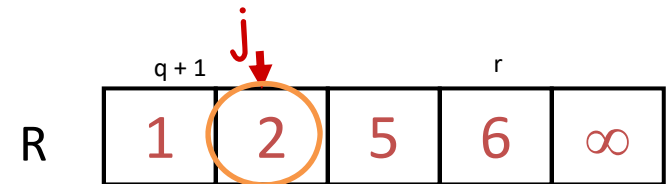
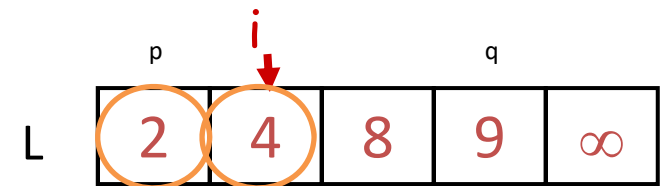
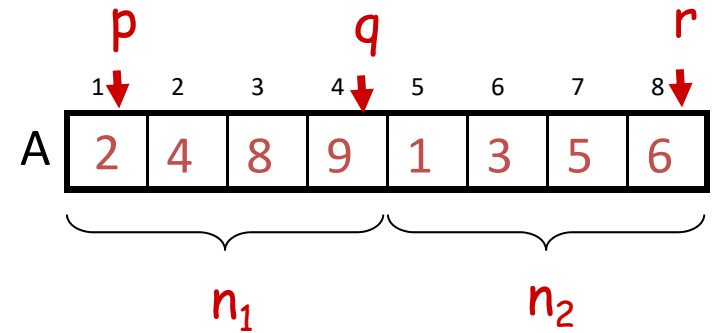
1. Tính n_1 và n_2
2. Sao n_1 phần tử đầu tiên vào $L[1 \dots n_1]$ và n_2 phần tử tiếp theo vào $R[1 \dots n_2]$
3. $L[n_1 + 1] \leftarrow \infty$; $R[n_2 + 1] \leftarrow \infty$
4. $i \leftarrow 1$; $j \leftarrow 1$
5. **for** $k \leftarrow p$ **to** r **do**
6. **if** $L[i] \leq R[j]$
7. **then** $A[k] \leftarrow L[i]$
8. $i \leftarrow i + 1$
9. **else** $A[k] \leftarrow R[j]$
10. $j \leftarrow j + 1$



Idea 1: Trộn (Merge) - Pseudocode

MERGE(A, p, q, r)

1. Tính n_1 và n_2
2. Sao n_1 phần tử đầu tiên vào $L[1 \dots n_1]$ và n_2 phần tử tiếp theo vào $R[1 \dots n_2]$
3. $L[n_1 + 1] \leftarrow \infty$; $R[n_2 + 1] \leftarrow \infty$
4. $i \leftarrow 1$; $j \leftarrow 1$
5. **for** $k \leftarrow p$ **to** r **do**
6. **if** $L[i] \leq R[j]$
7. **then** $A[k] \leftarrow L[i]$
8. $i \leftarrow i + 1$
9. **else** $A[k] \leftarrow R[j]$
10. $j \leftarrow j + 1$



Đánh giá độ phức tạp tính toán của Merge sort: $O(n \log n)$

Thời gian tính của thủ tục Merge:

- Khởi tạo (tạo hai mảng con tạm thời L và R):
 - $\Theta(n_1 + n_2) = \Theta(n)$
- Đưa các phần tử vào mảng kết quả (vòng lặp **for**):
 - n lần lặp, mỗi lần đòi hỏi thời gian hằng số $\Rightarrow \Theta(n)$
- Tổng cộng thời gian của trộn là: $\Theta(n)$

Thời gian tính của sắp xếp trộn Merge Sort:

- **Chia:** tính q như là giá trị trung bình của p và r : $\Theta(1)$
- **Trị:** giải đệ qui 2 bài toán con, mỗi bài toán kích thước $n/2 \Rightarrow 2T(n/2)$
- **Tổ hợp:** TRỘN (MERGE) trên các mảng con cỡ n phần tử đòi hỏi thời gian $\Theta(n)$

$$T(n) = \begin{cases} \Theta(1) & \text{nếu } n = 1 \\ 2T(n/2) + \Theta(n) & \text{nếu } n > 1 \end{cases}$$

Suy ra: $T(n) = \Theta(n \log n)$ (CM bằng qui nạp!)

MERGE(A, p, q, r)

1. Tính n_1 và n_2
2. Sao n_1 phần tử đầu tiên vào $L[1 \dots n_1]$ và n_2 phần tử tiếp theo vào $R[1 \dots n_2]$
3. $L[n_1 + 1] \leftarrow \infty$; $R[n_2 + 1] \leftarrow \infty$
4. $i \leftarrow 1$; $j \leftarrow 1$
5. **for** $k \leftarrow p$ **to** r **do**
6. **if** $L[i] \leq R[j]$
7. **then** $A[k] \leftarrow L[i]$
8. $i \leftarrow i + 1$
9. **else** $A[k] \leftarrow R[j]$
10. $j \leftarrow j + 1$

MERGE-SORT(A, p, r)

if $p < r$

then $q \leftarrow \lfloor (p + r) / 2 \rfloor$

 MERGE-SORT(A, p, q)

 MERGE-SORT(A, q + 1, r)

 MERGE(A, p, q, r)

endif

Idea 2: Trộn (Merge)

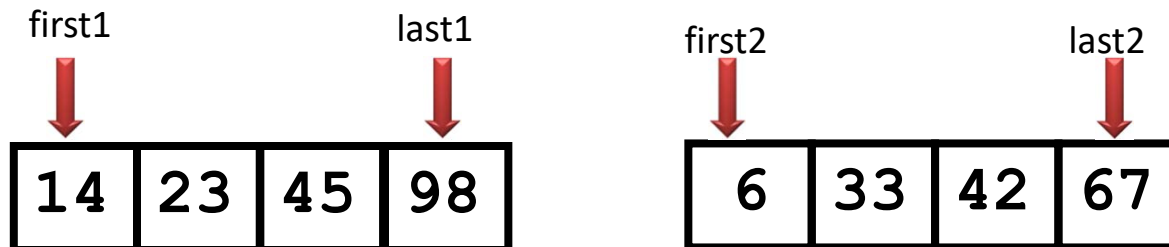
```
#define MAX_SIZE 500
void merge(int A[], int first, int mid, int last)
{
    int tempA[MAX_SIZE];    // mang phu
    int first1 = first; int last1 = mid;
    int first2 = mid + 1; int last2 = last;
    int index = first1;
    for (; (first1 <= last1) && (first2 <= last2); ++index)
    {
        if (A[first1] < A[first2])
        { tempA[index] = A[first1]; ++first1; }
        else
        { tempA[index] = A[first2]; ++first2; }
    }

    for (; first1 <= last1; ++first1, ++index)
        tempA[index] = A[first1]; // sao not dăy con 1
    for (; first2 <= last2; ++first2, ++index)
        tempA[index] = A[first2]; // sao not dăy con 2
    for (index = first; index <= last; ++index)
        A[index] = tempA[index]; // sao tra mang ket qua
} // end merge
```

A

| | | | | | | | |
|----|----|----|----|---|----|----|----|
| 14 | 23 | 45 | 98 | 6 | 33 | 42 | 67 |
|----|----|----|----|---|----|----|----|

A



```
#define MAX_SIZE 500
```

```
void merge(int A[], int first, int mid, int last)
```

```
{  
    int tempA[MAX_SIZE];    // mang phu  
    int first1 = first; int last1 = mid;  
    int first2 = mid + 1; int last2 = last;  
    int index = first1;  
    for (; (first1 <= last1) && (first2 <= last2); ++index)  
    {  
        if (A[first1] < A[first2])  
            {tempA[index] = A[first1]; ++first1;}  
        else  
            { tempA[index] = A[first2]; ++first2;}  
    }  
  
    for (; first1 <= last1; ++first1, ++index)  
        tempA[index] = A[first1]; // sao not dãy con 1  
    for (; first2 <= last2; ++first2, ++index)  
        tempA[index] = A[first2]; // sao not dãy con 2  
    for (index = first; index <= last; ++index)  
        A[index] = tempA[index]; // sao tra mang ket qua  
} // end merge
```

Idea 2: Trộn (Merge)

A

| | | | |
|----|----|----|----|
| 14 | 23 | 45 | 98 |
|----|----|----|----|

| | | | |
|---|----|----|----|
| 6 | 33 | 42 | 67 |
|---|----|----|----|

Merge

```
#define MAX_SIZE 500
```

```
void merge(int A[], int first, int mid, int last)
```

```
{  
    int tempA[MAX_SIZE];    // mang phu  
    int first1 = first; int last1 = mid;  
    int first2 = mid + 1; int last2 = last;  
    int index = first1;  
    for (; (first1 <= last1) && (first2 <= last2); ++index)  
    {  
        if (A[first1] < A[first2])  
            {tempA[index] = A[first1]; ++first1;}  
        else  
            { tempA[index] = A[first2]; ++first2;}  
    }  
  
    for (; first1 <= last1; ++first1, ++index)  
        tempA[index] = A[first1]; // sao not dãy con 1  
    for (; first2 <= last2; ++first2, ++index)  
        tempA[index] = A[first2]; // sao not dãy con 2  
    for (index = first; index <= last; ++index)  
        A[index] = tempA[index]; // sao tra mang ket qua  
} // end merge
```

Idea 2: Trộn (Merge)

A

| | | | |
|----|----|----|----|
| 14 | 23 | 45 | 98 |
|----|----|----|----|

| | | | |
|---|----|----|----|
| 6 | 33 | 42 | 67 |
|---|----|----|----|

tempA

| |
|---|
| 6 |
|---|

Merge

```
#define MAX_SIZE 500
```

```
void merge(int A[], int first, int mid, int last)
```

```
{  
    int tempA[MAX_SIZE];    // mang phu  
    int first1 = first; int last1 = mid;  
    int first2 = mid + 1; int last2 = last;  
    int index = first1;  
    for (; (first1 <= last1) && (first2 <= last2); ++index)  
    {  
        if (A[first1] < A[first2])  
            {tempA[index] = A[first1]; ++first1;}  
        else  
            { tempA[index] = A[first2]; ++first2;}  
    }  
  
    for (; first1 <= last1; ++first1, ++index)  
        tempA[index] = A[first1]; // sao not dãy con 1  
    for (; first2 <= last2; ++first2, ++index)  
        tempA[index] = A[first2]; // sao not dãy con 2  
    for (index = first; index <= last; ++index)  
        A[index] = tempA[index]; // sao tra mang ket qua  
} // end merge
```

Idea 2: Trộn (Merge)

A

| | | | |
|----|----|----|----|
| 14 | 23 | 45 | 98 |
|----|----|----|----|

| | | | |
|---|----|----|----|
| 6 | 33 | 42 | 67 |
|---|----|----|----|

tempA

| | |
|---|----|
| 6 | 14 |
|---|----|

Merge


```
#define MAX_SIZE 500
```

```
void merge(int A[], int first, int mid, int last)
```

```
{  
    int tempA[MAX_SIZE];    // mang phu  
    int first1 = first; int last1 = mid;  
    int first2 = mid + 1; int last2 = last;  
    int index = first1;  
    for (; (first1 <= last1) && (first2 <= last2); ++index)  
    {  
        if (A[first1] < A[first2])  
            {tempA[index] = A[first1]; ++first1;}  
        else  
            { tempA[index] = A[first2]; ++first2;}  
    }  
  
    for (; first1 <= last1; ++first1, ++index)  
        tempA[index] = A[first1]; // sao not dãy con 1  
    for (; first2 <= last2; ++first2, ++index)  
        tempA[index] = A[first2]; // sao not dãy con 2  
    for (index = first; index <= last; ++index)  
        A[index] = tempA[index]; // sao tra mang ket qua  
} // end merge
```

Idea 2: Trộn (Merge)

A

| | | | |
|----|----|----|----|
| 14 | 23 | 45 | 98 |
|----|----|----|----|

| | | | |
|---|----|----|----|
| 6 | 33 | 42 | 67 |
|---|----|----|----|

tempA

| | | |
|---|----|----|
| 6 | 14 | 23 |
|---|----|----|

Merge

```
#define MAX_SIZE 500
```

```
void merge(int A[], int first, int mid, int last)
```

```
{  
    int tempA[MAX_SIZE];    // mang phu  
    int first1 = first; int last1 = mid;  
    int first2 = mid + 1; int last2 = last;  
    int index = first1;  
    for (; (first1 <= last1) && (first2 <= last2); ++index)  
    {  
        if (A[first1] < A[first2])  
            {tempA[index] = A[first1]; ++first1;}  
        else  
            { tempA[index] = A[first2]; ++first2;}  
    }  
  
    for (; first1 <= last1; ++first1, ++index)  
        tempA[index] = A[first1]; // sao not dãy con 1  
    for (; first2 <= last2; ++first2, ++index)  
        tempA[index] = A[first2]; // sao not dãy con 2  
    for (index = first; index <= last; ++index)  
        A[index] = tempA[index]; // sao tra mang ket qua  
} // end merge
```

Idea 2: Trộn (Merge)

A

| | | | |
|----|----|----|----|
| 14 | 23 | 45 | 98 |
|----|----|----|----|

| | | | |
|---|----|----|----|
| 6 | 33 | 42 | 67 |
|---|----|----|----|

tempA

| | | | |
|---|----|----|----|
| 6 | 14 | 23 | 33 |
|---|----|----|----|

Merge


```
#define MAX_SIZE 500
```

```
void merge(int A[], int first, int mid, int last)
```

```
{  
    int tempA[MAX_SIZE];    // mang phu  
    int first1 = first; int last1 = mid;  
    int first2 = mid + 1; int last2 = last;  
    int index = first1;  
    for (; (first1 <= last1) && (first2 <= last2); ++index)  
    {  
        if (A[first1] < A[first2])  
            {tempA[index] = A[first1]; ++first1;}  
        else  
            { tempA[index] = A[first2]; ++first2;}  
    }  
  
    for (; first1 <= last1; ++first1, ++index)  
        tempA[index] = A[first1]; // sao not dãy con 1  
    for (; first2 <= last2; ++first2, ++index)  
        tempA[index] = A[first2]; // sao not dãy con 2  
    for (index = first; index <= last; ++index)  
        A[index] = tempA[index]; // sao tra mang ket qua  
} // end merge
```

Idea 2: Trộn (Merge)

A

| | | | |
|----|----|----|----|
| 14 | 23 | 45 | 98 |
|----|----|----|----|

| | | | |
|---|----|----|----|
| 6 | 33 | 42 | 67 |
|---|----|----|----|

tempA

| | | | | |
|---|----|----|----|----|
| 6 | 14 | 23 | 33 | 42 |
|---|----|----|----|----|

Merge

```
#define MAX_SIZE 500
```

```
void merge(int A[], int first, int mid, int last)
```

```
{  
    int tempA[MAX_SIZE];    // mang phu  
    int first1 = first; int last1 = mid;  
    int first2 = mid + 1; int last2 = last;  
    int index = first1;  
    for (; (first1 <= last1) && (first2 <= last2); ++index)  
    {  
        if (A[first1] < A[first2])  
            {tempA[index] = A[first1]; ++first1;}  
        else  
            { tempA[index] = A[first2]; ++first2;}  
    }  
  
    for (; first1 <= last1; ++first1, ++index)  
        tempA[index] = A[first1]; // sao not dãy con 1  
    for (; first2 <= last2; ++first2, ++index)  
        tempA[index] = A[first2]; // sao not dãy con 2  
    for (index = first; index <= last; ++index)  
        A[index] = tempA[index]; // sao tra mang ket qua  
} // end merge
```

Idea 2: Trộn (Merge)

A

| | | | |
|----|----|----|----|
| 14 | 23 | 45 | 98 |
|----|----|----|----|

| | | | |
|---|----|----|----|
| 6 | 33 | 42 | 67 |
|---|----|----|----|

tempA

| | | | | | |
|---|----|----|----|----|----|
| 6 | 14 | 23 | 33 | 42 | 45 |
|---|----|----|----|----|----|

Merge

```
#define MAX_SIZE 500
```

```
void merge(int A[], int first, int mid, int last)
```

```
{  
    int tempA[MAX_SIZE];    // mang phu  
    int first1 = first; int last1 = mid;  
    int first2 = mid + 1; int last2 = last;  
    int index = first1;  
    for (; (first1 <= last1) && (first2 <= last2); ++index)  
    {  
        if (A[first1] < A[first2])  
            {tempA[index] = A[first1]; ++first1;}  
        else  
            { tempA[index] = A[first2]; ++first2;}  
    }  
  
    for (; first1 <= last1; ++first1, ++index)  
        tempA[index] = A[first1]; // sao not dãy con 1  
    for (; first2 <= last2; ++first2, ++index)  
        tempA[index] = A[first2]; // sao not dãy con 2  
    for (index = first; index <= last; ++index)  
        A[index] = tempA[index]; // sao tra mang ket qua  
} // end merge
```

Idea 2: Trộn (Merge)

A

| | | | |
|----|----|----|----|
| 14 | 23 | 45 | 98 |
|----|----|----|----|

| | | | |
|---|----|----|----|
| 6 | 33 | 42 | 67 |
|---|----|----|----|

tempA

| | | | | | | |
|---|----|----|----|----|----|----|
| 6 | 14 | 23 | 33 | 42 | 45 | 67 |
|---|----|----|----|----|----|----|

Merge

```
#define MAX_SIZE 500
```

```
void merge(int A[], int first, int mid, int last)
```

```
{  
    int tempA[MAX_SIZE];    // mang phu  
    int first1 = first; int last1 = mid;  
    int first2 = mid + 1; int last2 = last;  
    int index = first1;  
    for (; (first1 <= last1) && (first2 <= last2); ++index)  
    {  
        if (A[first1] < A[first2])  
            {tempA[index] = A[first1]; ++first1;}  
        else  
            { tempA[index] = A[first2]; ++first2;}  
    }  
  
    for (; first1 <= last1; ++first1, ++index)  
        tempA[index] = A[first1]; // sao not dãy con 1  
    for (; first2 <= last2; ++first2, ++index)  
        tempA[index] = A[first2]; // sao not dãy con 2  
    for (index = first; index <= last; ++index)  
        A[index] = tempA[index]; // sao tra mang ket qua  
} // end merge
```

Idea 2: Trộn (Merge)

A

| | | | |
|----|----|----|----|
| 14 | 23 | 45 | 98 |
|----|----|----|----|

| | | | |
|---|----|----|----|
| 6 | 33 | 42 | 67 |
|---|----|----|----|

tempA

| | | | | | | | |
|---|----|----|----|----|----|----|----|
| 6 | 14 | 23 | 33 | 42 | 45 | 67 | 98 |
|---|----|----|----|----|----|----|----|

Merge


```
#define MAX_SIZE 500
```

```
void merge(int A[], int first, int mid, int last)
```

```
{
```

```
    int tempA[MAX_SIZE];    // mang phu
```

```
    int first1 = first; int last1 = mid;
```

```
    int first2 = mid + 1; int last2 = last;
```

```
    int index = first1;
```

```
    for (; (first1 <= last1) && (first2 <= last2); ++index)
```

```
    {
```

```
        if (A[first1] < A[first2])
```

```
        {tempA[index] = A[first1]; ++first1;}
```

```
        else
```

```
        { tempA[index] = A[first2]; ++first2;}
```

```
    }
```

```
    for (; first1 <= last1; ++first1, ++index)
```

```
        tempA[index] = A[first1]; // sao not dăy con 1
```

```
    for (; first2 <= last2; ++first2, ++index)
```

```
        tempA[index] = A[first2]; // sao not dăy con 2
```

```
    for (index = first; index <= last; ++index)
```

```
        A[index] = tempA[index]; // sao tra mang ket qua
```

```
} // end merge
```

```
void mergeSort(int A[], int first, int last)
{
    if (first < last)
    {
        // chia thành hai dãy con
        int mid = (first + last)/2;    // chỉ số điểm giữa
        // sắp xếp dãy con trái A[first..mid]
        mergeSort(A, first, mid);
        // sắp xếp dãy con phải A[mid+1..last]
        mergeSort(A, mid+1, last);
        // Tron hai dãy con
        merge(A, first, mid, last);
    } // end if
} // end mergesort
```

```
void main()
{
    int a[5] = {8,4,3,2,1};
    mergeSort(a,0,4);
    for (int i = 0; i<5; i++)
        printf("%d \n",a[i]);
}
```

Summary of Sorting Algorithms

| Algorithm | Time | Notes |
|----------------|---------------|--|
| selection-sort | $O(n^2)$ | <ul style="list-style-type: none">◆ slow◆ in-place◆ for small data sets (< 1K) |
| insertion-sort | $O(n^2)$ | <ul style="list-style-type: none">◆ slow◆ in-place◆ for small data sets (< 1K) |
| merge-sort | $O(n \log n)$ | <ul style="list-style-type: none">◆ fast◆ not in-place → require extra space $\approx n$◆ sequential data access◆ for huge data sets (> 1M) |

Các thuật toán sắp xếp

1. Sắp xếp chèn (Insertion sort)
 2. Sắp xếp chọn (Selection sort)
 3. Sắp xếp trộn (Merge sort)
 4. Sắp xếp nhanh (Quick sort)
- } Devide and conquer

4. Sắp xếp nhanh

4.1. 2-way partitioning

4.2. Bentley-McIlroy 3-way partitioning

4.3. Hàm qsort có sẵn trong thư viện stdlib.h

4. Sắp xếp nhanh

4.1. 2-way partitioning

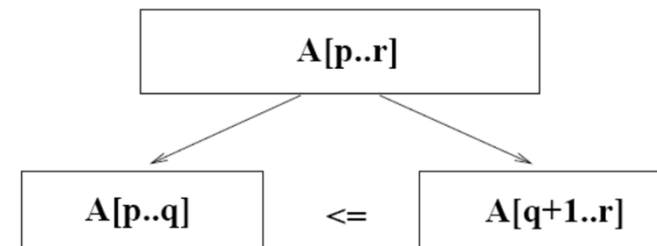
4.2. Bentley-McIlroy 3-way partitioning

4.3. Hàm qsort có sẵn trong thư viện stdlib.h

4. Sắp xếp nhanh (Quick sort): 2-way partitioning

Thuật toán có thể mô tả đệ qui như sau (có dạng tương tự như merge sort):

1. **Neo đệ qui** (Base case). Nếu dãy chỉ còn không quá một phần tử thì nó là dãy được sắp và trả lại ngay dãy này mà không phải làm gì cả.
2. **Chia** (Divide):
 - Chọn một phần tử trong dãy và gọi nó là **phần tử chốt p** (pivot).
 - Chia dãy đã cho ra thành hai dãy con: Dãy con trái (L) gồm những phần tử \leq phần tử chốt, còn dãy con phải (R) gồm các phần tử \geq phần tử chốt. Thao tác này được gọi là "**Phân đoạn**" (Partition).

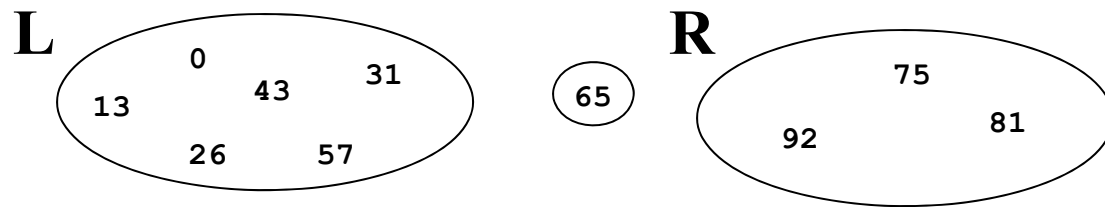
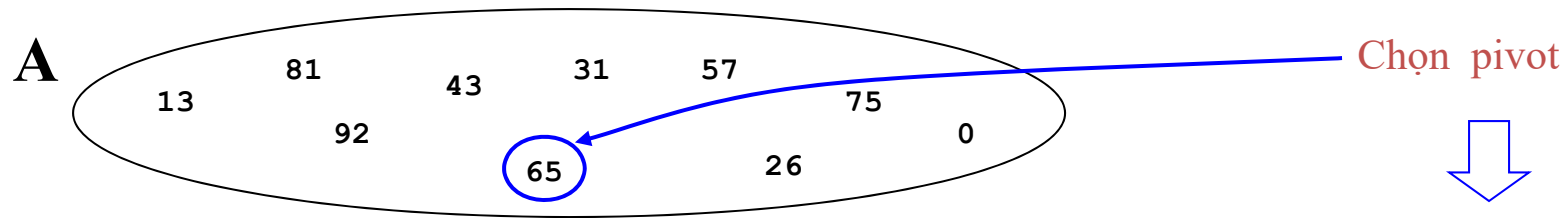


3. **Trị** (Conquer): Lặp lại một cách đệ qui thuật toán QuickSort đối với hai dãy con $L = A[p \dots q]$ và $R = A[q+1 \dots r]$.
4. **Tổng hợp** (Combine): Dãy được sắp xếp là $L \ p \ R$.

Ngược lại với Merge Sort, trong Quick Sort thao tác chia là phức tạp, nhưng thao tác tổng hợp lại đơn giản.

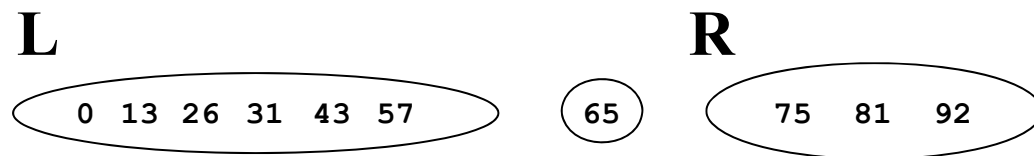
Điểm mấu chốt để thực hiện Quick Sort chính là thao tác chia. Phụ thuộc vào thuật toán thực hiện thao tác này mà ta có các dạng Quick Sort cụ thể.

4. Sắp xếp nhanh (Quick sort): Ví dụ minh họa

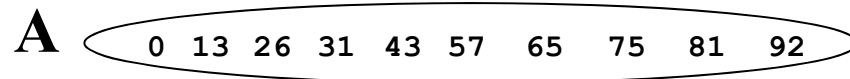


Phân đoạn A thành 2 nửa:

- Nửa trái: gồm các phần tử $\leq \text{pivot}$
- Nửa phải: gồm các phần tử $\geq \text{pivot}$



QuickSort(L) và
QuickSort(R)



OK! A được sắp

Sơ đồ tổng quát sắp xếp nhanh (Quick sort)

- Sơ đồ tổng quát của QS có thể mô tả như sau:

Quick-Sort($A, Left, Right$)

```
1.  if ( $Left < Right$ ) {  
2.       $Pivot = Partition(A, Left, Right);$   
3.      Quick-Sort( $A, Left, Pivot - 1$ );  
4.      Quick-Sort( $A, Pivot + 1, Right$ );  
5.  }
```

Hàm $Partition(A, Left, Right)$ thực hiện chia $A[Left..Right]$ thành hai đoạn $A[Left..Pivot - 1]$ và $A[Pivot + 1..Right]$ sao cho:

- Các phần tử trong $A[Left..Pivot - 1]$ là nhỏ hơn hoặc bằng $A[Pivot]$
- Các phần tử trong $A[Pivot + 1..Right]$ là lớn hơn hoặc bằng $A[Pivot]$.

Lệnh gọi thực hiện thuật toán **Quick-Sort($A, 1, n$)**

Sơ đồ tổng quát sắp xếp nhanh (Quick sort)

Knuth cho rằng khi dãy con chỉ còn một số lượng không lớn phần tử (theo ông là không quá 9 phần tử) thì ta nên sử dụng các thuật toán đơn giản để sắp xếp dãy này, chứ không nên tiếp tục chia nhỏ. Thuật toán trong tình huống như vậy có thể mô tả như sau:

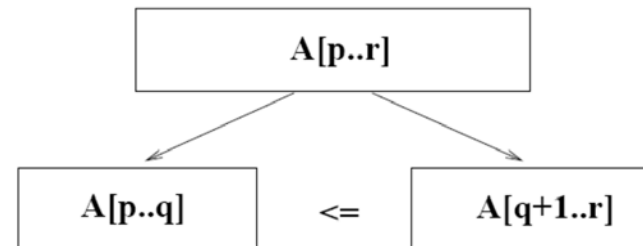
Quick-Sort(A, Left, Right)

1. **if** (*Right - Left* < n_0)
2. InsertionSort(*A, Left, Right*);
3. **else** {
4. *Pivot* = Partition(*A, Left, Right*);
5. Quick-Sort(*A, Left, Pivot - 1*);
6. Quick-Sort(*A, Pivot + 1, Right*);
7. }

Thao tác chia trong sắp xếp nhanh (Quick sort)

Chia (Divide):

- Chọn một phần tử trong dãy và gọi nó là **phần tử chốt p** (pivot).
- Chia dãy đã cho ra thành hai dãy con: Dãy con trái (L) gồm những phần tử \leq phần tử chốt, còn dãy con phải (R) gồm các phần tử \geq phần tử chốt. Thao tác này được gọi là "**Phân đoạn**" (Partition).

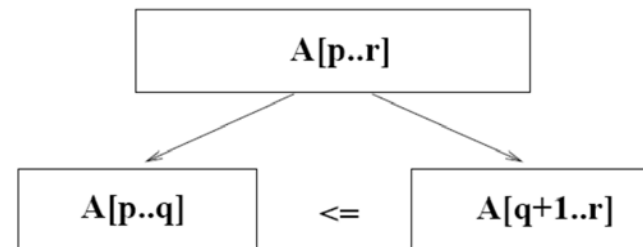


- Thao tác phân đoạn có thể cài đặt (tại chỗ) với thời gian $\Theta(n)$.
- Việc chọn phần tử chốt có vai trò quyết định đối với hiệu quả của thuật toán. Tốt nhất nếu chọn được phần tử chốt là phần tử có giá trị trung bình trong danh sách (ta gọi phần tử như vậy là **trung vị/median**). Khi đó, sau **$\log_2 n$** lần phân đoạn ta sẽ đạt tới danh sách với kích thước bằng 1. Tuy nhiên, điều đó rất khó thực hiện. Người ta thường sử dụng các cách chọn phần tử chốt sau đây:
 - Chọn phần tử trái nhất (đứng đầu) làm phần tử chốt.
 - Chọn phần tử phải nhất (đứng cuối) làm phần tử chốt.
 - Chọn phần tử đứng giữa danh sách làm phần tử chốt.
 - Chọn phần tử trung vị trong 3 phần tử đứng đầu, đứng giữa và đứng cuối làm phần tử chốt (Knuth).
 - Chọn ngẫu nhiên một phần tử làm phần tử chốt.

Thao tác chia trong sắp xếp nhanh (Quick sort)

Chia (Divide):

- Chọn một phần tử trong dãy và gọi nó là **phần tử chốt p (pivot)**.
- Chia dãy đã cho ra thành hai dãy con: Dãy con trái (L) gồm những phần tử \leq phần tử chốt, còn dãy con phải (R) gồm các phần tử \geq phần tử chốt. Thao tác này được gọi là "**Phân đoạn**" (Partition).



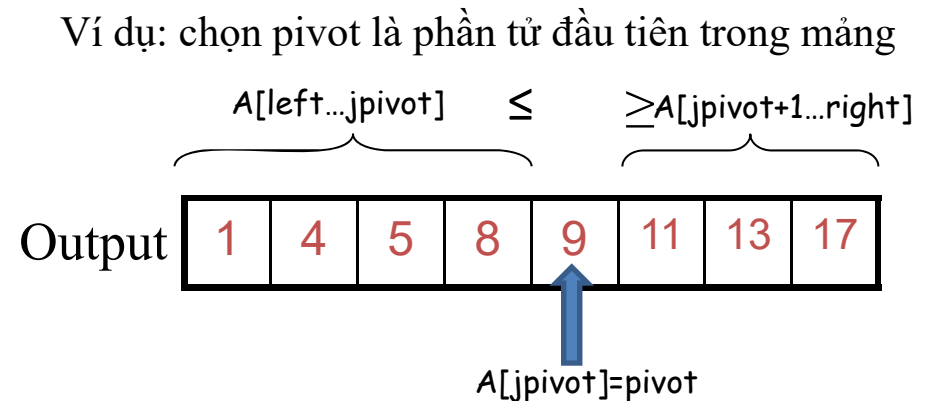
- Hiệu quả của thuật toán phụ thuộc rất nhiều vào việc phần tử nào được chọn làm phần tử chốt:
 - Thời gian tính trong tình huống tồi nhất của QS là $O(n^2)$. Trường hợp xấu nhất xảy ra khi danh sách là đã được sắp xếp và phần tử chốt được chọn là phần tử trái nhất của dãy.
 - Nếu phần tử chốt được chọn ngẫu nhiên, thì QS có độ phức tạp tính toán là $O(n \log n)$.

Thuật toán phân đoạn: 2-way partitioning

Ta xây dựng hàm **Partition(A, left, right)** làm việc sau:

- **Input:** Mảng $A[\text{left} .. \text{right}]$.
- **Output:** Phân bố lại các phần tử của mảng đầu vào dựa vào phần tử *pivot* được chọn và trả lại chỉ số *jpivot* thoả mãn:
 - $a[\text{jpivot}]$ chứa giá trị của *pivot*,
 - $a[i] \leq a[\text{jpivot}]$, với mọi $\text{left} \leq i < \text{jpivot}$,
 - $a[j] \geq a[\text{jpivot}]$, với mọi $\text{jpivot} < j \leq \text{right}$.

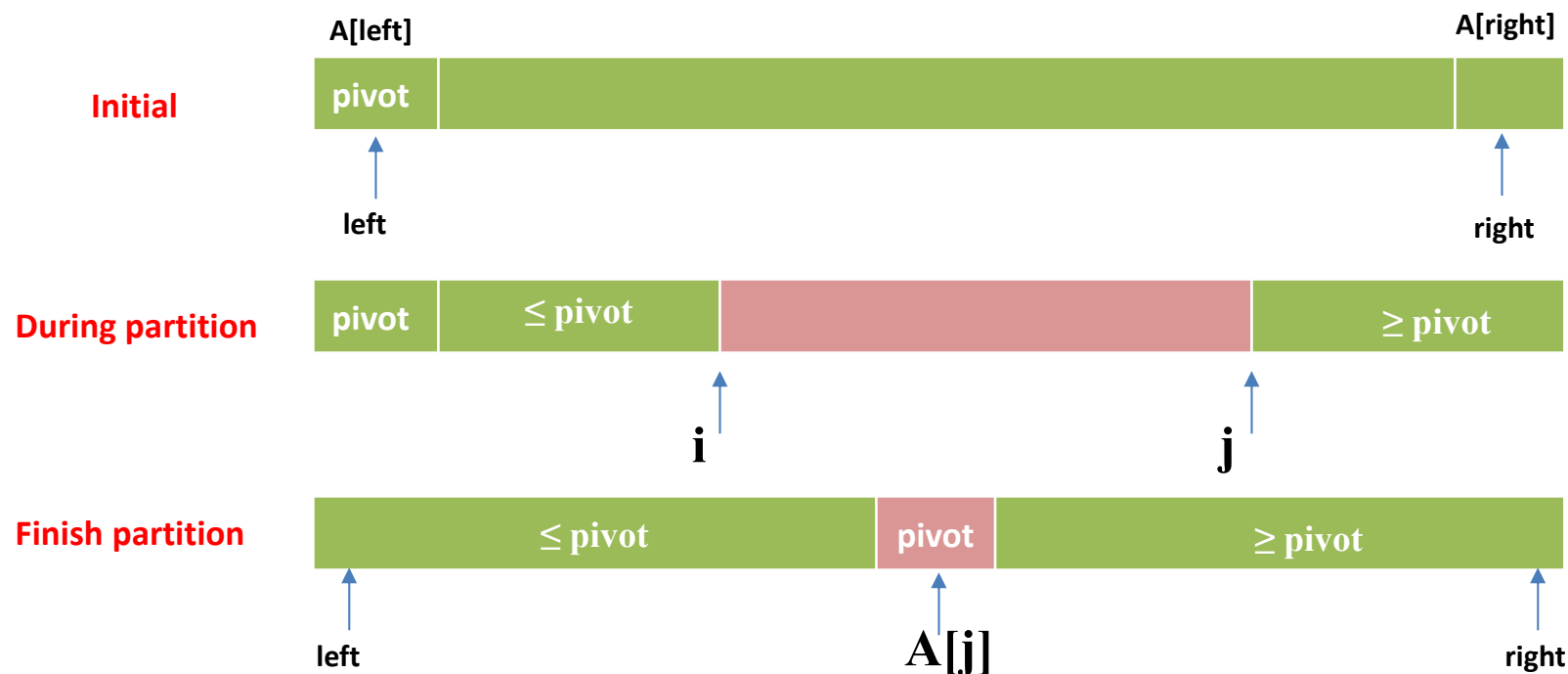
trong đó *pivot* là giá trị phần tử chốt được chọn



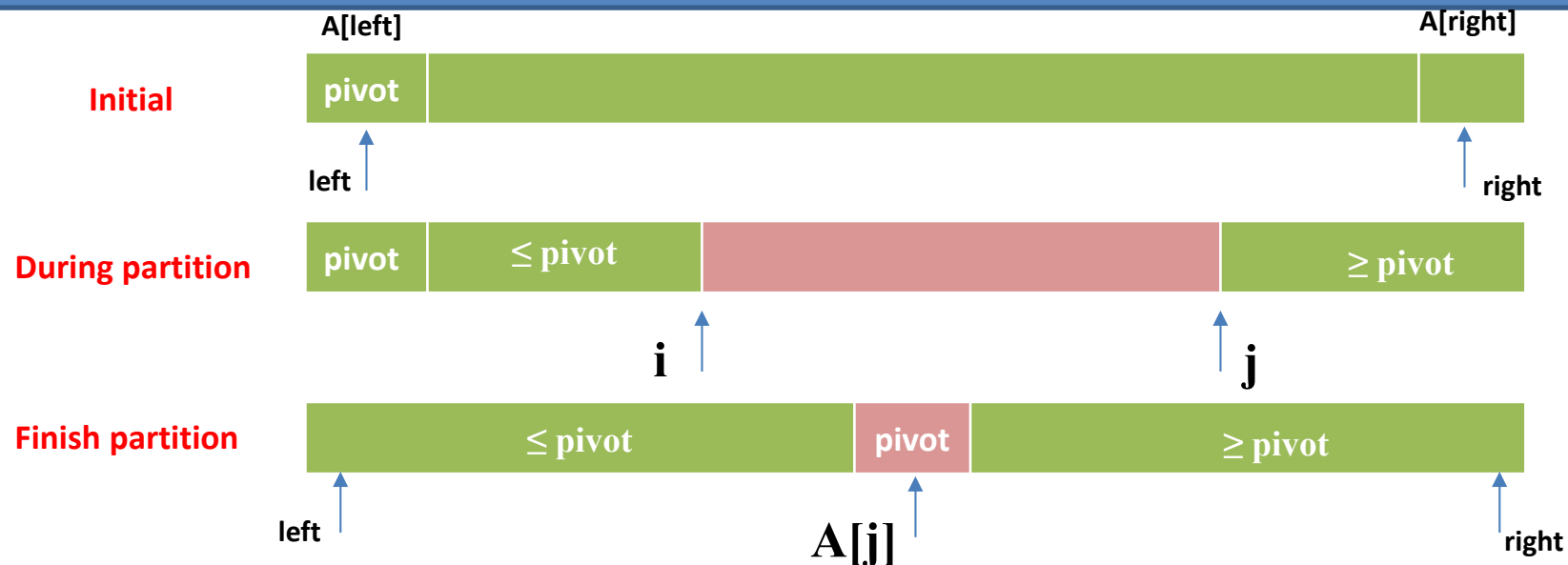
Phần tử chốt là phần tử đứng đầu: : 2-way partitioning

Hàm phân đoạn (Partition) dịch chuyển các phần tử của dãy cần sắp xếp $A[\text{left}] \dots A[\text{right}]$ để thu được một dãy mới thỏa mãn 3 điều kiện sau:

- Phần tử chốt pivot sẽ được đặt về đúng vị trí của nó (kí hiệu là vị trí thứ j) trong dãy sắp xếp
- Các phần tử từ $A[\text{left}] \dots A[j-1]$ đều \leq phần tử chốt
- Các phần tử từ $A[j+1] \dots A[\text{right}]$ đều \geq phần tử chốt



Phần tử chốt là phần tử đứng đầu: : 2-way partitioning



- Chọn phần tử chốt là phần tử đứng đầu dãy $\text{pivot} = A[\text{left}]$ (phần tử này khi kết thúc hàm Partition, nó sẽ được đặt về vị trí j là vị trí thực sự của nó trong dãy A khi các phần tử được sắp xếp tăng dần)
- $i = \text{left} + 1$; Duyệt từ trái: Duyệt từ phần tử $A[i]$ về cuối dãy để tìm phần tử đầu tiên $\text{FIRST1} \geq \text{pivot}$
- $j = \text{right}$; Duyệt từ phải: Duyệt từ phần tử $A[j]$ về đầu dãy để tìm phần tử đầu tiên $\text{FIRST2} \leq \text{pivot}$
- SWAP (FIRST1 , FIRST2) vì 2 phần tử này đang nằm không đúng vị trí của nó
- Tiếp tục duyệt từ trái và từ phải để hoán đổi vị trí các phần tử nếu cần thiết, dừng duyệt khi $i \geq j$
- Cuối cùng, SWAP($A[\text{left}] = \text{pivot}$, $A[j]$) – hoán đổi vị trí phần tử chốt với phần tử cuối cùng của nửa trái
- return j

Ví dụ quá trình thực hiện hàm partition khi chọn phần tử chốt là phần tử đứng đầu dãy

- $i = \text{left} + 1$; Duyệt từ trái: Duyệt từ phần tử $A[i]$ về cuối dãy để tìm phần tử đầu tiên $\text{FIRST1} \geq \text{pivot}$
- $j = \text{right}$; Duyệt từ phải: Duyệt từ phần tử $A[j]$ về đầu dãy để tìm phần tử đầu tiên $\text{FIRST2} \leq \text{pivot}$
- SWAP (FIRST1 , FIRST2) vì 2 phần tử này đang nằm không đúng vị trí của nó
- Tiếp tục duyệt từ trái và từ phải để hoán đổi vị trí các phần tử nếu cần thiết, dừng duyệt khi $i \geq j$
- Cuối cùng, SWAP($A[\text{left}] = \text{pivot}$, $A[j]$) – hoán đổi vị trí phần tử chốt với phần tử cuối cùng của nửa trái

| | | A[] | | | | | | | | | | | | | | | | | |
|-----------------------|--|------|----|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|
| | | i | j | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| initial values | | 0 | 16 | K | R | A | T | E | L | E | P | U | I | M | Q | C | X | O | S |
| scan left, scan right | | 1 | 12 | K | R | A | T | E | L | E | P | U | I | M | Q | C | X | O | S |
| exchange | | 1 | 12 | K | C | A | T | E | L | E | P | U | I | M | Q | R | X | O | S |
| scan left, scan right | | 3 | 9 | K | C | A | T | E | L | E | P | U | I | M | Q | R | X | O | S |
| exchange | | 3 | 9 | K | C | A | I | E | L | E | P | U | T | M | Q | R | X | O | S |
| scan left, scan right | | 5 | 6 | K | C | A | I | E | L | E | P | U | T | M | Q | R | X | O | S |
| exchange | | 5 | 6 | K | C | A | I | E | E | L | P | U | T | M | Q | R | X | O | S |
| scan left, scan right | | 6 | 5 | K | C | A | I | E | E | L | P | U | T | M | Q | R | X | O | S |
| final exchange | | 6 | 5 | E | C | A | I | E | K | L | P | U | T | M | Q | R | X | O | S |
| result | | | 5 | E | C | A | I | E | K | L | P | U | T | M | Q | R | X | O | S |

Partitioning trace (array contents before and after each exchange)

Phần tử chốt là phần tử đứng đầu dãy: 2-way partitioning

Partition(A, left, right)

i = *left*; *j* = *right* + 1;

pivot = *A*[*left*];



pivot được chọn là phần tử đứng đầu

while (true) **do**

{

i = *i* + 1;

 //Tìm từ trái sang phần tử đầu tiên \geq pivot:

while *i* \leq *right* **and** *A*[*i*] < *pivot* **do** *i* = *i* + 1;

j = *j* - 1;

 //Tìm từ phải sang phần tử đầu tiên \leq pivot:

while *j* \geq *left* **and** *pivot* < *A*[*j*] **do** *j* = *j* - 1;

if (*i* \geq *j*) **break**;

swap(*A*[*i*] , *A*[*j*]);

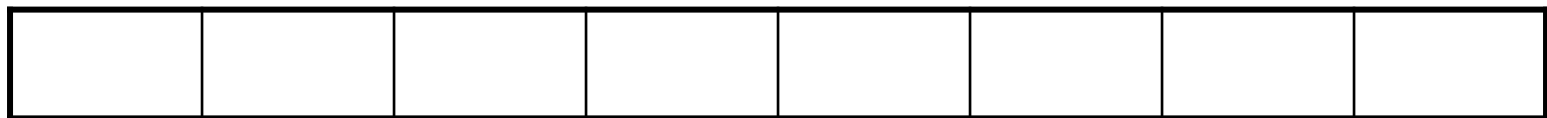


j là chỉ số (jpivot) cần trả lại, do đó cần đổi chỗ *a*[*left*] và *a*[*j*]

}

swap(*A*[*j*], *A*[*left*]);

return *j*;

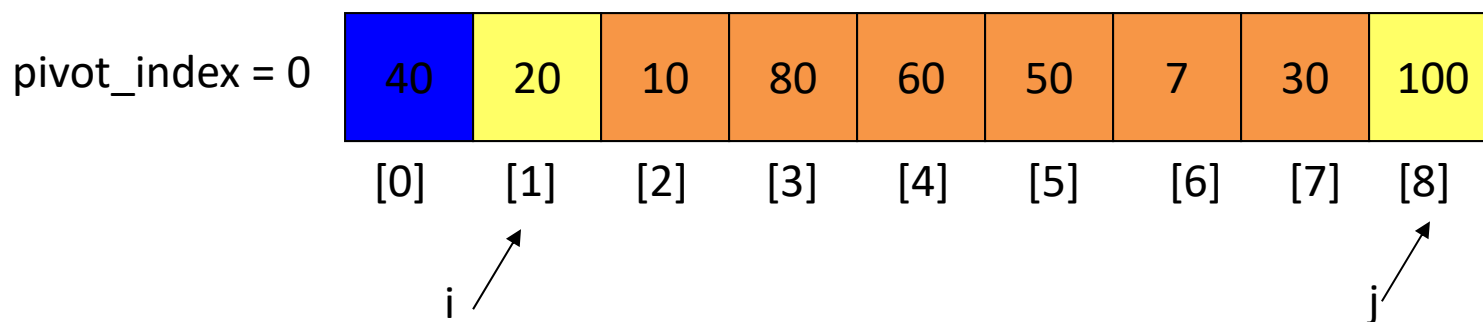


Sau khi chọn *pivot*, dịch các con trỏ *i* và *j* từ đầu và cuối mảng và đổi chỗ cặp phần tử thoả mãn $a[i] > pivot$ và $a[j] \leq pivot$

i

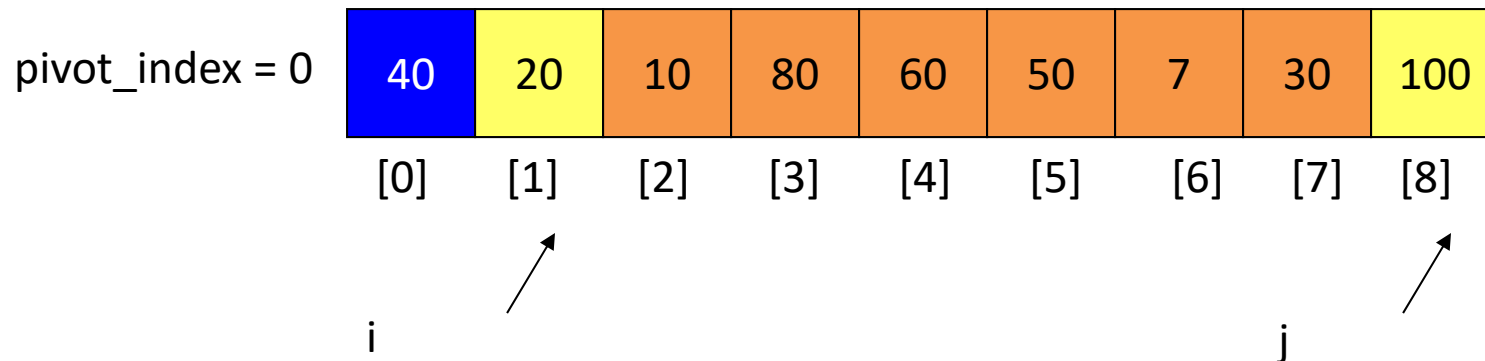
j

Phần tử chốt là phần tử đứng đầu dãy: 2-way partitioning



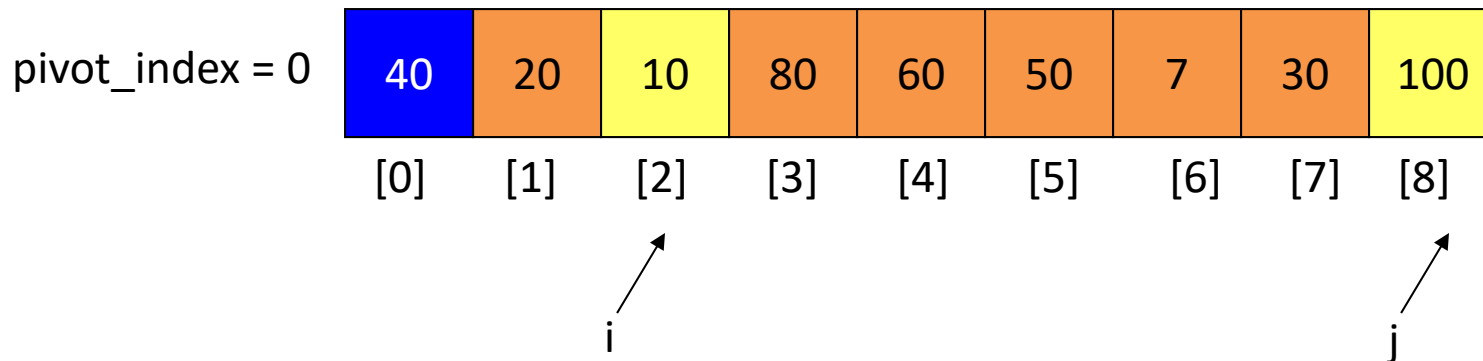
Phần tử chốt là phần tử đứng đầu dãy: 2-way partitioning

```
1. While i <= right and A[i] < pivot
    ++i
```



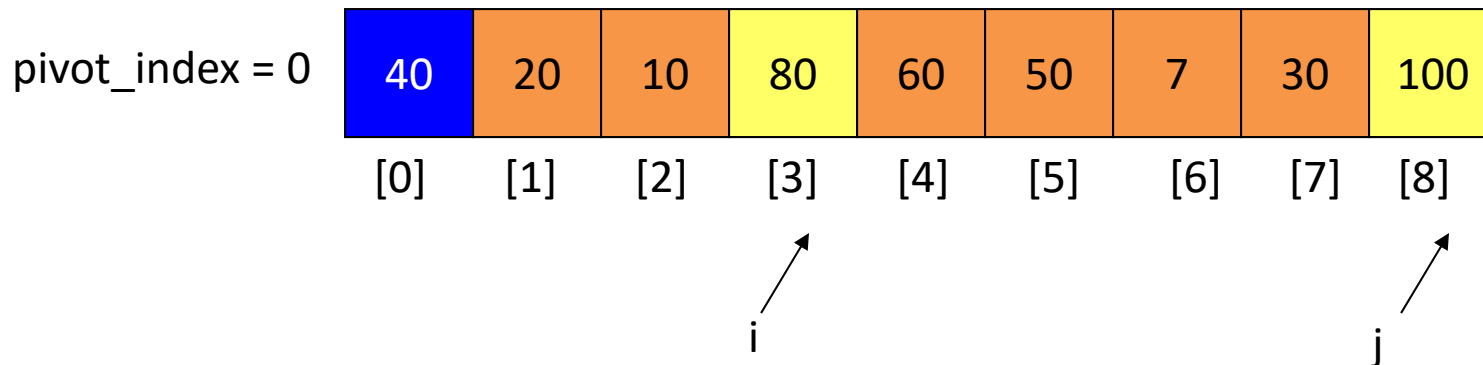
Phần tử chốt là phần tử đứng đầu dãy: 2-way partitioning

1. While $i \leq \text{right}$ and $A[i] < \text{pivot}$
 $++i$



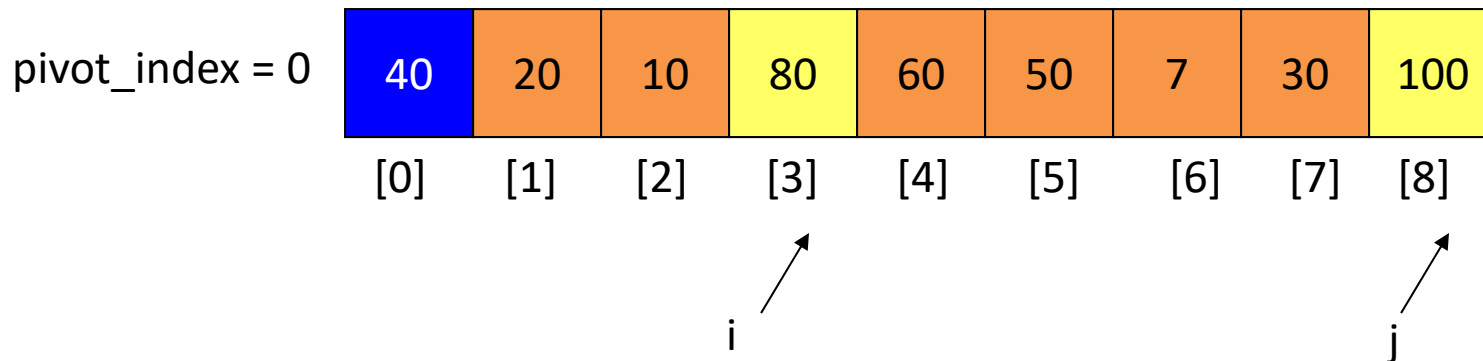
Phần tử chốt là phần tử đứng đầu dãy: 2-way partitioning

1. While $i \leq \text{right}$ and $A[i] < \text{pivot}$
 $++i$



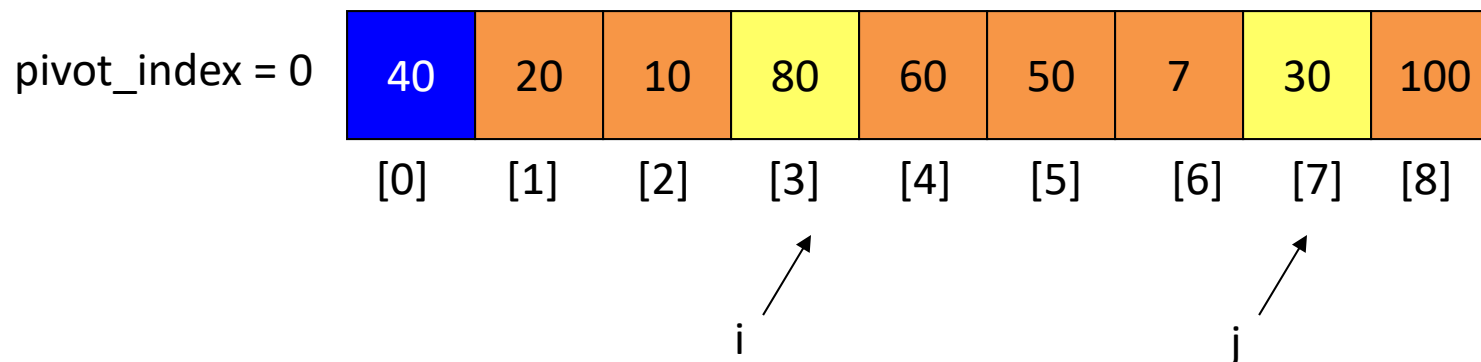
Phần tử chốt là phần tử đứng đầu dãy: 2-way partitioning

1. While $i \leq \text{right}$ and $A[i] < \text{pivot}$
 $++i$
2. While $j \geq \text{left}$ and $A[j] > \text{pivot}$
 $--j$



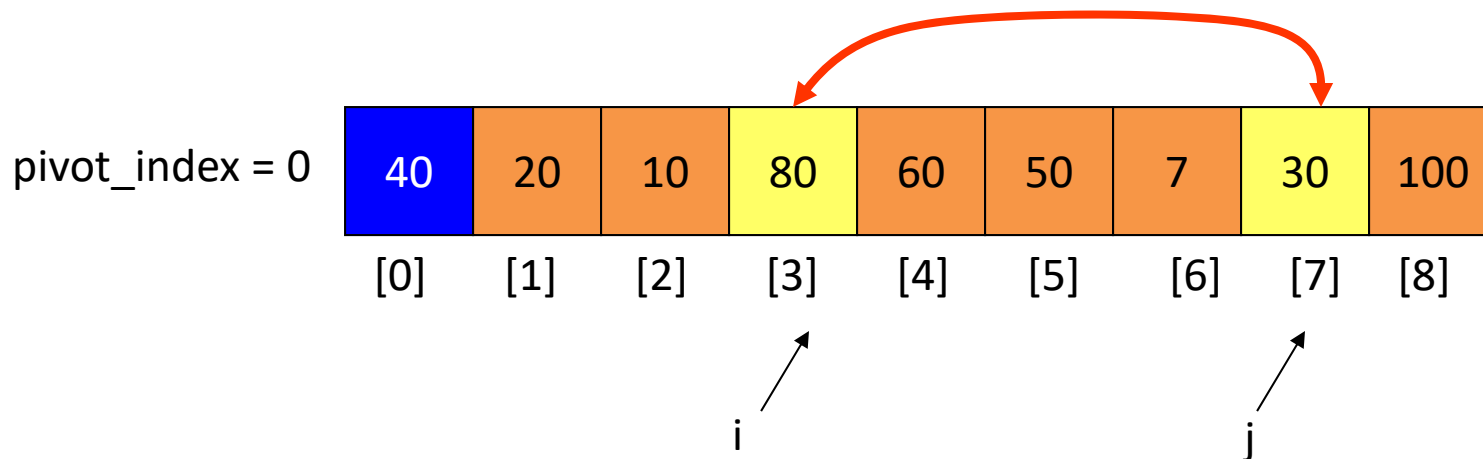
Phần tử chốt là phần tử đứng đầu dãy: 2-way partitioning

1. While $i \leq \text{right}$ and $A[i] < \text{pivot}$
 $++i$
2. While $j \geq \text{left}$ and $A[j] > \text{pivot}$
 $--j$



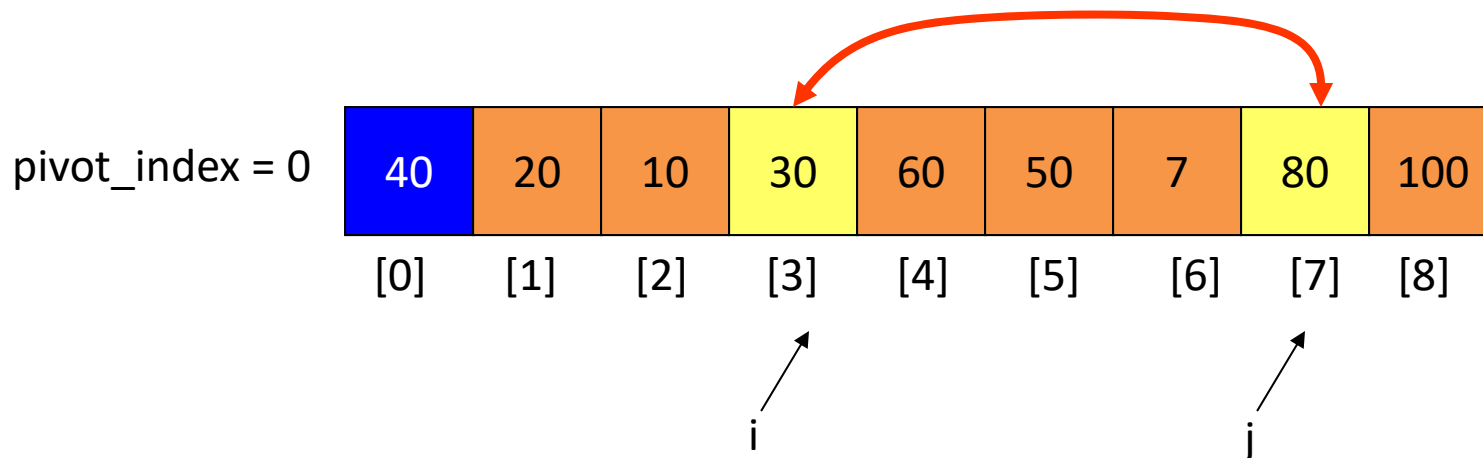
Phần tử chốt là phần tử đứng đầu dãy: 2-way partitioning

1. While $i \leq \text{right}$ and $A[i] < \text{pivot}$
 $++i$
2. While $j \geq \text{left}$ and $A[j] > \text{pivot}$
 $--j$
3. If $i < j$
 $\text{swap}(A[i], A[j])$



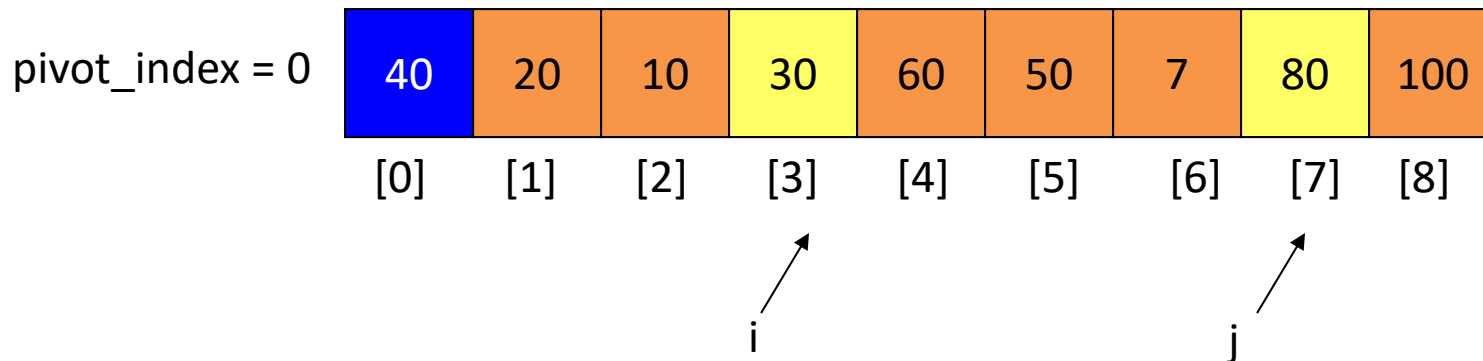
Phần tử chốt là phần tử đứng đầu dãy: 2-way partitioning

1. While $i \leq \text{right}$ and $A[i] < \text{pivot}$
 $++i$
2. While $j \geq \text{left}$ and $A[j] > \text{pivot}$
 $--j$
3. If $i < j$
 $\text{swap}(A[i], A[j])$



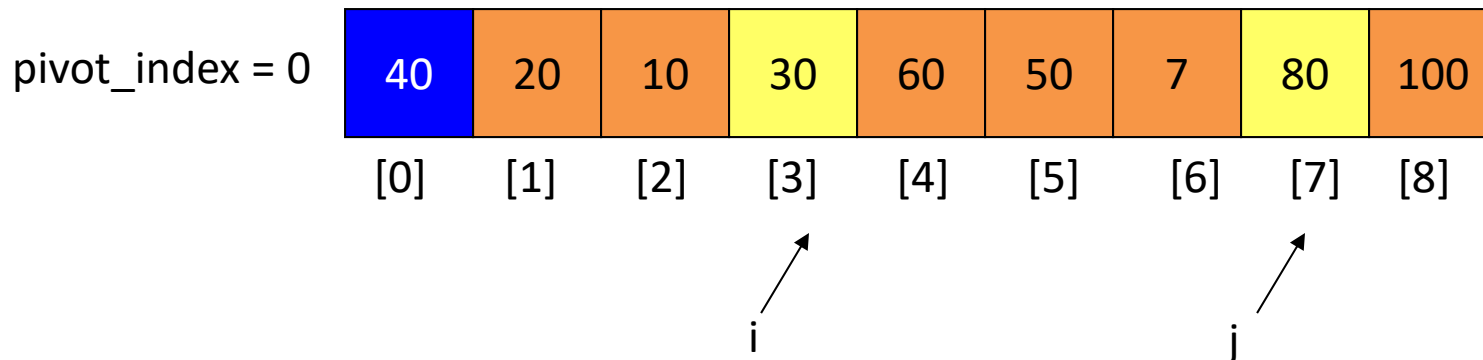
Phần tử chốt là phần tử đứng đầu dãy: 2-way partitioning

1. While $i \leq \text{right}$ and $A[i] < \text{pivot}$
 $++i$
2. While $j \geq \text{left}$ and $A[j] > \text{pivot}$
 $--j$
3. If $i < j$
 $\text{swap}(A[i], A[j])$
4. While $j > i$, go to 1.



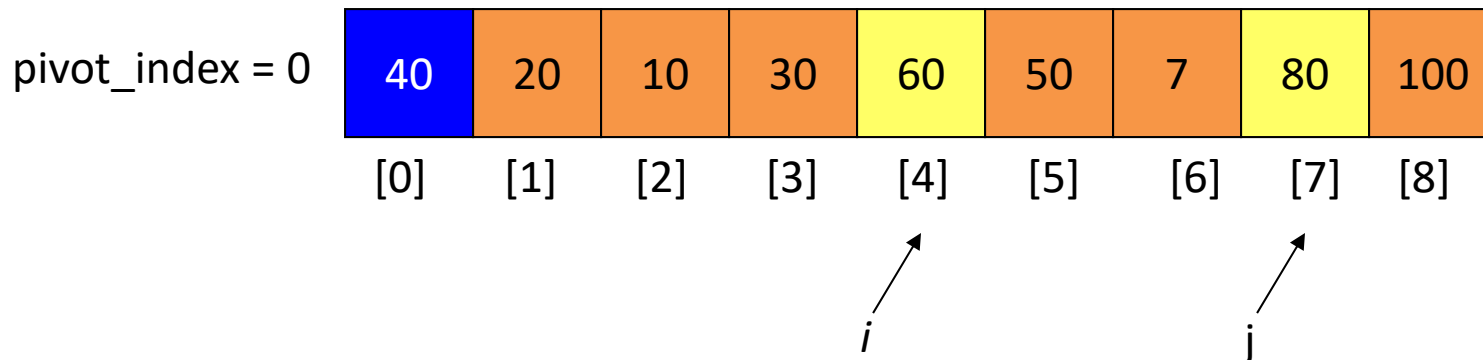
Phần tử chốt là phần tử đứng đầu

- 1. While $i \leq \text{right}$ and $A[i] < \text{pivot}$
 $++i$
- 2. While $j \geq \text{left}$ and $A[j] > \text{pivot}$
 $--j$
- 3. If $i < j$
 $\text{swap}(A[i], A[j])$
- 4. While $j > i$, go to 1.



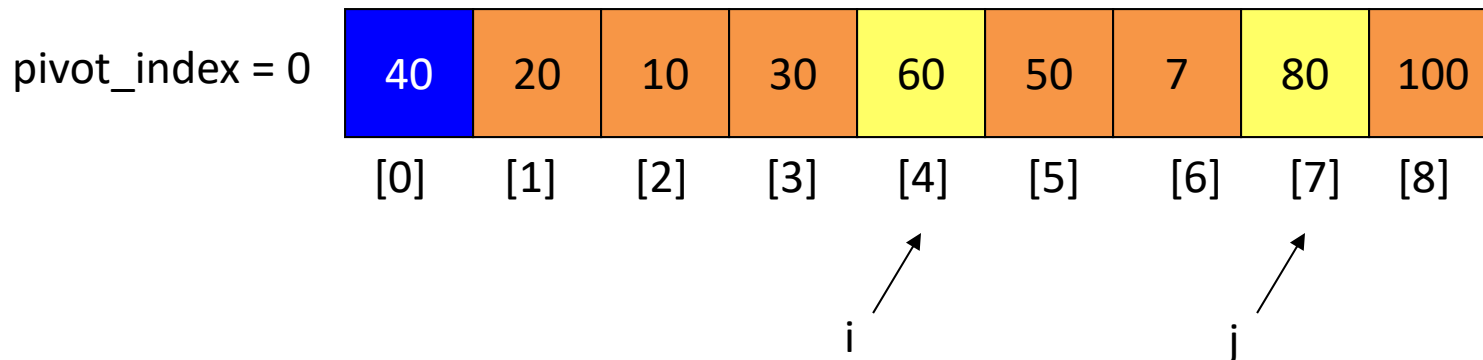
Phần tử chốt là phần tử đứng đầu

- 1. While $i \leq \text{right}$ and $A[i] < \text{pivot}$
 $++i$
- 2. While $j \geq \text{left}$ and $A[j] > \text{pivot}$
 $--j$
- 3. If $i < j$
 $\text{swap}(A[i], A[j])$
- 4. While $j > i$, go to 1.



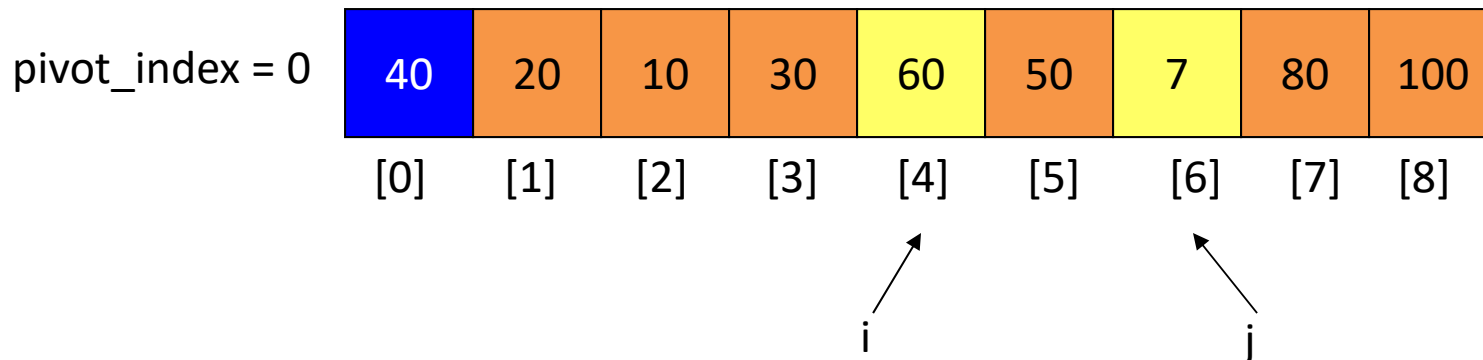
Phần tử chốt là phần tử đứng đầu

1. While $i \leq \text{right}$ and $A[i] < \text{pivot}$
 $++i$
- 2. While $j \geq \text{left}$ and $A[j] > \text{pivot}$
 $--j$
3. If $i < j$
 $\text{swap}(A[i], A[j])$
4. While $j > i$, go to 1.



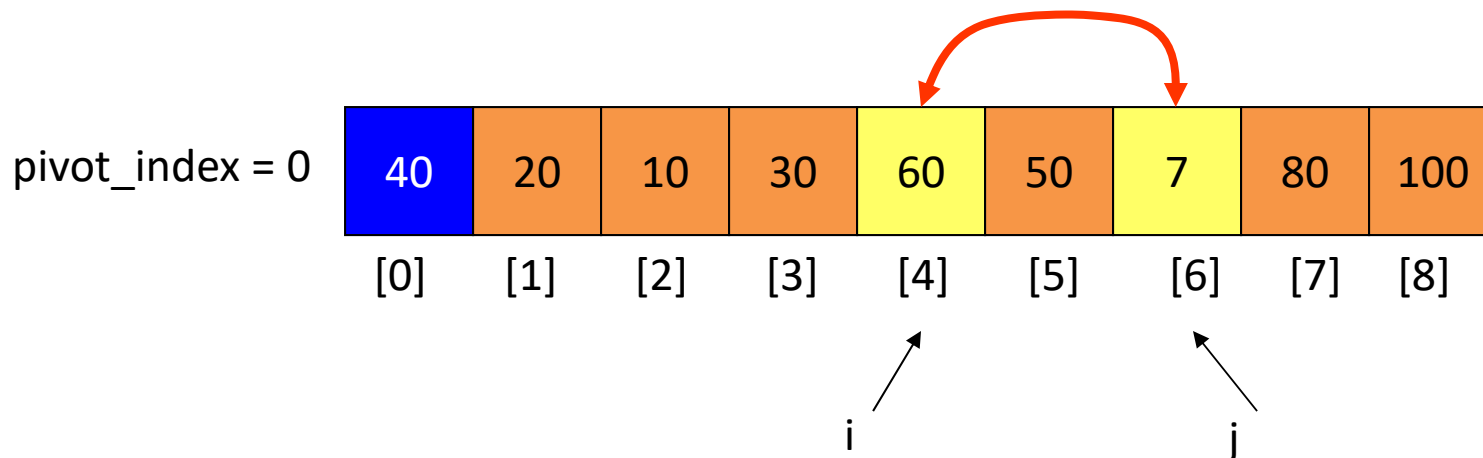
Phần tử chốt là phần tử đứng đầu

1. While $i \leq \text{right}$ and $A[i] < \text{pivot}$
 $++i$
- 2. While $j \geq \text{left}$ and $A[j] > \text{pivot}$
 $--j$
3. If $i < j$
 $\text{swap}(A[i], A[j])$
4. While $j > i$, go to 1.



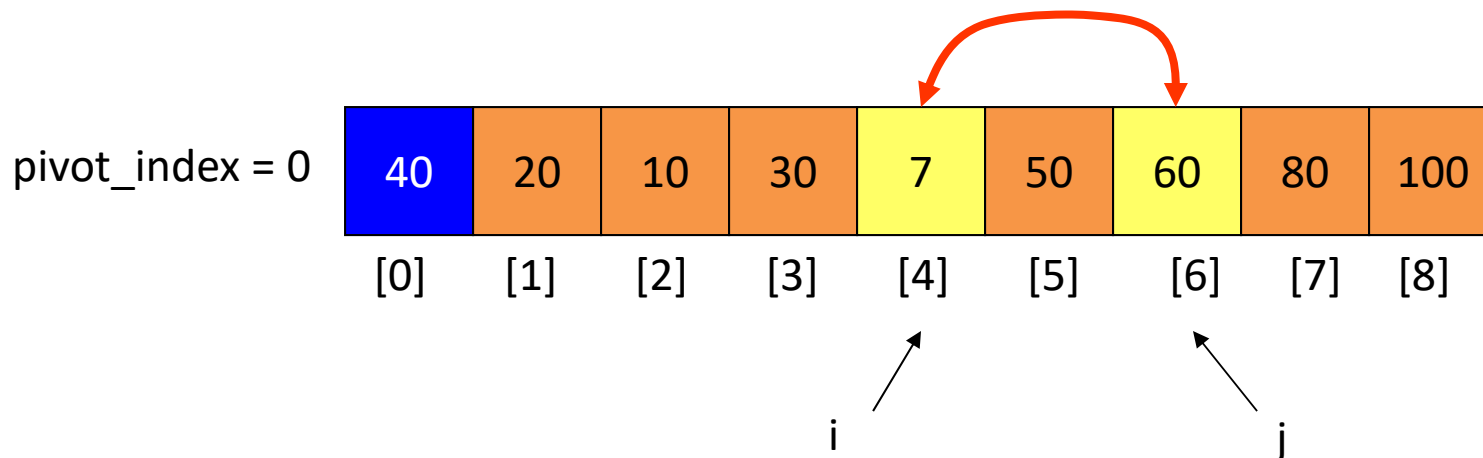
Phần tử chốt là phần tử đứng đầu

1. While $i \leq \text{right}$ and $A[i] < \text{pivot}$
 $++i$
2. While $j \geq \text{left}$ and $A[j] > \text{pivot}$
 $--j$
- 3. If $i < j$
 $\text{swap}(A[i], A[j])$
4. While $j > i$, go to 1.



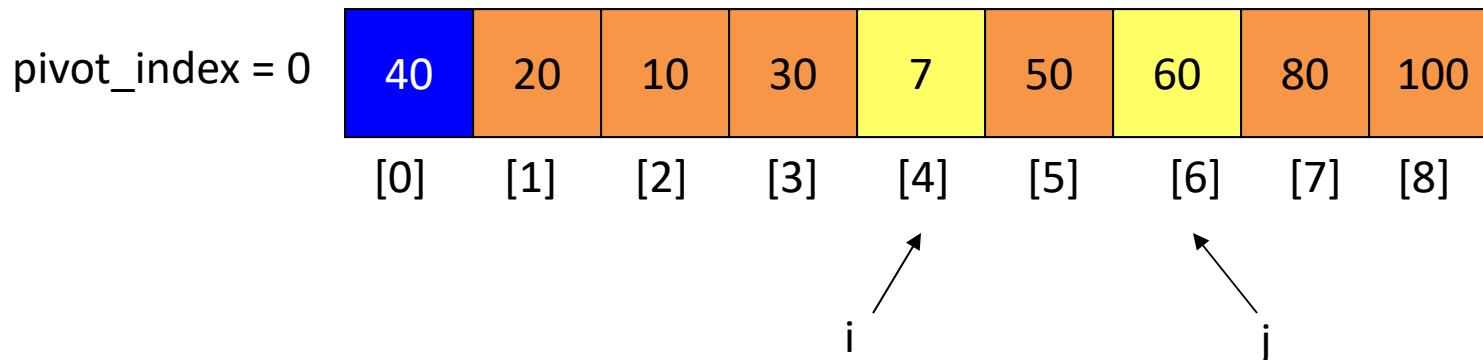
Phần tử chốt là phần tử đứng đầu

1. While $i \leq \text{right}$ and $A[i] < \text{pivot}$
 $++i$
2. While $j \geq \text{left}$ and $A[j] > \text{pivot}$
 $--j$
- 3. If $i < j$
 $\text{swap}(A[i], A[j])$
4. While $j > i$, go to 1.



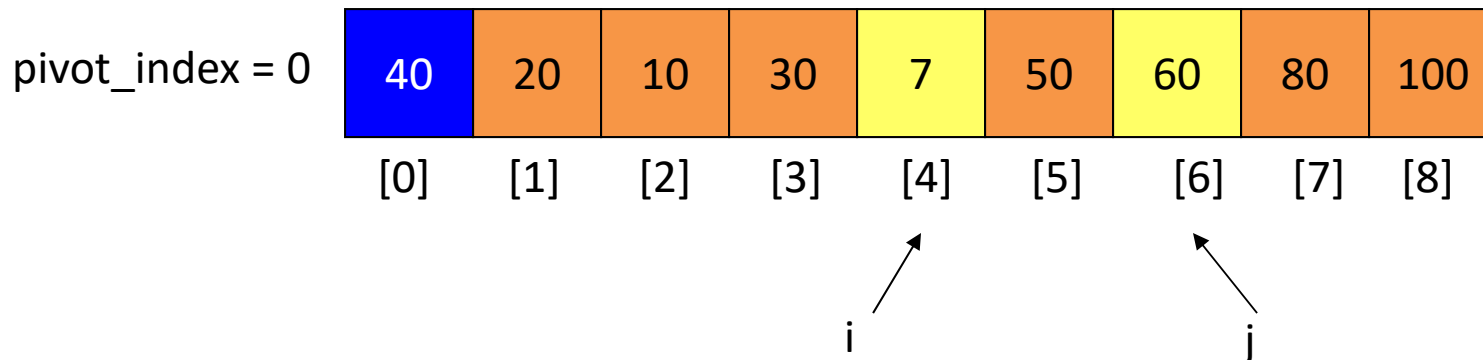
Phần tử chốt là phần tử đứng đầu

1. While $i \leq \text{right}$ and $A[i] < \text{pivot}$
 $++i$
2. While $j \geq \text{left}$ and $A[j] > \text{pivot}$
 $--j$
3. If $i < j$
 $\text{swap}(A[i], A[j])$
- 4. While $j > i$, go to 1.



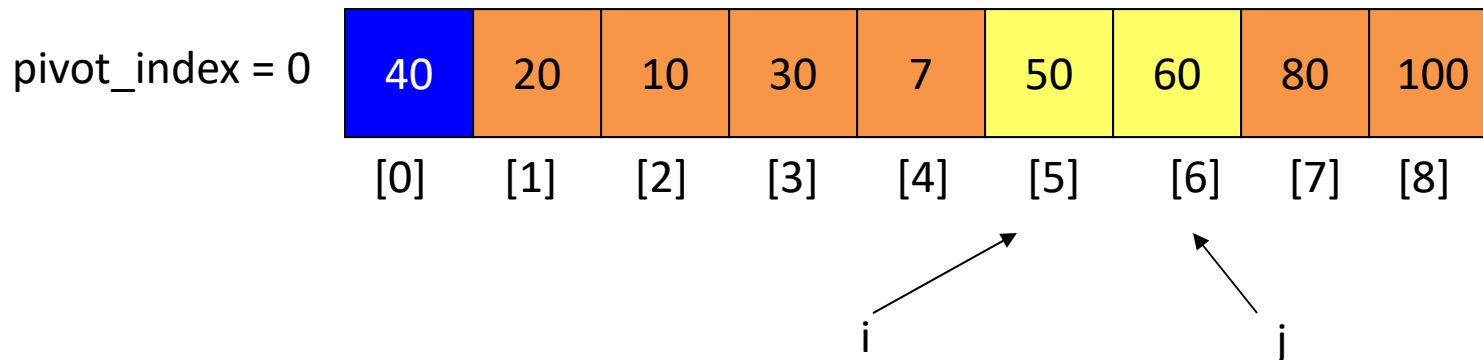
Phần tử chốt là phần tử đứng đầu

- 1. While $i \leq \text{right}$ and $A[i] < \text{pivot}$
 $++i$
- 2. While $j \geq \text{left}$ and $A[j] > \text{pivot}$
 $--j$
- 3. If $i < j$
 $\text{swap}(A[i], A[j])$
- 4. While $j > i$, go to 1.



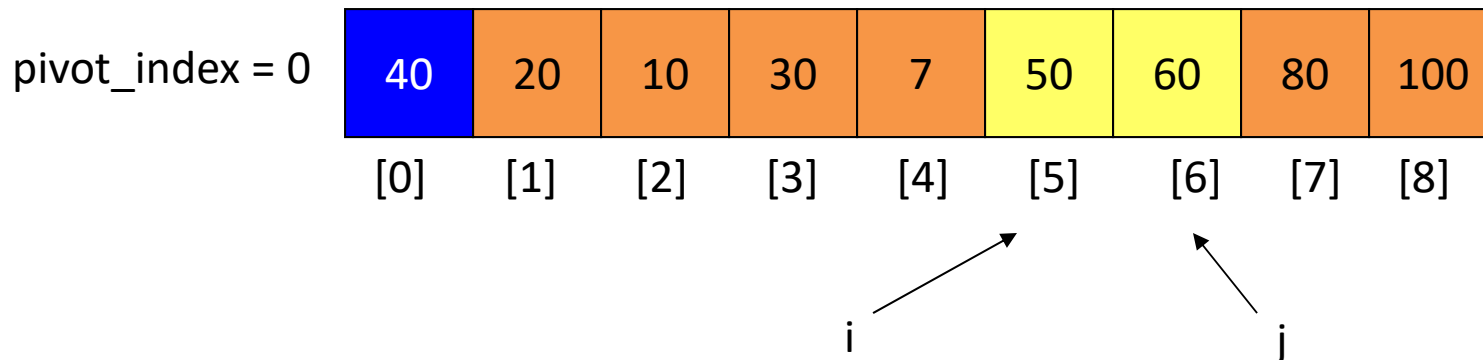
Phần tử chốt là phần tử đứng đầu

- 1. While $i \leq \text{right}$ and $A[i] < \text{pivot}$
 $++i$
- 2. While $j \geq \text{left}$ and $A[j] > \text{pivot}$
 $--j$
- 3. If $i < j$
 $\text{swap}(A[i], A[j])$
- 4. While $j > i$, go to 1.



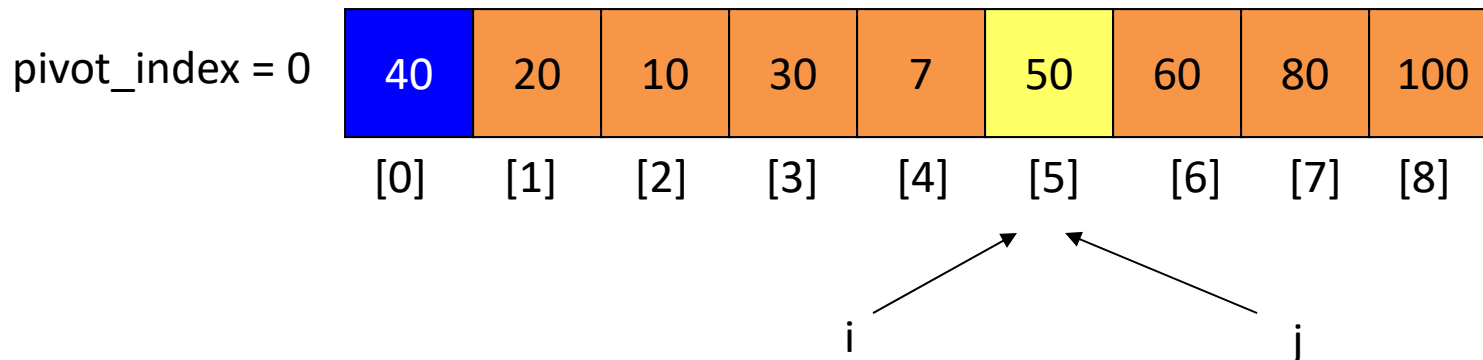
Phần tử chốt là phần tử đứng đầu

1. While $i \leq \text{right}$ and $A[i] < \text{pivot}$
 $++i$
- 2. While $j \geq \text{left}$ and $A[j] > \text{pivot}$
 $--j$
3. If $i < j$
 $\text{swap}(A[i], A[j])$
4. While $j > i$, go to 1.



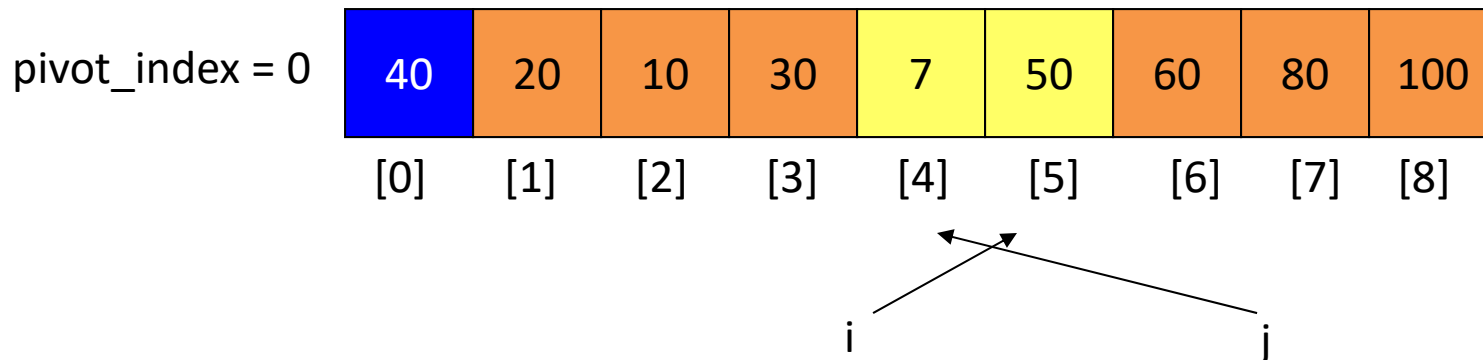
Phần tử chốt là phần tử đứng đầu

1. While $i \leq \text{right}$ and $A[i] < \text{pivot}$
 $++i$
- 2. While $j \geq \text{left}$ and $A[j] > \text{pivot}$
 $--j$
3. If $i < j$
 $\text{swap}(A[i], A[j])$
4. While $j > i$, go to 1.



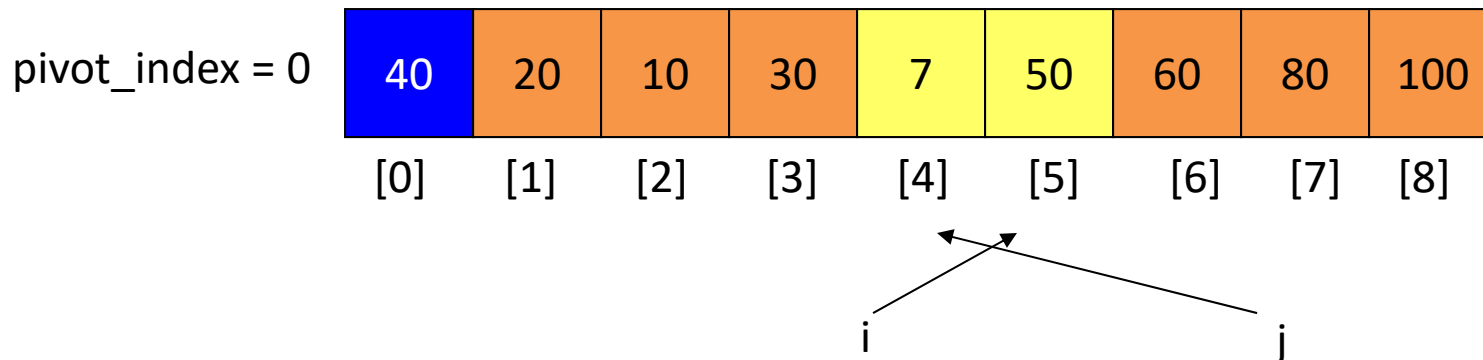
Phần tử chốt là phần tử đứng đầu

1. While $i \leq \text{right}$ and $A[i] < \text{pivot}$
 $++i$
- 2. While $j \geq \text{left}$ and $A[j] > \text{pivot}$
 $--j$
3. If $i < j$
 $\text{swap}(A[i], A[j])$
4. While $j > i$, go to 1.



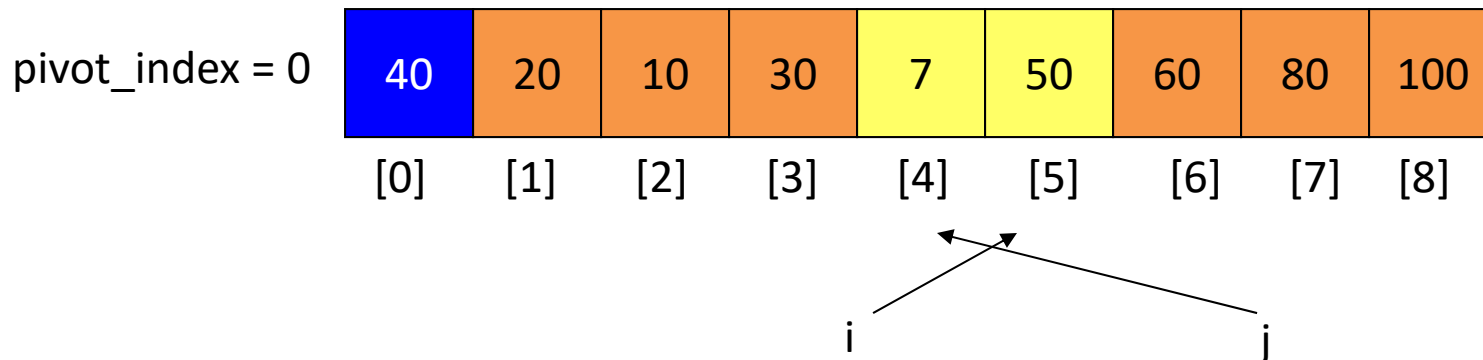
Phần tử chốt là phần tử đứng đầu

1. While $i \leq \text{right}$ and $A[i] < \text{pivot}$
 $++i$
2. While $j \geq \text{left}$ and $A[j] > \text{pivot}$
 $--j$
- 3. If $i < j$
 $\text{swap}(A[i], A[j])$
4. While $j > i$, go to 1.



Phần tử chốt là phần tử đứng đầu dãy: 2-way partitioning

1. While $i \leq \text{right}$ and $A[i] < \text{pivot}$
 $++i$
2. While $j \geq \text{left}$ and $A[j] > \text{pivot}$
 $--j$
3. If $i < j$
 $\text{swap}(A[i], A[j])$
- 4. While $j > i$, go to 1.

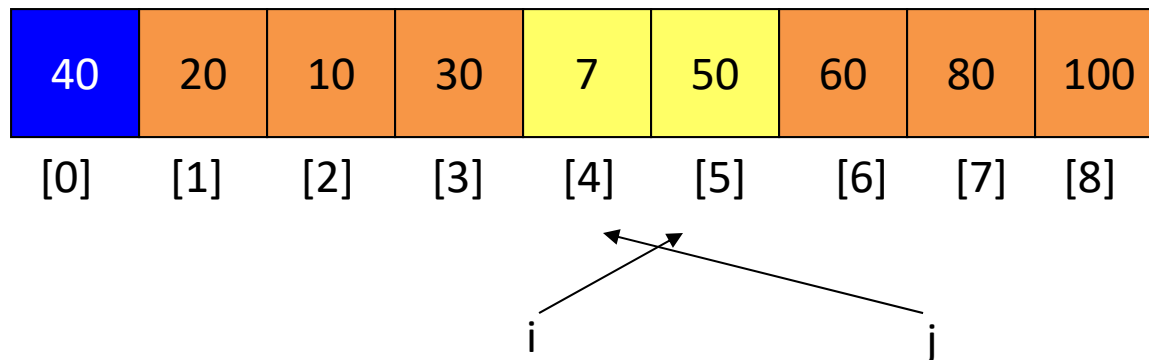


Phần tử chốt là phần tử đứng đầu dãy: 2-way partitioning

1. While $i \leq \text{right}$ and $A[i] < \text{pivot}$
 $++i$
2. While $j \geq \text{left}$ and $A[j] > \text{pivot}$
 $--j$
3. If $i < j$
 $\text{swap}(A[i], A[j])$
4. While $j > i$, go to 1.

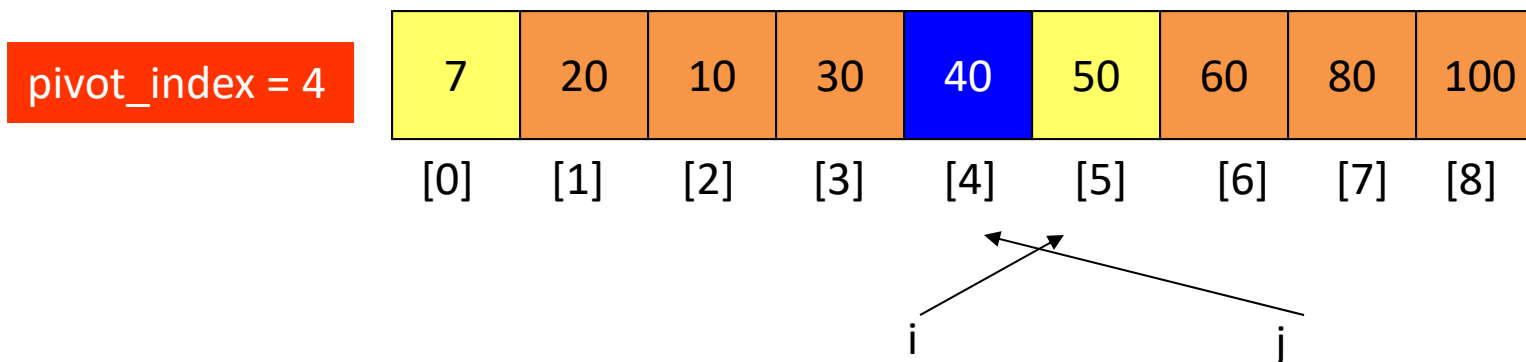


pivot_index = 0

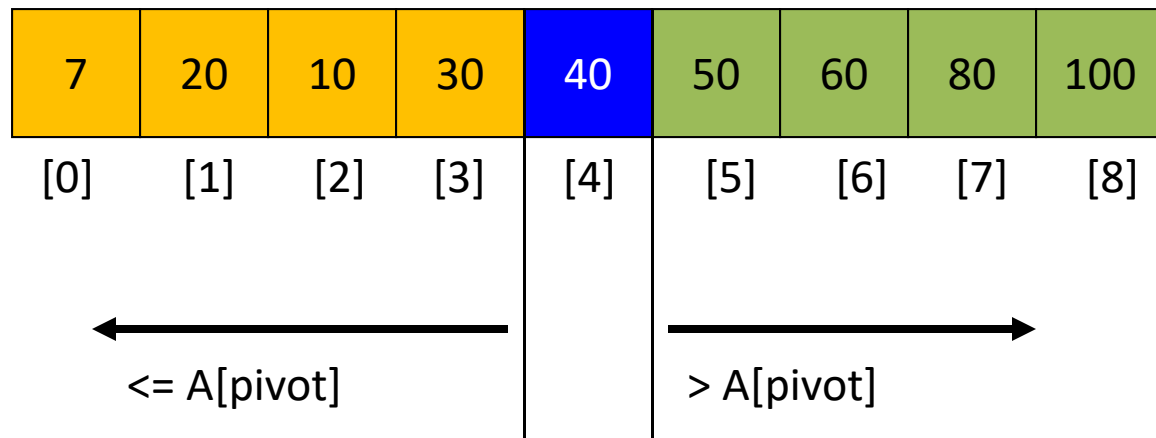


Phần tử chốt là phần tử đứng đầu dãy: 2-way partitioning

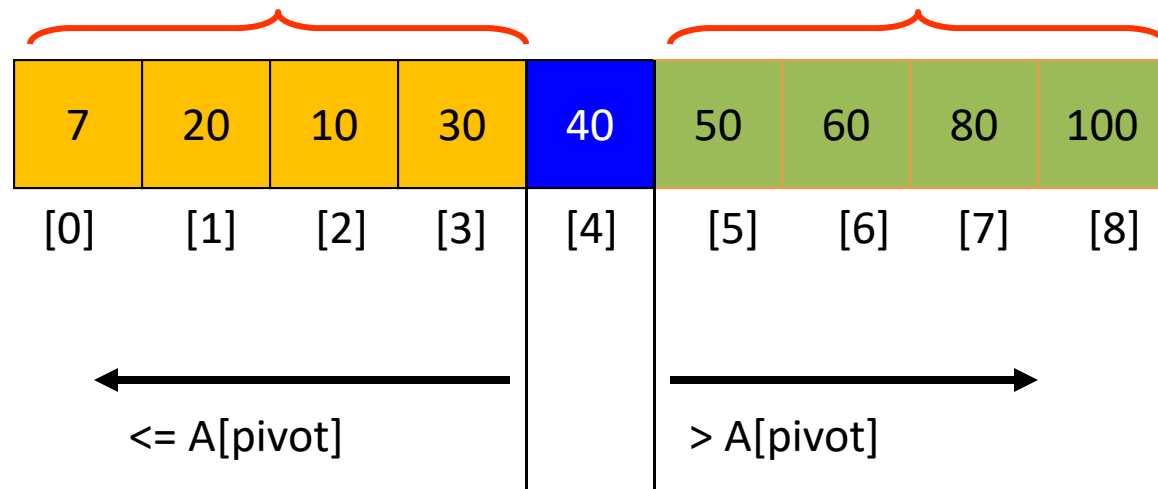
1. While $i \leq \text{right}$ and $A[i] < \text{pivot}$
 $++i$
2. While $j \geq \text{left}$ and $A[j] > \text{pivot}$
 $--j$
3. If $i < j$
 $\text{swap}(A[i], A[j])$
4. While $j > i$, go to 1.
- 5. $\text{Swap}(A[j], A[\text{pivot_index}])$



Dãy thu được sau khi gọi Partition(A,0,8)



Recursion: Quicksort Sub-arrays



Source code 2-way partitioning QuickSort:

Phần tử chốt là phần tử đứng đầu dãy

```
//QuickSort: Chọn phần tử đầu làm phần tử chốt:
int Partition(int A[], int left, int right)
{
    int i = left; int j = right+1;
    int pivot = A[left];
    while (true)
    {
        //Tìm tu trái sang phần tử đầu tiên >=pivot:
        i = i + 1;
        while (i <= right && A[i] < pivot) i=i+1;
        //Tìm tu phải sang phần tử đầu tiên <=pivot:
        j -=1;
        while (j >= left && pivot < A[j]) j =j-1;
        if (i >= j) break;
        swap(&A[i], &A[j]);
    }
    swap(&A[j], &A[left]);
    return j;
}
```

```
void swap(int *a, int *b)
{
    int temp =*a;
    *a=*b;
    *b=temp;
}
```

```
//QuickSort: Chọn phần tử đầu làm phần tử chốt:
int Partition(int A[], int left, int right)
{
    int i = left; int j = right+1;
    int pivot = A[left];
    while (true)
    {
        //Tìm tu trái sang phần tử đầu tiên >pivot:
        while (A[++i] < pivot)
            if (i==right) break;
        //Tìm tu phải sang trái phần tử đầu tiên < pivot
        while (pivot < A[--j]);
        // if (j==left) break; //không cần thiết vì phần tử chốt a[left] acts as sentinel
        if (i >= j) break;
        swap(&A[i], &A[j]);
    }
    swap(&A[j], &A[left]);
    return j;
}
```

```
void QuickSort(int A[], int Left, int Right)
{
    int index_Pivot;
    if (Left < Right )
    {
        index_Pivot =Partition(A,Left,Right);
        QuickSort(A,Left,index_Pivot-1);
        QuickSort(A, index_Pivot +1, Right);
    }
}
```

Source code 2-way partitioning QuickSort: Phần tử chốt là phần tử đứng giữa dãy

```
int PartitionMid(int A[], int left, int right)
{
    int pivot = A[(left + right)/2];
    while (left < right){
        //Tim tu trai sang phai phan tu dau tien >= pivot:
        while (A[left] < pivot) left++;
        //Tim tu phai sang trai phan tu dau tien <= pivot:
        while (A[right] > pivot) right--;
        if (left < right)
        {
            //doi cho 2 phan tu do cho nhau:
            swap(&A[left], &A[right]);
            left++; right--;
        }
    }
    return right;
}
```

```
void QuickSort(int A[], int Left, int Right)
{
    int index_Pivot;
    if (Left < Right )
    {
        index_Pivot = PartitionMid(A, Left, Right);
        QuickSort(A, Left, index_Pivot-1);
        QuickSort(A, index_Pivot +1, Right);
    }
}
```

Source code 2-way partitioning QuickSort: Phần tử chốt là phần tử đứng cuối dãy

```
int PartitionLast(int A[], int left, int right) {
    int pivot = A[right];
    int j = left - 1;
    for (int i = left; i < right; i++) {
        if (pivot >= A[i])
        {
            j = j + 1;
            swap(&A[i], &A[j]);
        }
    }
    A[right] = A[j + 1];
    A[j + 1] = pivot;
    return (j + 1);
}
```

```
void QuickSort(int A[], int Left, int Right)
{
    int index_Pivot;
    if (Left < Right )
    {
        index_Pivot = PartitionLast(A, Left, Right);
        QuickSort(A, Left, index_Pivot - 1);
        QuickSort(A, index_Pivot + 1, Right);
    }
}
```

4. Sắp xếp nhanh (Quick sort)

- Thuật toán sắp xếp nhanh được phát triển bởi Hoare năm 1960 khi ông đang làm việc cho hãng máy tính nhỏ Elliott Brothers ở Anh, được ông dùng để dịch tiếng Nga sang tiếng Anh.
- QS có thời gian tính trung bình là $O(n \log n)$, tuy nhiên thời gian tính tồi nhất của nó lại là $O(n^2)$.
- QS là thuật toán sắp xếp tại chỗ, nhưng nó không có tính ổn định.
- QS khá đơn giản về lý thuyết, nhưng lại không dễ cài đặt.

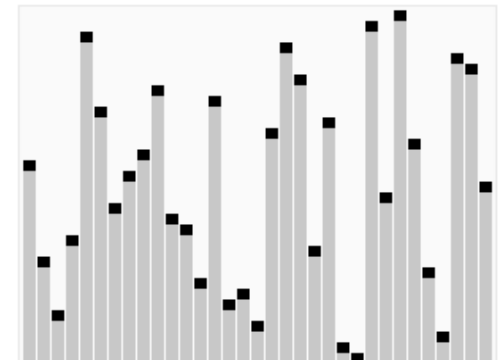


C.A.R. Hoare

January 11, 1934

ACM Turing Award, 1980

Photo: 2006



đường nằm ngang cho biết pivot

4. Sắp xếp nhanh (Quick sort)

- Worst case:
 - Số phép so sánh cần thực hiện $\sim n^2/2$
- Average Case: số phép so sánh cần thực hiện $\sim 1.39n \log n$
 - Số phép so sánh cần thực hiện nhiều hơn $\sim 39\%$ so với sắp xếp trộn trong trường hợp tồi nhất
 - Nhưng nhanh hơn sắp xếp trộn vì ít phải di chuyển các phần tử

Running time estimates

- Home computer: giả thiết có thể thực hiện 10^8 phép so sánh trong 1 giây
- Super computer: giả thiết máy tính có thể thực hiện 10^{12} phép so sánh trong 1 giây

| | insertion sort (N^2) | | | mergesort ($N \log N$) | | | quicksort ($N \log N$) | | |
|----------|--------------------------|-----------|-----------|--------------------------|----------|---------|--------------------------|---------|---------|
| computer | thousand | million | billion | thousand | million | billion | thousand | million | billion |
| home | instant | 2.8 hours | 317 years | instant | 1 second | 18 min | instant | 0.6 sec | 12 min |
| super | instant | 1 second | 1 week | instant | instant | instant | instant | instant | instant |

4. Sắp xếp nhanh

4.1. 2-way partitioning

4.2. Bentley-McIlroy 3-way partitioning

4.3. Hàm qsort có sẵn trong thư viện stdlib.h

Dãy cần sắp xếp có các khóa lặp lại (duplicate keys)

Mục đích của dạng sắp xếp này là nhóm các bản ghi có cùng khóa.

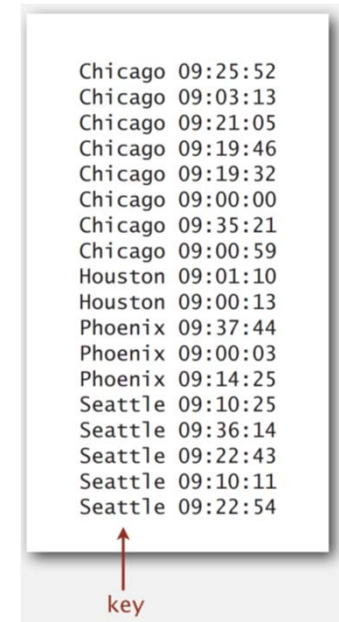
- Sắp dân số theo tuổi
- Tìm các điểm thẳng hàng
- Loại bỏ các thư trùng lặp trong mailing list.
- Sắp xếp các người xin việc theo trường đã theo học.

Đặc điểm của loại ứng dụng này:

- File lớn
- Có số lượng nhỏ các khóa

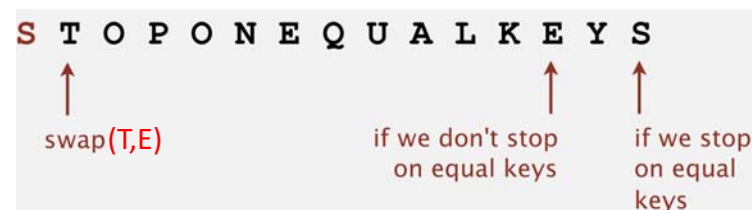
Thuật toán:

- Mergesort với duplicate keys: $\sim O(n \log n)$: yêu cầu $1/(2n \log n)$ đến $n \log n$ phép so sánh
- Quicksort với duplicate keys: $O(n^2)$ trừ khi việc phân chia dừng khi các khóa giống nhau (Điều này được phát hiện vào những năm 1990 khi người dùng sử dụng hàm qsort có sẵn của C)



| | |
|---------|----------|
| Chicago | 09:25:52 |
| Chicago | 09:03:13 |
| Chicago | 09:21:05 |
| Chicago | 09:19:46 |
| Chicago | 09:19:32 |
| Chicago | 09:00:00 |
| Chicago | 09:35:21 |
| Chicago | 09:00:59 |
| Houston | 09:01:10 |
| Houston | 09:00:13 |
| Phoenix | 09:37:44 |
| Phoenix | 09:00:03 |
| Phoenix | 09:14:25 |
| Seattle | 09:10:25 |
| Seattle | 09:36:14 |
| Seattle | 09:22:43 |
| Seattle | 09:10:11 |
| Seattle | 09:22:54 |

↑
key

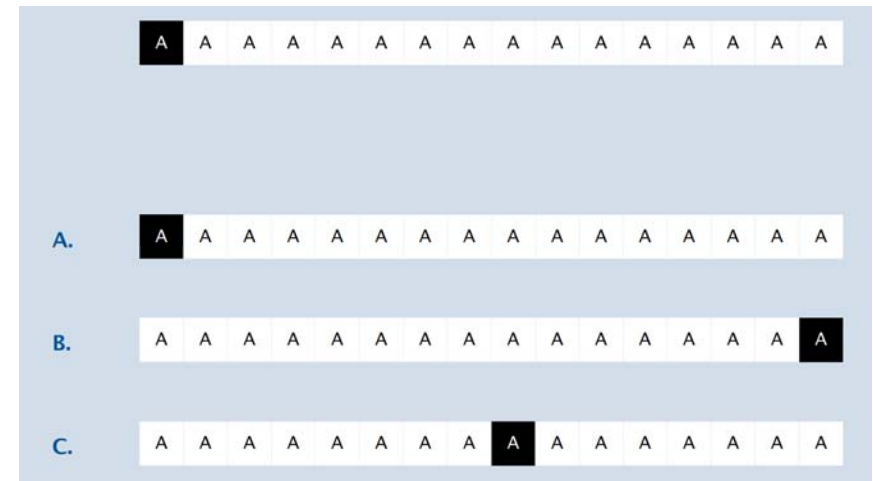


Dãy cần sắp xếp có các khóa lặp lại (duplicate keys)

Quicksort với duplicate keys: $O(n^2)$ trừ khi việc phân chia dừng khi các khóa giống nhau (Điều này được phát hiện vào những năm 1990 khi người dùng sử dụng hàm qsort có sẵn của C)

Ví dụ: Chọn phần tử đầu làm phần tử chốt. Khi thủ tục Partition của QuickSort kết thúc, mảng đã cho là mảng nào ?

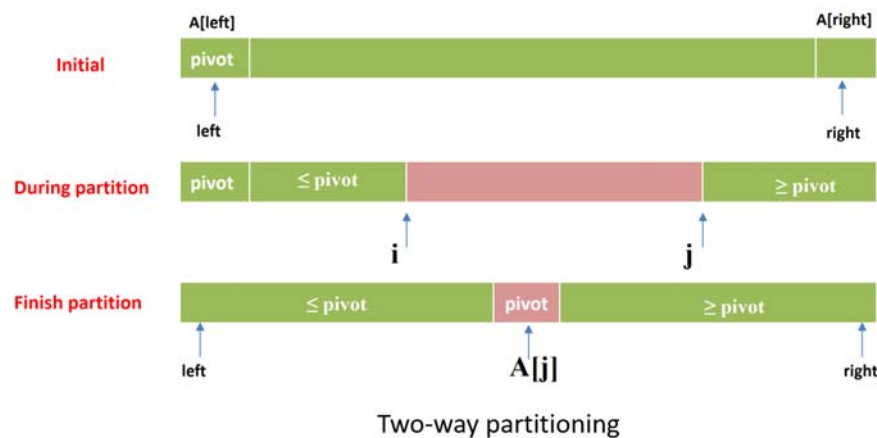
| | | a[] | | | | | | | | | | | | | | | |
|---|----|------|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|
| i | j | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| | | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A |
| 1 | 15 | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A |
| 1 | 15 | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A |
| 2 | 14 | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A |
| 2 | 14 | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A |
| 3 | 13 | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A |
| 3 | 13 | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A |
| 4 | 12 | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A |
| 4 | 12 | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A |
| 5 | 11 | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A |
| 5 | 11 | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A |
| 6 | 10 | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A |
| 6 | 10 | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A |
| 7 | 9 | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A |
| 7 | 9 | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A |
| 8 | 8 | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A |
| 8 | 8 | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A |



Dãy cần sắp xếp có các khóa lặp lại (duplicate keys)

Quicksort với duplicate keys: $O(n^2)$ trừ khi việc phân chia dừng khi các khóa giống nhau (Điều này được phát hiện những năm 1990 khi người dùng sử dụng hàm qsort có sẵn của C):

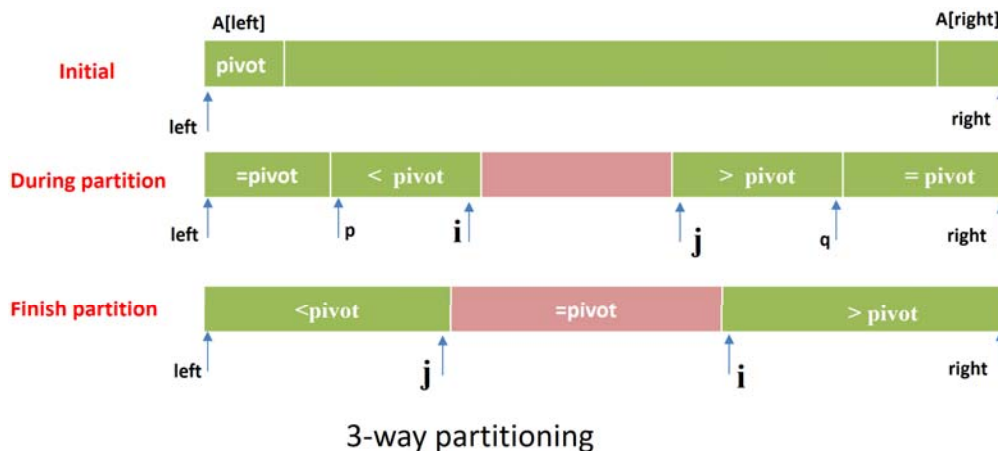
- Sau khi thực hiện partition, Quick Sort tiếp tục được gọi đệ quy ở 2 nửa dãy:



Gọi đệ quy:

- $QS2way(left, j-1)$
- $QS2way(j+1, right)$

➔ Hai tác giả Jon L. Bentley và M. Douglas McIlroy (1993) cải tiến:



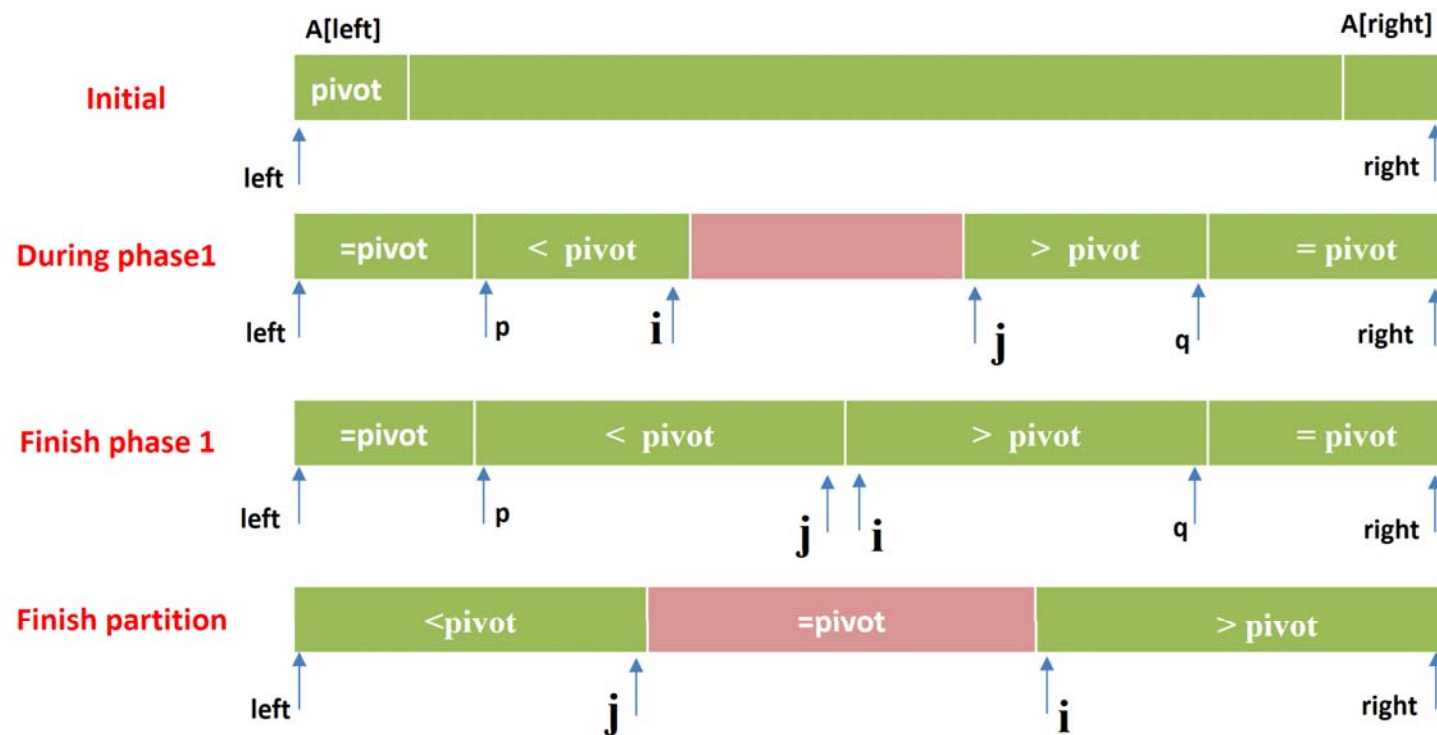
Gọi đệ quy:

- $QS3way(left, j)$
- $QS3way(i, right)$

Bentley-McIlroy 3-way partitioning: phần tử chốt là phần tử đầu dãy

Phân chia mảng thành 3 phần:

- Các phần tử từ $A[\text{left}] \dots A[j]$ đều nhỏ hơn phần tử chốt pivot
- Các phần tử từ $A[i] \dots A[\text{right}]$ đều lớn hơn phần tử chốt pivot
- Các phần tử từ $A[j+1] \dots A[i-1]$ đều bằng phần tử chốt pivot



Bentley-McIlroy 3-way partitioning

Bentley-McIlroy 3-way partitioning: phần tử chốt là phần tử đầu dãy

Phase 1:

Pivot = A[left];

$p = \text{left} + 1; q = \text{right};$

$i = p; j = q;$

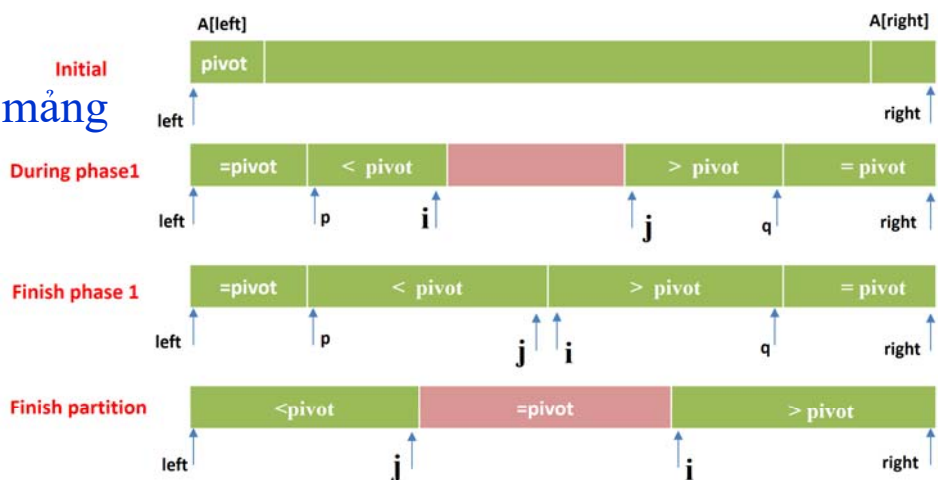
Repeat until i and j pointers cross ($j < i$)

- Duyệt i từ trái sang phải chừng nào $A[i] < \text{pivot}$ //Tìm phần tử đầu tiên từ trái sang $\geq \text{pivot}$
- Duyệt j từ phải sang trái chừng nào $A[j] > \text{pivot}$ //Tìm phần tử đầu tiên từ phải sang $\leq \text{pivot}$
- If i and j pointers cross ($j < i$): break;
else

- Swap($A[i], A[j]$)
- If ($A[i] == \text{pivot}$) {swap($A[i], A[p]$); $p++$;}
- If ($A[j] == \text{pivot}$) {swap($A[j], A[q]$); $q--$;}

Phase 2: Chuyển các phần tử bằng nhau về giữa mảng

- Duyệt j và p từ phải sang trái:
 - swap ($A[j], A[p]$)
- Duyệt i và q từ trái sang phải:
 - swap ($A[i], A[q]$)



Bentley-McIlroy 3-way partitioning

Ví dụ: Bentley-McIlroy 3-way partitioning: phần tử chốt là phần tử đầu dãy

Phase 1:

Pivot = A[left];

$p = \text{left} + 1$; $q = \text{right}$;

$i = p$; $j = q$;

Repeat until i and j pointers cross ($j < i$)

- Duyệt i từ trái sang phải chừng nào $A[i] < \text{pivot}$ //Tìm phần tử đầu tiên từ trái sang $\geq \text{pivot}$
- Duyệt j từ phải sang trái chừng nào $A[j] > \text{pivot}$ //Tìm phần tử đầu tiên từ phải sang $\leq \text{pivot}$
- If i and j pointers cross ($j < i$): break;
- else
 - Swap($A[i]$, $A[j]$)
 - If ($A[i] == \text{pivot}$) {swap($A[i]$, $A[p]$); $p++$;}
 - If ($A[j] == \text{pivot}$) {swap($A[j]$, $A[q]$); $q--$;}



Duyệt i từ trái sang phải chừng nào $A[i] < \text{pivot}$
//Tìm phần tử đầu tiên từ trái sang $\geq \text{pivot}$

Duyệt j từ phải sang trái chừng nào $A[j] > \text{pivot}$
//Tìm phần tử đầu tiên từ phải sang $\leq \text{pivot}$

Duyệt j từ phải sang trái chừng nào $A[j] > \text{pivot}$
//Tìm phần tử đầu tiên từ phải sang $\leq \text{pivot}$

Swap($A[i]$, $A[j]$): đổi chỗ X và C

Ví dụ: Bentley-McIlroy 3-way partitioning: phần tử chốt là phần tử đầu dãy



Phase 1:

Pivot = A[left];

p = left+1; q = right;

i = p; j = q;

Repeat until i and j pointers cross ($j < i$)

- Duyệt i từ trái sang phải chừng nào $A[i] < \text{pivot}$ //Tìm phần tử đầu tiên từ trái sang $\geq \text{pivot}$
- Duyệt j từ phải sang trái chừng nào $A[j] > \text{pivot}$ //Tìm phần tử đầu tiên từ phải sang $\leq \text{pivot}$
- If i and j pointers cross ($j < i$): break;
- else
 - Swap(A[i], A[j])
 - If (A[i] == pivot) {swap(A[i], A[p]); p++;}
 - If (A[j] == pivot) {swap(A[j], A[q]); q--;}

Duyệt i từ trái sang phải chừng nào $A[i] < \text{pivot}$
//Tìm phần tử đầu tiên từ trái sang $\geq \text{pivot}$

Duyệt j từ phải sang trái chừng nào $A[j] > \text{pivot}$
//Tìm phần tử đầu tiên từ phải sang $\leq \text{pivot}$

Swap(A[i], A[j]): đổi chỗ W và P

$A[i] = \text{pivot} \rightarrow \text{Swap}(A[i], A[p]):$ đổi chỗ P và A
p++;

Duyệt i từ trái sang phải chừng nào $A[i] < \text{pivot}$
//Tìm phần tử đầu tiên từ trái sang $\geq \text{pivot}$

Ví dụ: Bentley-McIlroy 3-way partitioning: phần tử chốt là phần tử đầu dãy

Phase 1:

Pivot = A[left];

$p = \text{left} + 1; q = \text{right};$

$i = p; j = q;$

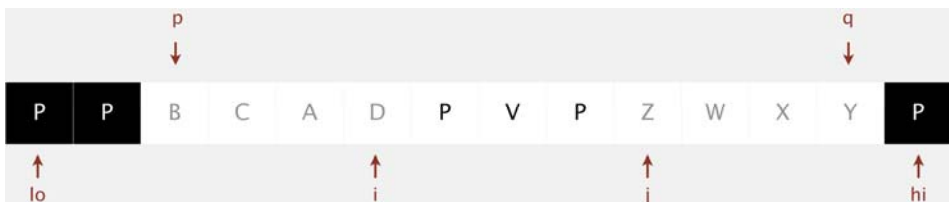
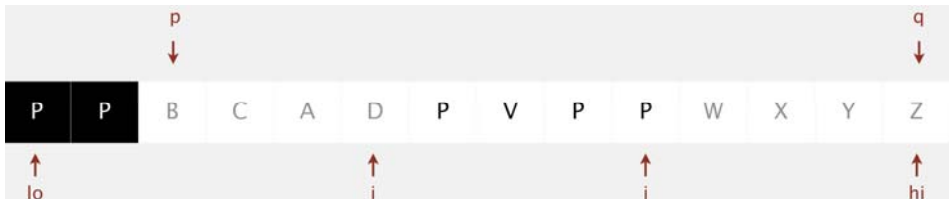
Repeat until i and j pointers cross ($j < i$)

- Duyệt i từ trái sang phải chừng nào $A[i] < \text{pivot}$ //Tìm phần tử đầu tiên từ trái sang $\geq \text{pivot}$
- Duyệt j từ phải sang trái chừng nào $A[j] > \text{pivot}$ //Tìm phần tử đầu tiên từ phải sang $\leq \text{pivot}$
- If i and j pointers cross ($j < i$): break;
- else
 - Swap($A[i], A[j]$)
 - If ($A[i] == \text{pivot}$) {swap($A[i], A[p]$); $p++$;}
 - If ($A[j] == \text{pivot}$) {swap($A[j], A[q]$); $q--$;}

Duyệt j từ phải sang trái chừng nào $A[j] > \text{pivot}$
//Tìm phần tử đầu tiên từ phải sang $\leq \text{pivot}$

Swap($A[i], A[j]$): đổi chỗ P và D

$A[j] = \text{pivot} \rightarrow \text{Swap}(A[j], A[q])$: đổi chỗ P và Z
 $q--$;



Ví dụ: Bentley-McIlroy 3-way partitioning: phần tử chốt là phần tử đầu dãy

Phase 1:

Pivot = A[left];

$p = \text{left} + 1; q = \text{right};$

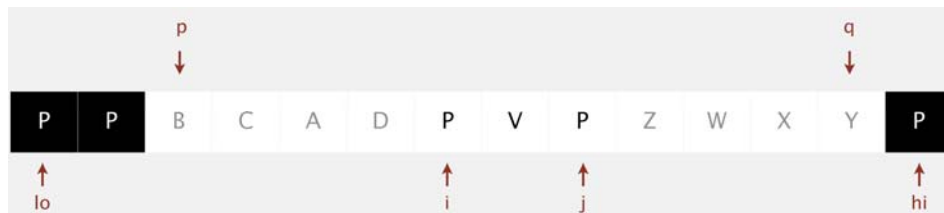
$i = p; j = q;$

Repeat until i and j pointers cross ($j < i$)

- Duyệt i từ trái sang phải chừng nào $A[i] < \text{pivot}$ //Tìm phần tử đầu tiên từ trái sang $\geq \text{pivot}$
- Duyệt j từ phải sang trái chừng nào $A[j] > \text{pivot}$ //Tìm phần tử đầu tiên từ phải sang $\leq \text{pivot}$
- If i and j pointers cross ($j < i$): break;
- else
 - Swap($A[i], A[j]$)
 - If ($A[i] == \text{pivot}$) {swap($A[j], A[p]$); $p++$;}
 - If ($A[j] == \text{pivot}$) {swap($A[j], A[q]$); $q--$;}

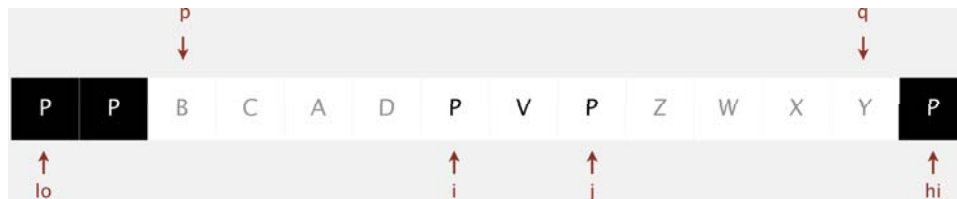


Duyệt i từ trái sang phải chừng nào $A[i] < \text{pivot}$
//Tìm phần tử đầu tiên từ trái sang $\geq \text{pivot}$

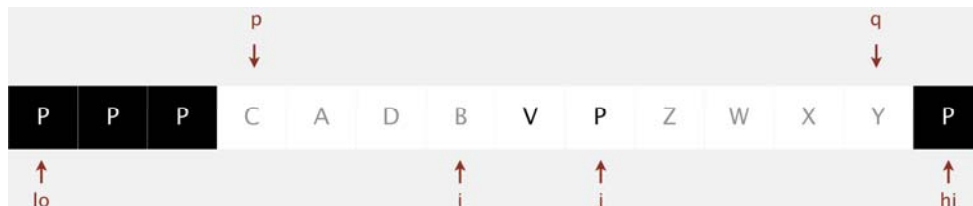


Duyệt j từ phải sang trái chừng nào $A[j] > \text{pivot}$
//Tìm phần tử đầu tiên từ phải sang $\leq \text{pivot}$

Swap($A[i], A[j]$)



$A[i] = \text{pivot} \rightarrow \text{Swap}(A[i], A[p]):$ đổi chỗ P với B
 $p++$;



$A[j] = \text{pivot} \rightarrow \text{Swap}(A[j], A[q]):$ đổi chỗ P với Y
 $q--$;

Ví dụ: Bentley-McIlroy 3-way partitioning: phần tử chốt là phần tử đầu dãy

Phase 1:

Pivot = A[left];

$p = \text{left} + 1$; $q = \text{right}$;

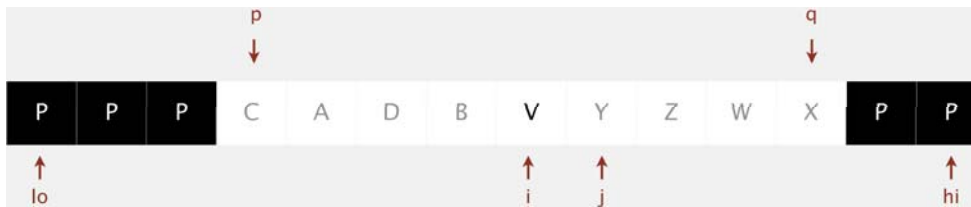
$i = p$; $j = q$;

Repeat until i and j pointers cross ($j < i$)

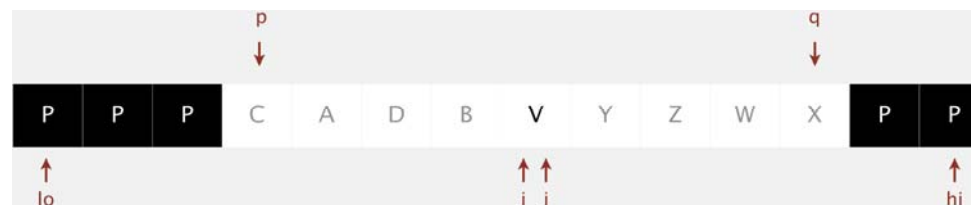
- Duyệt i từ trái sang phải chừng nào $A[i] < \text{pivot}$ //Tìm phần tử đầu tiên từ trái sang $\geq \text{pivot}$
- Duyệt j từ phải sang trái chừng nào $A[j] > \text{pivot}$ //Tìm phần tử đầu tiên từ phải sang $\leq \text{pivot}$
- If i and j pointers cross ($j < i$): break;
- else
 - Swap($A[i]$, $A[j]$)
 - If ($A[i] == \text{pivot}$) {swap($A[i]$, $A[p]$); $p++$;}
 - If ($A[j] == \text{pivot}$) {swap($A[j]$, $A[q]$); $q--$;}



Duyệt i từ trái sang phải chừng nào $A[i] < \text{pivot}$
//Tìm phần tử đầu tiên từ trái sang $\geq \text{pivot}$



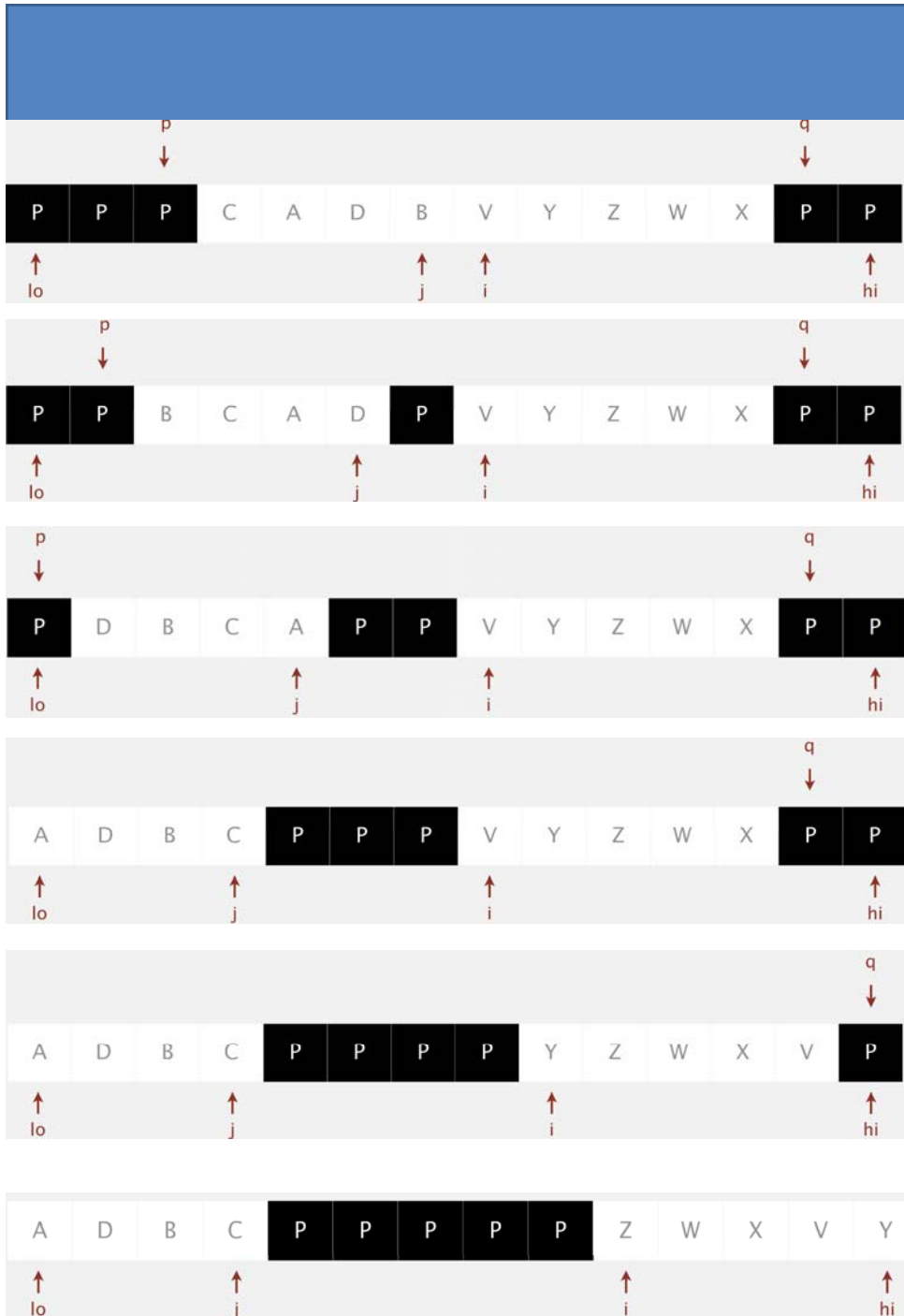
Duyệt j từ phải sang trái chừng nào $A[j] > \text{pivot}$
//Tìm phần tử đầu tiên từ phải sang $\leq \text{pivot}$



Duyệt j từ phải sang trái chừng nào $A[j] > \text{pivot}$
//Tìm phần tử đầu tiên từ phải sang $\leq \text{pivot}$



$j < i$: pointer cross
➔ stop Phase 1



Phase 2: Chuyển các phần tử bằng nhau về giữa mảng

- Duyệt j và p từ phải sang trái:
 - swap ($A[j], A[p]$)
- Duyệt i và q từ trái sang phải:
 - swap ($A[i], A[q]$)

Swap($A[j], A[p]$): đổi chỗ B và P

Swap($A[j], A[p]$): đổi chỗ D và P

Swap($A[j], A[p]$): đổi chỗ A và P

Swap($A[i], A[q]$): đổi chỗ V và P

Swap($A[i], A[q]$): đổi chỗ Y và P

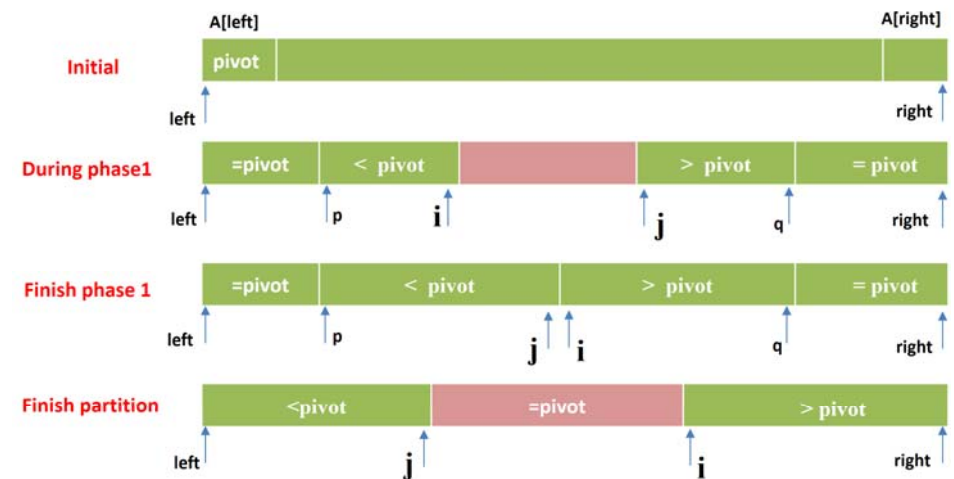
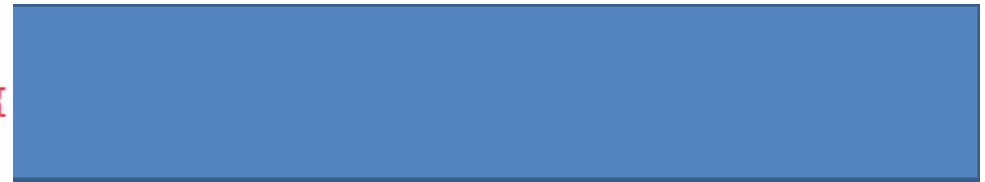
Finish phase 2

➔ Terminate Partition

```

/*Bentley-McIlroy 3-way partitioning quick sort
Phan tu chot pivot la phan tu dau day */
void QuickSort3way_Left(int A[], int left, int right) {
    if (right <= left) return;
    int pivot = A[left];
    int i = left, j = right+1;
    int p = left+1, q = right;
    //Phase 1:
    while (1) {
        //tim phan tu dau tien tu trai sang >=pivot:
        while (A[++i] < pivot)
            if (i==right) break;
        //tim phan tu dau tien tu phai sang <=pivot:
        while (pivot < A[--j]);
        if (i >= j) break;
        swap(&A[i],&A[j]);
        if (A[i]==pivot) {
            swap(&A[p],&A[i]);
            p++;
        }
        if (A[j]==pivot){
            swap(&A[q],&A[j]);
            q--;
        }
    }
}

```



Bentley-McIlroy 3-way partitioning

```

//Phase 2:
/* 2.1 Hoan doi cac phan tu ben trai day A[left]..A[p-1]
co gia tri bang phan tu chot voi cac phan tu bat dau tu A[j] tro ve dau day..*/
for (int k = left; k < p; k++,j--) swap(&A[k],&A[j]);
/*2.2 Hoan doi cac phan tu ben phai day A[q+1]...A[right]
co gia tri bang phan tu chot voi cac phan tu bat dau tu A[i] tro ve cuoi day..*/
for (int k = q+1; k <=right; k++,i++) swap(&A[k],&A[i]);

//Goi de quy day con trai:
QuickSort3way_Left(A, left, j);
//Goi de quy day con phai:
QuickSort3way_Left(A, i, right);
}

```

```
int d[14] = {16,1,2,24,23,16,16,22,16,4,16,3,25,26};
printf("Goi quicksort3way, chon phan tu dau day la phan tu chot\n");
QuickSort3way_Left(d,0,13);

printf("Result of quick sort3way: \n");
for (int i = 0; i<14; i++)
    printf("%d \n",d[i]);
```

Bài tập 1

- Viết hàm:

```
void QuickSort3way_Right(int A[ ], int left, int right)
```

Thực hiện thuật toán Bentley-McIlroy 3-way partitioning với phần tử chốt là phần tử cuối dãy

```

/*Bentley-McIlroy 3-way partitioning quick sort
Phan tu chot pivot la phan tu cuoi day */
void QuickSort3way_Right(int A[], int left, int right) {
/* This function partitions a[] in three parts
a) a[l..i] contains all elements smaller than pivot
b) a[i+1..j-1] contains all occurrences of pivot
c) a[j..r] contains all elements greater than pivot */
    if (right <= left) return;
    int pivot = A[right];
    int i = left-1; int j = right;
    int p = left-1, q = right;

//Phase 1:
while (1)
{
    // From left, find the first element greater than
    // or equal to v. This loop will definitely terminate
    // as v is last element:
    while (A[++i] < A[right]);

    // From right, find the first element smaller than or
    // equal to v:
    while (A[right] < A[--j])
        if (j == left) break;

    // If i and j cross, then we are done:
    if (i >= j) break;

    // Swap, so that smaller goes on left greater goes on right
    swap(&A[i], &A[j]);
}
}

```



```

// Move all same left occurrence of pivot to beginning of
// array and keep count using p:
if (A[i] == A[right])
{
    p++;
    swap(&A[p], &A[i]);
}

// Move all same right occurrence of pivot to end of array
// and keep count using q:
if (A[j] == A[right])
{
    q--;
    swap(&A[j], &A[q]);
}
}

//Phase 2:
// Move all left same occurrences from beginning
// to adjacent to arr[i]
j = i-1;
for (int k = left; k <= p; k++) {
    swap(&A[k], &A[j]);
    j--;
}

// Move all right same occurrences from end
// to adjacent to arr[i]
for (int k = right; k >= q; k--){
    swap(&A[i], &A[k]);
    i++;
}

//Goi de quy nua trai va nua phai:
QuickSort3way_Right(A, left, j);
QuickSort3way_Right(A, i, right);
}

```

Bài tập 2

- Viết 2 hàm sắp xếp
 - 2-way partitioning:
`void QuickSort2way(int a[], int left, int right);`
 - 3-way partitioning:
`void QuickSort3way(int a[], int left, int right);`
- Tạo 1 mảng gồm 10 triệu số nguyên ngẫu nhiên từ 1-10
 - Viết hàm tạo mảng dữ liệu lưu trữ trong bộ nhớ động, kích thước mảng được truyền qua tham số
`int * createArray(int size);`
- Hiển thị thời gian sắp xếp của mỗi giải thuật khi chạy trên cùng mảng vừa tạo
 - Chạy trên cùng 1 mảng → cần hàm cho phép sao chép một mảng số nguyên đã có:

`int * dumpArray(int *p, int size);`

Bài tập 2

```
#define SMALL_NUMBER 20
#define HUGE_NUMBER 10000000
main() {
    int *a1, *a2;
    a1 = createArray(SMALL_NUMBER);
    a2 = dumpArray(a1, SMALL_NUMBER);
    QuickSort2way(a1, 0, SMALL_NUMBER-1);
    // kiểm tra mảng a1 có được sắp xếp theo đúng thứ tự tăng dần ko:
    .....
    QuickSort3way(a2, 0, SMALL_NUMBER-1);
    // kiểm tra mảng a1 có được sắp xếp theo đúng thứ tự tăng dần ko:
    .....
    free (a1);
    free (a2);
    a1 = createArray(HUGE_NUMBER);
    a2 = dumpArray(a1, HUGE_NUMBER);
    // compare the time to execute sorting:
    .....
}
```

Trong hàm main():

- đầu tiên kiểm tra việc sắp xếp đúng với số lượng nhỏ phần tử.
- Sau đó tiến hành so sánh thời gian với số lượng lớn các phần tử.

```
Mang gom 1 trieu phan tu:
Running time of 2-way Quicksort: 0.079000 seconds
Running time of 3-way Quicksort: 0.031000 seconds
```

```
Mang gom 10 trieu phan tu:
Running time of 2-way Quicksort: 0.841000 seconds
Running time of 3-way Quicksort: 0.283000 seconds
```

```
Mang gom 100 trieu phan tu:
Running time of 2-way Quicksort: 9.361000 seconds
Running time of 3-way Quicksort: 2.702000 seconds
```

```
Mang gom 1 ti phan tu:
Running time of 2-way Quicksort: 116.542000 seconds
Running time of 3-way Quicksort: 34.638000 seconds
```

Gợi ý (1): sinh số ngẫu nhiên

- Gọi hàm `rand()` để khởi tạo số ngẫu nhiên trong phạm vi từ 0 tới `RAND_MAX` với `RAND_MAX` là hằng số có giá trị mặc định khác nhau giữa các lần chạy nhưng luôn có giá trị ít nhất là 32767

- `#include <stdlib.h>`
- `i = rand();`

```
#include <stdio.h>
#include <stdlib.h>

int main () {
    int i, n;
    n = 5;
    /* Print 5 random numbers from 0 to 10 */
    for( i = 0 ; i < n ; i++ )
        printf("%d\n", rand() % 10);
    return(0);
}
```

Các lần chạy đều sinh ra các số ngẫu nhiên như nhau

Gợi ý (1): sinh số ngẫu nhiên

- Gọi hàm `rand()` để khởi tạo số ngẫu nhiên trong phạm vi từ 0 tới `RAND_MAX` với `RAND_MAX` là hằng số có giá trị mặc định khác nhau giữa các lần chạy nhưng luôn có giá trị ít nhất là 32767

- `#include <stdlib.h>`
- `i = rand();`

```
n = 5;
/* Print 5 random numbers from 1 to 10 */
for( i = 0 ; i < n ; i++ )
    printf("%d\n", 1 + rand() % 9);
```

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
```

```
int main () {
    int i, n;
    time_t t;
```

```
/* Intializes random number generator */
//srand((unsigned) time(&t));
srand(time(NULL));
```

```
n = 5;
/* Print 5 random numbers from 0 to 10 */
for( i = 0 ; i < n ; i++ )
    printf("%d\n", rand() % 10);
return(0);
}
```

Gợi ý (1): sinh số ngẫu nhiên

- Sinh số ngẫu nhiên từ có giá trị nằm trong khoảng [Min,Max]:

```
int number =Min + rand() % (Max-Min) ;
```

```
//tao mang du lieu kieu int, luu tru bo nho dong
int * createArray(int size)
{
    int i, *p;
    /* Cap phát bo nho cho mang so nguyên gom size so */
    p = (int *)malloc(size * sizeof(int));
    if (p == NULL)
    {
        printf("Cap phát bo nho không thành công!\n");
        return NULL;
    }
    srand(time(NULL));
    for (i=0;i<size;i++) p[i] = 1 + rand()%9;
    return p;
}
```

//Hàm cho phép sao chép một mảng số nguyên đã có

```
int *dumpArray(int *p, int size)
{
    int i, *q;
    q = (int *)malloc(size * sizeof(int));
    if (q == NULL)
    {
        printf("Cap phát bo nho không thành công!\n");
        return NULL;
    }
    for (i=0;i<size;i++) q[i] = p[i];
    return q;
}
```

```
/*Neu mang a duoc sap xep theo thu tu tang dan, ham tra ve gia tri 1;  
Neu khong ham tra ve gia tri 0*/
```

```
int checkAscending(int *a, int l, int r){  
    for (int i=l;i<r;i++)  
        if (a[i]>a[i+1]) return 0;  
    return 1;  
}
```

```
#define SMALL_NUMBER 100  
#define HUGE_NUMBER 1000000000  
#include <stdio.h>  
#include <stdlib.h>  
#include <time.h>  
int main() {  
    int *a1, *a2;  
    a1 = createArray(SMALL_NUMBER);  
    a2 = dumpArray(a1, SMALL_NUMBER);  
  
    QuickSort2way(a1, 0, SMALL_NUMBER-1);  
    int kq=checkAscending(a1,0, SMALL_NUMBER-1);  
    if (kq==0) printf("2-way QuickSort chay sai \n");  
    else printf("2-way QuickSort OK \n");  
  
    QuickSort3way(a2, 0, SMALL_NUMBER-1);  
    kq=checkAscending(a2,0, SMALL_NUMBER-1);  
    if (kq==0) printf("3-way QuickSort chay sai \n");  
    else printf("3-way QuickSort OK \n");  
  
    free(a1);free(a2);  
    return 0;  
}
```

Trong hàm main():

- đầu tiên kiểm tra việc sắp xếp đúng với số lượng nhỏ phần tử.

Gợi ý (2): kiểm tra thời gian chạy (Cách 1)

```
#include <time.h>
#include <stdio.h>
time_t start, end;

start = time(NULL);

/* put your algorithm here */

end = time(NULL);

printf("Run in %f seconds.\n", difftime(end, start));
```


Gợi ý (2): kiểm tra thời gian chạy (Cách 2)

```
#include <time.h>
#include <stdio.h>
time_t start,end;

start = clock( );

/* put your algorithm here */

end = clock( );

double runtime = ((double) (end - start)) / CLOCKS_PER_SEC;
printf("Run in %f seconds.\n", runtime);
```

```

#define SMALL_NUMBER 100
#define HUGE_NUMBER 1000000000
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main() {
    int *a1, *a2;
    a1 = createArray(SMALL_NUMBER);
    a2 = dumpArray(a1, SMALL_NUMBER);

    QuickSort2way(a1, 0, SMALL_NUMBER-1);
    int kq=checkAscending(a1,0, SMALL_NUMBER-1);
    if (kq==0) printf("2-way QuickSort chay sai \n");
    else printf("2-way QuickSort OK \n");

    QuickSort3way(a2, 0, SMALL_NUMBER-1);
    kq=checkAscending(a2,0, SMALL_NUMBER-1);
    if (kq==0) printf("3-way QuickSort chay sai \n");
    else printf("3-way QuickSort OK \n");

    free(a1);free(a2);

    a1 = createArray(HUGE_NUMBER);
    a2 = dumpArray(a1, HUGE_NUMBER);
    // compare the time to execute sorting:
    time_t start, end1, end2;
    start = clock();
    QuickSort2way(a1, 0, HUGE_NUMBER-1);
    end1 = clock();
    QuickSort3way(a2, 0, HUGE_NUMBER-1);
    end2 = clock();
    printf("Mang gom %d phan tu: \n",HUGE_NUMBER);
    printf("Running time of 2-way Quicksort: %f seconds \n",((double) (end1 - start)) / CLOCKS_PER_SEC);
    printf("Running time of 3-way Quicksort: %f seconds \n",((double) (end2 - end1)) / CLOCKS_PER_SEC);
    free(a1);free(a2);
    return 0;
}

```

Trong hàm main():

- đầu tiên kiểm tra việc sắp xếp đúng với số lượng nhỏ phần tử.
- Sau đó tiến hành so sánh thời gian với số lượng lớn các phần tử.

Bài tập 2 (tiếp)

- Tạo mảng a ngẫu nhiên gồm 1.000.000 phần tử có giá trị khoảng 0-10
- Sao chép mảng a sang mảng b, c, d, e
- Tính thời gian sắp xếp mảng theo thứ tự tăng dần của các giải thuật
 - 3way partition quicksort trên mảng a
 - 2way partition quick sort trên mảng b
 - Selection sort trên mảng c
 - Insertion sort trên mảng d
 - Merge sort trên mảng e
- Hiển thị 5 thời gian này ra màn hình

4. Sắp xếp nhanh

4.1. 2-way partitioning

4.2. Bentley-McIlroy 3-way partitioning

4.3. Hàm qsort có sẵn trong thư viện stdlib.h

4.3. Hàm qsort có sẵn trong thư viện stdlib.h

Hàm qsort trong C (dùng thư viện stdlib.h):

```
void qsort(  
    void *buf,  
    size_t num,  
    size_t size,  
    int (*compare)(void const *,void const *)  
);
```

- Hàm qsort, sắp xếp **buf** (bao gồm **num** phần tử, mỗi phần tử có kích thước **size**)
Hàm compare được sử dụng để so sánh 2 phần tử trong **buf**. Hàm này trả về:
 - 0 : hai tham số giống nhau
 - > 0 : tham số thứ nhất lớn hơn
 - < 0 : tham số thứ nhất nhỏ hơn.

```
int (*compare)(void const *,void const *)
```

Hàm compare được sử dụng để so sánh 2 phần tử. Hàm này trả về:

- 0 : hai tham số giống nhau
- > 0 : tham số thứ nhất lớn hơn
- < 0 : tham số thứ nhất nhỏ hơn.

Viết hàm: **int int_compare(void const* x, void const *y)** để so sánh hai phần tử kiểu int

```
int int_compare(void const* x, void const *y) {  
    int m, n;  
    m = *((int*)x);  
    n = *((int*)y);  
    if ( m == n ) return 0;  
    if ( m > n ) return 1;  
    if ( m < n ) return -1;  
}
```

Ví dụ 1: Sắp xếp mảng các số nguyên theo thứ tự tăng dần dùng hàm qsort có sẵn trong C

```
#include <stdio.h>
#include <stdlib.h>
int int_compare(void const* x, void const *y) {
    int m, n;
    m = *((int*)x);
    n = *((int*)y);
    if ( m == n ) return 0;
    if ( m > n ) return 1;
    if ( m < n ) return -1;
}

int main () {
    int a[] = { 20, 16, 100, 2, 12 };
    int n=5;
    printf("Day truooc khi sap xep: \n");
    for(int i = 0 ; i < n; i++ ) printf("%d ", arr[i]);
    /*Gọi hàm qsort có sẵn trong thư viện stdlib.h: */
    qsort (.....) ;

    printf("\n Dãy sắp xếp tăng dần bằng hàm qsort có sẵn trong C: \n");
    for(int i = 0 ; i < n; i++ ) printf("%d ", a[i]);
    return 0;
}
```

Ví dụ 1: Sắp xếp mảng các số nguyên theo thứ tự tăng dần dùng hàm qsort có sẵn trong C

```
#include <stdio.h>
#include <stdlib.h>
int int_compare(void const* x, void const *y) {
    int m, n;
    m = *((int*)x);
    n = *((int*)y);
    if ( m == n ) return 0;
    if ( m > n ) return 1;
    if ( m < n ) return -1;
    return ( m - n );
}
```

```
int main () {
    int a[] = { 20, 16, 100, 2, 12 };
    int n=5;
    printf("Day truooc khi sap xep: \n");
    for(int i = 0 ; i < n; i++ ) printf("%d ", arr[i]);
    /*Gọi hàm qsort có sẵn trong thư viện stdlib.h: */
    qsort(a, n, sizeof(int), int_compare);

    printf("\n Dãy sắp xếp tăng dần bằng hàm qsort có sẵn trong C: \n");
    for(int i = 0 ; i < n; i++ ) printf("%d ", a[i]);
    return 0;
}
```


Ví dụ 2: Sắp xếp mảng các số nguyên theo thứ tự giảm dần dùng hàm qsort có sẵn trong C

```
#include <stdio.h>
#include <stdlib.h>
int int_compare(void const* x, void const *y) {
    int m, n;
    m = *((int*)x);
    n = *((int*)y);
    if ( m == n ) return 0;
    if ( m > n ) return 1;
    if ( m < n ) return -1;
}

int main () {
    int a[] = { 20, 16, 100, 2, 12 };
    int n=5;
    printf("Day truooc khi sap xep: \n");
    for(int i = 0 ; i < n; i++ ) printf("%d ", arr[i]);
    /*Gọi hàm qsort có sẵn trong thư viện stdlib.h: */
    qsort(a, n, sizeof(int), int_compare);

    printf("\n Dãy sắp xếp giảm dần bằng hàm qsort có sẵn trong C: \n");
    for(int i = 0 ; i < n; i++ ) printf("%d ", a[i]);
    return 0;
}
```

Bài tập 3

Cho mảng các bản ghi lưu trữ thông tin về sinh viên lớp Việt Nhật K63 có cấu trúc như sau:

```
typedef struct
{
    int age, marks;
    char name[20];
} Student;
```

Viết chương trình:

- Nhập thông tin sinh viên từ bàn phím, lưu vào một mảng VNK63:
- Dùng hàm qsort có sẵn của C, sắp xếp mảng VNK63 theo thứ tự tăng dần về điểm của sinh viên

```
int int_markcompare (const void * x, const void * y);
```

- Dùng hàm qsort có sẵn của C, sắp xếp mảng VNK63 theo thứ tự giảm dần về tuổi của sinh viên

```
int int_agecompare (const void * x, const void * y);
```

- In thông tin danh sách sinh viên:

```
void PrintStudent(Student sv[], int size);
```

Sắp xếp sinh viên theo thứ tự điểm tăng dần

```
int (*compare)(void const *,void const *)
```

Hàm compare được sử dụng để so sánh 2 phần tử. Hàm này trả về:

- 0 : hai tham số giống nhau
- > 0 : tham số thứ nhất lớn hơn
- < 0 : tham số thứ nhất nhỏ hơn.

Viết hàm: `int mark_compare(void const* x, void const *y)` để so sánh điểm của 2 sinh viên

```
int mark_compare(void const* x, void const *y) {  
    int m, n;  
    m = ((Student *)x)->mark;  
    n = ((Student *)y)->mark;  
    return (m-n); //sắp xếp tăng dần  
    /*if ( m == n ) return 0;  
    if ( m > n ) return 1;  
    if ( m < n ) return -1;*/  
}
```

```
#include <stdio.h>
#include <stdlib.h>
```

```
typedef struct
```

```
{
    int age, mark;
    char name[20];
}Student;
```

```
void InputStudent(Student sv[], int *size){
    printf("So luong sinh vien ban muon nhap (<100)? "); scanf("%d", size);
    for (int i=0; i< *size; i++){
        printf("***** Nhap thong tin cho sinh vien thu %d ***** \n",i+1);
        printf("Name:"); scanf("%s",sv[i].name);
        printf("Age:"); scanf("%d", &sv[i].age);
        printf("Mark:"); scanf("%d", &sv[i].mark);
    }
}
```

```
void PrintStudent(Student sv[], int size){
    printf("Danh sach sinh vien: \n");
    for (int i=0; i<size; i++){
        printf("***** Sinh vien thu %d ***** \n",i+1);
        printf("Name: %s \n",sv[i].name);
        printf("Age: %d \n", sv[i].age);
        printf("Mark: %d \n", sv[i].mark);
    }
}
```

```
int int_markcompare (const void * x, const void * y) {
    int m = ((Student *)x)->mark;
    int n = ((Student *)y)->mark;
    return (m-n); //Sap xep tang dan
}

int main () {
    Student VNK63[10]; int n;
    InputStudent(VNK63, &n);
    qsort(VNK63, n, sizeof(Student), int_markcompare);
    printf("Danh sach sinh vien sau khi sap xep diem theo thu tu tang dan: \n");
    PrintStudent(VNK63, n);
    return 0;
}
```

Sắp xếp sinh viên theo thứ tự tuổi giảm dần

```
int (*compare)(void const *,void const *)
```

Hàm compare được sử dụng để so sánh 2 phần tử. Hàm này trả về:

- 0 : hai tham số giống nhau
- > 0 : tham số thứ nhất lớn hơn
- < 0 : tham số thứ nhất nhỏ hơn.

Viết hàm: `int age_compare(void const* x, void const *y)` để so sánh tuổi của 2 sinh viên

```
int age_compare(void const* x, void const *y) {  
    int m, n;  
    m = ((Student *)x)->age;  
    n = ((Student *)y)->age;  
    return (n-m); //sắp xếp giảm dần  
    /*if ( m == n ) return 0;  
    if ( m > n ) return -1;  
    if ( m < n ) return 1;*/  
}
```

➔ Câu lệnh gọi qsort để sắp xếp mảng VNK63 theo thứ tự giảm dần về tuổi:

```
qsort(VNK63, n, sizeof(Student), int_agecompare);
```

```

int int_agecompare (const void * x, const void * y) {
    int m = ((Student *)x)->age;
    int n = ((Student *)y)->age;
    return (n-m); //Sap xep giam dan
}

int main () {
    Student VNK63[10]; int n;
    InputStudent(VNK63, &n);
    qsort(VNK63, n, sizeof(Student), int_markcompare);
    printf("Danh sach sinh vien sau khi sap xep diem theo thu tu tang dan: \n");
    PrintStudent(VNK63, n);

    qsort(VNK63, n, sizeof(Student), int_agecompare);
    printf("Danh sach sinh vien sau khi sap xep tuoi theo thu tu giam dan: \n");
    PrintStudent(VNK63, n);
    return 0;
}

```

Bài tập 3

Cho mảng các bản ghi lưu trữ thông tin về sinh viên lớp Việt Nhật K63 có cấu trúc như sau:

```
typedef struct
{
    int age, marks;
    char name[20];
}Student;
```

Viết chương trình:

- Dùng hàm qsort có sẵn của C, sắp xếp tên các sinh viên trong mảng VNK63 theo thứ tự abc

```
int string_compare (const void * x, const void * y);
int string_compare(const void *x, const void *y)
{
    Student *s1 = (Student *)x;
    Student *s2 = (Student *)y;
    return strcmp(s1->name, s2->name);
    /* strcmp functions works exactly as expected from
    comparison function */
}
```

Trong hàm main gọi: `qsort(VNK63, n, sizeof(Student), string_compare);`
`PrintStudent(VNK63, n);`

Con trỏ hàm (function pointer)

Con trỏ hàm cũng là 1 con trỏ có định kiểu. Ta có thể sử dụng con trỏ hàm để gọi hàm khi đã biết địa chỉ của hàm

Ví dụ:

- Khai báo một hàm kiểu int có 2 tham số:
 - `int min(int a, int b);`
- Khai báo một con trỏ hàm để trỏ tới hàm ở trên:
 - `int (*pf) (int, int);`
- Gán một con trỏ hàm tới hàm:
 - `pf = &min;`
- Gọi hàm qua con trỏ:
 - `int ans = pf(5,4); // tương tự với int ans = min(5,4)`

```
#include <stdio.h>
int min(int a,int b)
{
    if (a>b) return b;
    return a;
}
int main()
{
    int (*pf)(int,int);
    pf=min;
    printf("Min cua 4 va 5 la %d",pf(5,4));
    return 0;
}
```

`int (*compare) (void const *,void const *)`

Trong hàm qsort, compare là con trỏ hàm, tham chiếu tới hàm sắp xếp 2 phần tử

Cách khai báo con trỏ hàm:

`<type> (* <name_of_pointer>) (<data_type_of_parameters>);`

Con trỏ hàm (function pointer)

Cách khai báo con trỏ hàm:

```
<type> (* <name_of_pointer>) ( <data_type_of_parameters>);
```

Ví dụ:

- Viết hàm swap: `void swap(int *value1, int *value2)`
- Tạo một con trỏ hàm trỏ đến hàm swap:

```
void (*pswap) (int *, int *) = swap;
```

- Gọi hàm swap thông qua con trỏ hàm:

```
pswap(&a, &b); //tương đương với gọi: swap(&a, &b);
```

```
#include <stdio.h>
void swap(int *value1, int *value2)
{
    int temp = *value1;
    *value1 = *value2;
    *value2 = temp;
}

int main()
{
    void (*pswap) (int *, int *) = swap;
    int a = 6, b=10;
    printf("Gia tri a = %d, b =%d \n",a,b);
    pswap(&a,&b); //~ swap(&a, &b)
    printf("Sau khi hoan doi, gia tri a = %d, b =%d \n",a,b);
    return 0;
}
```

Bài tập 3

- Tạo mảng a ngẫu nhiên gồm 1.000.000 phần tử có giá trị khoảng 0-10
- Sao chép mảng a sang mảng b, c
- Tính thời gian sắp xếp mảng theo thứ tự giảm dần của các giải thuật
 - 3way partition quicksort trên mảng a
 - 2way partition quick sort trên mảng b
 - Selection sort trên mảng c
- Hiển thị 3 thời gian này ra màn hình

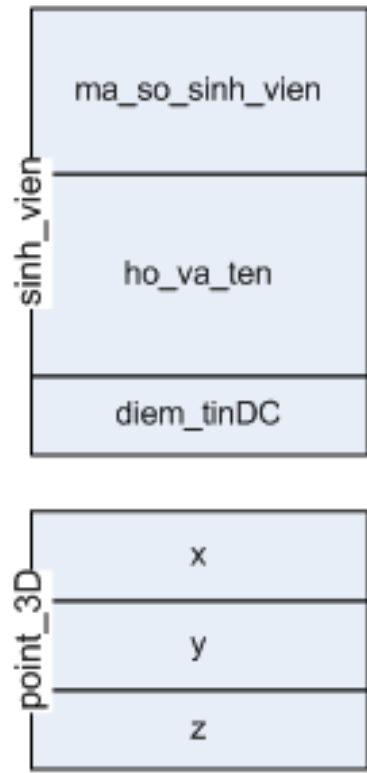
Bài tập 3

- Tạo mảng a ngẫu nhiên gồm 1.000.000 phần tử có giá trị khoảng 0-10
- Sao chép mảng a sang mảng b, c
- Tính thời gian sắp xếp mảng theo thứ tự giảm dần của các giải thuật
 - 3way partition quicksort trên mảng a
 - 2way partition quick sort trên mảng b
 - Selection sort trên mảng c
- Hiển thị 3 thời gian này ra màn hình

Khai báo kiểu dữ liệu cấu trúc

- Khai báo kiểu cấu trúc

```
struct tên_cấu_trúc{  
    <khai báo các trường >  
}
```



- Ví dụ

```
struct sinh_vien{  
    char ma_so_sinh_vien[10];  
    char ho_va_ten[30];  
    float diem_tinDC;  
}
```

```
struct point_3D{  
    float x;  
    float y;  
    float z;  
}
```

Khai báo biến cấu trúc

- Cú pháp:

```
struct <tên_cấu_trúc> <tên_biến_cấu_trúc>;
```

- Ví dụ:

```
struct sinh_vien a, b, c;
```

- Kết hợp khai báo

```
struct [tên_cấu_trúc] {  
    <khai báo các trường dữ liệu>;  
} tên_biến_cấu_trúc;
```

Khai báo biến cấu trúc

- Các cấu trúc có thể được khai báo lồng nhau

```
struct diem_thi {  
    float dToan, dLy, dHoa;  
}  
struct thi_sinh{  
    char SBD[10];  
    char ho_va_ten[30];  
    struct diem_thi ket_qua;  
} thi_sinh_1, thi_sinh_2;
```

- Có thể khai báo trực tiếp các trường dữ liệu của một cấu trúc bên trong cấu trúc khác:

```
struct thi_sinh{  
    char SBD[10];  
    char ho_va_ten[30];  
    struct diem_thi{  
        float dToan, dLy, dHoa;  
    } ket_qua;  
} thi_sinh_1, thi_sinh_2;
```

Khai báo biến cấu trúc

- Có thể gán giá trị khởi đầu cho một biến cấu trúc, theo nguyên tắc như kiểu mảng:

Ví dụ:

```
struct Date{
    int day;
    int month;
    int year;
};
struct{
    char Ten[20];
    struct Date NS;
} SV = {"Tran Anh", 20, 12, 1990 };
```

```
struct{
    char Ten[20];
    struct Date{
        int day;
        int month;
        int year;
    } NS;
} SV = {"Tran Anh", 20, 12, 1990 };
```


Định nghĩa kiểu dữ liệu với typedef

- Mục đích
 - Đặt tên mới cho kiểu dữ liệu cấu trúc
 - Giúp khai báo biến “quen thuộc” và ít sai hơn
- Cú pháp

```
typedef    struct <tên_cũ> <tên_mới>;
```

- Ví dụ

```
typedef char message[80];  
message str="xin chao cac ban";
```

Định nghĩa kiểu dữ liệu với typedef

- Với kiểu cấu trúc

```
typedef struct tên_cũ tên_mới
```

```
typedef struct [tên_cũ] {  
    <khai báo các trường >;  
} danh_sách_các_tên_mới;
```

- Chú ý: cho phép đặt tên_mới trùng tên_cũ

Ví dụ:

```
struct point_3D{  
    float x, y, z;  
}
```

```
struct point_3D M;
```

```
typedef struct point_3D toa_do_3_chieu;
```

```
toa_do_3_chieu N;
```

```
typedef struct {  
    float x, y, z;  
}point_3D;  
point_3D M;  
point_3D N;
```

Ví dụ

```
typedef struct tên_cũ {  
    <khai báo các trường >;  
} danh_sách_các_tên_mới;
```

Ví dụ:

```
typedef struct point_2D {  
    float x, y;  
} point_2D, diem_2_chieu, ten_bat_ki;  
point_2D X;  
diem_2_chieu Y;  
ten_bat_ki Z;
```

=> point_2D, diem_2_chieu, ten_bat_ki là các tên cấu trúc,
không phải tên biến

Xử lý dữ liệu cấu trúc

- Truy cập các trường dữ liệu
- Phép gán giữa các biến cấu trúc

Truy cập các trường dữ liệu

- Cú pháp

<tên_biến_cấu_trúc>.<tên_trường>

- Lưu ý
 - Dấu “.” là toán tử truy cập vào trường dữ liệu trong cấu trúc
 - Nếu trường dữ liệu là một cấu trúc => sử dụng tiếp dấu “.” để truy cập vào thành phần mức sâu hơn

Ví dụ

```
#include <stdio.h>
void main() {
    struct {
        char Ten[20];
        struct Date {
            int day;
            int month;
            int year;
        } NS;
    } SV = {"Tran Anh", 20, 12, 1990 };

    printf(" Sinh vien %s (%d/%d/%d)",
SV.Ten, SV.NS.day, SV.NS.month, SV.NS.year) ;
}
```

Sinh vien Tran Anh (20/12/1990)

Phép gán giữa các biến cấu trúc

- Muốn sao chép dữ liệu từ biến cấu trúc này sang biến cấu trúc khác cùng kiểu
 - gán lần lượt từng trường trong hai biến cấu trúc → “thủ công”
 - C cung cấp phép gán hai biến cấu trúc cùng kiểu:
`biến_cấu_trúc_1 = biến_cấu_trúc_2;`

Phép gán giữa các biến cấu trúc

Ví dụ:

- Xây dựng cấu trúc gồm họ tên và điểm môn lập trình C của sinh viên
- a, b, c là 3 biến cấu trúc.
- Nhập giá trị cho biến a.
- Gán b=a,
- Gán từng trường của a cho c.
- So sánh a, b và c ?

```
#include<stdio.h>
typedef struct{
    char hoten[20];
    int diem;
}sinhvien;
void main(){
    sinhvien a,b,c;
    printf("Nhap thong tin sinh vien\n");
    printf("Ho ten: ");gets(a.hoten);
    printf("Diem:");scanf("%d",&a.diem);
    b=a;
    strcpy(c.hoten,a.hoten);
    c.diem=a.diem;
    printf("Bien a: %-20s%3d\n",a.hoten,a.diem);
    printf("Bien b: %-20s%3d\n",b.hoten,b.diem);
    printf("Bien c: %-20s%3d\n",c.hoten,c.diem);
}
```

```
Nhap thong tin sinh vien
Ho ten: Nguyen Van Anh
Diem:9
Bien a: Nguyen Van Anh    9
Bien b: Nguyen Van Anh    9
Bien c: Nguyen Van Anh    9
```


Bộ nhớ động

Khi khai báo các biến mảng với độ dài cố định:

- ta chỉ lưu trữ một số lượng cố định các biến.
- kích thước không thể thay đổi sau khi biên dịch

Tuy nhiên, không phải lúc nào chúng ta cũng biết trước được chính xác dung lượng chúng ta cần.

➔ Việc dùng bộ nhớ động cho phép xác định bộ nhớ cần thiết trong quá trình thực hiện của chương trình, đồng thời giải phóng chúng khi không còn cần đến để dùng bộ nhớ cho việc khác.

1. Cấp phát: để cấp phát vùng nhớ cho con trỏ ta dùng thư viện chuẩn **stdlib.h**

1. **malloc**

2. **calloc**

3. **alloc**

2. Giải phóng

1. **free**

Cấp phát động malloc

malloc (memory allocation) : trả về địa chỉ byte đầu tiên của vùng bộ nhớ được cấp phát nếu cấp phát thành công, hoặc trả về NULL nếu cấp phát thất bại → luôn cần kiểm tra bộ nhớ có được cấp phát thành công hay không.

Ví dụ: `int *pointer = (int *) malloc(100);`

- Nếu được cấp phát thành công, 100 bytes bộ nhớ này sẽ nằm trên vùng heap. Vùng nhớ mới được cấp phát này có thể lưu được tối đa $100/4 = 25$ số nguyên int (4 bytes) hoặc tối đa $100/2 = 50$ số nguyên int (2 bytes).
- Để tránh khác biệt về kích thước dữ liệu (ví dụ: kiểu int có thể là 2 hoặc 4 bytes tùy thuộc vào kiến trúc máy tính và hệ điều hành), ta có thể sử dụng:

`int *pointer = (int *) malloc (25 * sizeof(int));`

→ cấp phát bộ nhớ cho 1 mảng số nguyên 25 phần tử

Hàm malloc trả về con trỏ kiểu void. Con trỏ kiểu void có thể ép được sang bất kỳ kiểu dữ liệu nào → do đó ta dùng (int *) để ép sang kiểu int

```
kieu_con_trỏ *ten_con_trỏ = (kieu_con_trỏ *) malloc (sizeof(kieu_con_trỏ));  
kieu_con_trỏ *ten_con_trỏ = (kieu_con_trỏ *) malloc (size * sizeof(kieu_con_trỏ));
```

Cấp phát động calloc

calloc (contiguous allocation) : cũng giống như malloc, calloc được dùng để cấp phát bộ nhớ động. Tuy nhiên, toàn bộ vùng nhớ được cấp phát bởi hàm calloc() sẽ được gán giá trị 0.

Ví dụ: Cấp phát bộ nhớ cho 1 mảng nguyên 25 phần tử

```
int *pointer = (int *) calloc(25, sizeof(int));
```

```
int *pointer = (int *) malloc (25 * sizeof(int));
```

Giải phóng bộ nhớ free

Khác với biến cục bộ và tham số của một hàm nằm trên vùng nhớ stack (sẽ được tự động giải phóng ngay sau khi ra khỏi phạm vi của hàm), vùng nhớ được cấp phát động nằm trên vùng heap sẽ không được giải phóng → Nếu không được giải phóng, chương trình có thể sẽ bị memory leak.

Để giải phóng vùng nhớ được cấp phát bởi malloc(), calloc(), realloc(), ta dùng hàm free()

Ví dụ:

```
int *pointer = (int *) calloc(25, sizeof(int));  
int *pointer = (int *) malloc (25 * sizeof(int));
```

→ free(pointer);

Chú ý: không xóa một vùng nhớ đã được cấp phát 2 lần

free(); → không làm gì cả

Tái cấp phát realloc

realloc (re-allocation) : cấp phát lại trên chính vùng nhớ đã cấp phát trước đó do vùng nhớ cấp phát trước đó không đủ hoặc quá lớn.

Ví dụ:

Cấp phát bộ nhớ cho 1 mảng nguyên 25 phần tử bằng malloc

```
int *pointer = (int *) malloc (25 * sizeof(int));
```

Cấp phát lại bộ nhớ chỉ cần 20 phần tử:

```
→ pointer = (int *) realloc(pointer, 20*sizeof(int));
```

Cấp phát lại bộ nhớ chỉ cần 50 phần tử:

```
→ pointer = (int *) realloc(pointer, 50*sizeof(int));
```

Bài 7

Nhập vào một dãy số nguyên từ bàn phím. In dãy số theo thứ tự ngược lại. Yêu cầu dùng cấp phát động

```
int main(void)
{
    int i, n, *p;

    printf("Nhập số lượng phần tử của mảng = ");
    scanf("%d", &n);

    /* Cấp phát bộ nhớ cho mảng số nguyên gồm n số */
    p = (int *)malloc(n * sizeof(int));
    if (p == NULL)
    {
        printf("Cấp phát bộ nhớ không thành công!\n");
        return 1;
    }
    /* Nhập các phần tử của mảng */
    printf("Hãy nhập các số:\n");
    ...
    for (i = 0; i < n; i++) scanf("%d", &p[i]);
    /* Hiển thị các phần tử của mảng theo thứ tự ngược lại */
    ...
    printf("Các phần tử theo thứ tự ngược lại là:\n");
    for (i = n - 1; i >= 0; --i) printf("%d ", p[i]);

    /* Giải phóng bộ nhớ đã cấp phát */
    free(p);
    return 0;
}
```

Bài 8

Cho khai báo hàm như sau:

- `char *my_strcat(char *s1, char *s2);`

Hàm **my_strcat**: nhận đầu vào là 2 chuỗi s1 và s2, đầu ra là con trỏ trỏ tới bộ nhớ cấp phát động lưu trữ chuỗi s1 nối với s2

Ví dụ: s1 = "hello"; s2 = "world" → chuỗi nối s1 với s2 = "hello world"

```
#define MAX_LEN 100
int main(void)
{
    char str1[MAX_LEN + 1], str2[MAX_LEN + 1];
    char *cat_str;
    printf("Hãy nhập vào 2 chuỗi\n");
    scanf("%100s", str1);
    scanf("%100s", str2);

    cat_str = my_strcat(str1, str2);

    printf("Chuỗi nối %s và %s là chuỗi: %s\n", str1, str2, cat_str);

    return 0;
}
```

Bài 8

Cho khai báo hàm như sau:

- `char *my_strcat(char *s1, char *s2);`

Hàm **my_strcat**: nhận đầu vào là 2 chuỗi s1 và s2, đầu ra là con trỏ trỏ tới bộ nhớ cấp phát động lưu trữ chuỗi s1 nối với s2

Ví dụ: s1 = "hello"; s2 = "world" → chuỗi nối s1 với s2 = "hello world"

```
char *my_strcat(char *str1, char *str2)
```

```
{
    int len1, len2;
    char *result;

    len1 = strlen(str1);
    len2 = strlen(str2);

    //Cấp phát bộ nhớ cho result:
    result = (char*)malloc((len1 + len2 + 1) * sizeof(char));
    //Kiểm tra cấp phát có thành công hay không:
    if (result == NULL) {
        printf("Allocation failed! Check memory\n");
        return NULL;
    }

    strcpy(result, str1);
    strcpy(result + len1, str2);

    return result;
}
```

?? Giải phóng bộ nhớ đã cấp phát

Bài 8

Cho khai báo hàm như sau:

- `char *my_strcat(char *s1, char *s2);`

Hàm **my_strcat**: nhận đầu vào là 2 chuỗi s1 và s2, đầu ra là con trỏ trỏ tới bộ nhớ cấp phát động lưu trữ chuỗi s1 nối với s2

Ví dụ: s1 = "hello"; s2 = "world" → chuỗi nối s1 với s2 = "hello world"

```
#define MAX_LEN 100
int main(void)
{
    char str1[MAX_LEN + 1], str2[MAX_LEN + 1];
    char *cat_str;
    printf("Hãy nhập vào 2 chuỗi\n");
    scanf("%100s", str1);
    scanf("%100s", str2);

    cat_str = my_strcat(str1, str2);

    if (cat_str == NULL)
    {
        printf("Cấp phát bộ nhớ bị lỗi!!!! \n");
        return 1;
    }
    printf("Chuỗi nối %s và %s là chuỗi: %s\n", str1, str2, cat_str);

    free(cat_str);
    return 0;
}
```

Case Study 1:

Bài toán: Xếp 1 file lớn chứa các bản ghi có kích thước nhỏ có trật tự ngẫu nhiên

Ví dụ: xử lý các bản ghi giao dịch của một công ty điện thoại

Lựa chọn thuật toán:

1. Sắp xếp nhanh (Quick sort)

- YES: $O(n \log n)$

2. Sắp xếp chèn (Insertion sort)

- No: $O(n^2)$ với các file có trật tự ngẫu nhiên

3. Sắp xếp chọn (Selection sort)

- NO: $O(n^2)$

Case Study 2:

Bài toán: Xếp 1 file lớn chứa các bản ghi gần như đã theo trật tự

Ví dụ: sắp xếp lại 1 CSDL lớn sau khi có một vài thay đổi nhỏ

Lựa chọn thuật toán:

1. Sắp xếp chèn (Insertion sort)

- YES: thời gian tuyến tính $O(n)$ với phần lớn dữ liệu đã được sắp xếp

2. Sắp xếp chọn (Selection sort)

- NO: $O(n^2)$

3. Sắp xếp nhanh (Quick sort)

- $O(n \log n)$: nhiều khả năng câu trả lời là “NO” (sắp xếp chèn đơn giản và nhanh hơn)

Case Study 3:

Bài toán: Xếp 1 file lớn chứa nhiều bản ghi với các khóa nhỏ

Ví dụ: tổ chức lại các file MP3

Lựa chọn thuật toán:

1. Sắp xếp chèn (Insertion sort)

- NO: quá nhiều phép đổi chỗ

2. Sắp xếp trộn (Merge sort)

- $O(n \log n)$: nhiều khả năng câu trả lời là “NO” (sắp xếp chọn đơn giản và nhanh hơn)

3. Sắp xếp chọn (Selection sort)

- YES: ($O(n)$ với một số giả thiết hợp lý)

Ví dụ: 5000 bản ghi, mỗi bản ghi có 2 triệu byte với các khóa 100 byte. Xét average case ($O(n)$ phép đổi chỗ, $n^2/2$ phép so sánh):

- Chi phí cho các phép so sánh: $100 \times 5000^2 / 2 = 1.25$ tỉ
- Chi phí đổi chỗ: $2000000 \times 5000 = 10$ nghìn tỉ
- Mergesort chậm hơn $\log(n) = \log(5000)$ lần