CÁC THUẬT TOÁN SẮP XẾP

1.Thuật toán sắp xếp chọn (Selection sort).

Tư tưởng: Ở mỗi bước của thuật toán luôn đưa phần tử nhỏ nhất và đưa về đầu dãy. Để sắp xếp dãy có n phần tử cần thực hiện n - 1 bước.

Độ phức tạp : $O(n^2)$

Pseudocode

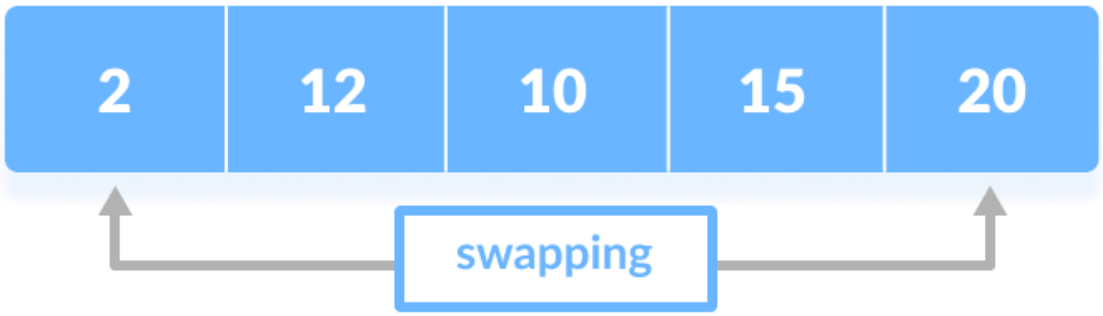
```
selectionSort(array, size)
  repeat (size - 1) times
  set the first unsorted element as the minimum
  for each of the unsorted elements
    if element < currentMinimum
      set element as new minimum
  swap minimum with first unsorted position
end selectionSort
```

Giả sử mảng [20, 12, 10, 15, 2]

Bước 1:

Chọn 20 là phần tử nhỏ nhất, sau đó duyệt các phần tử đứng sau 20, cập nhật chỉ số của phần tử nhỏ nhất sau đó hoán vị với 20.

Sau khi tìm được vị trí của phần tử nhỏ nhất sẽ hoán vị 20 với phần tử nhỏ nhất đó



2. Thuật toán sắp xếp nổi bọt (Bubble sort).

Tư tưởng : So sánh 2 phần tử liền kề và đưa phần tử lớn nhất nổi về cuối dãy

Độ phức tạp $O(n^2)$.

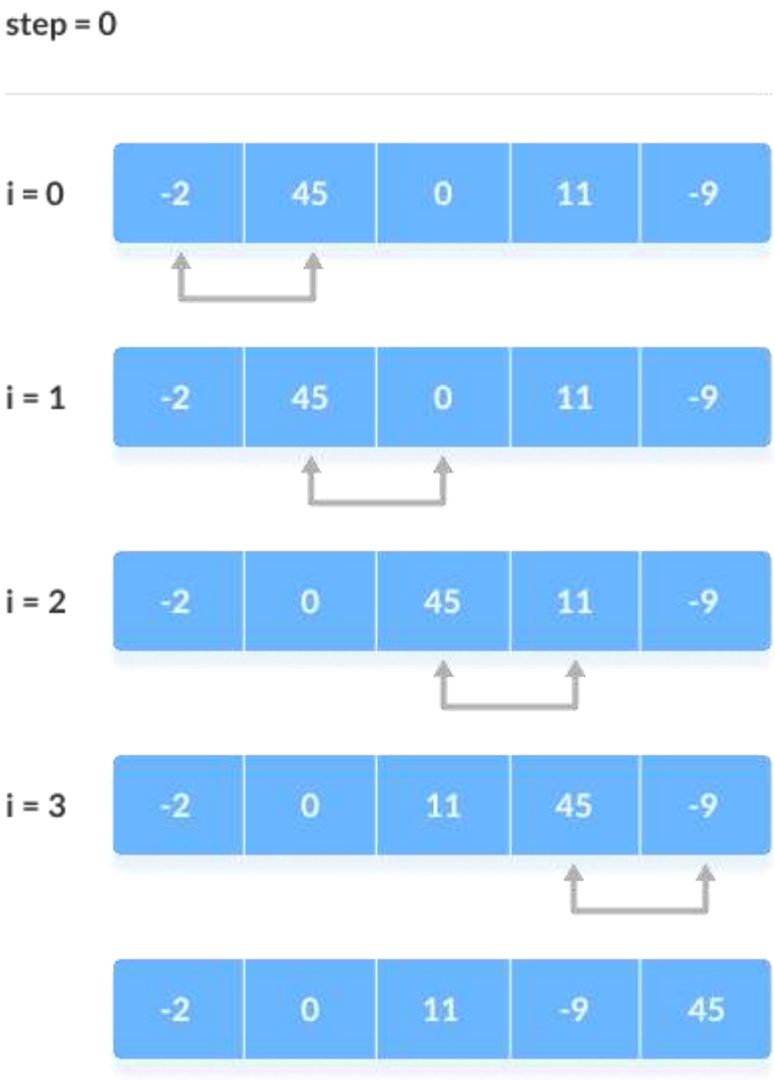
Pseudocode

```
1  bubbleSort(array)
2    for i <- 1 to indexOfLastUnsortedElement-1
3      if leftElement > rightElement
4        swap leftElement and rightElement
5  end bubbleSort
```

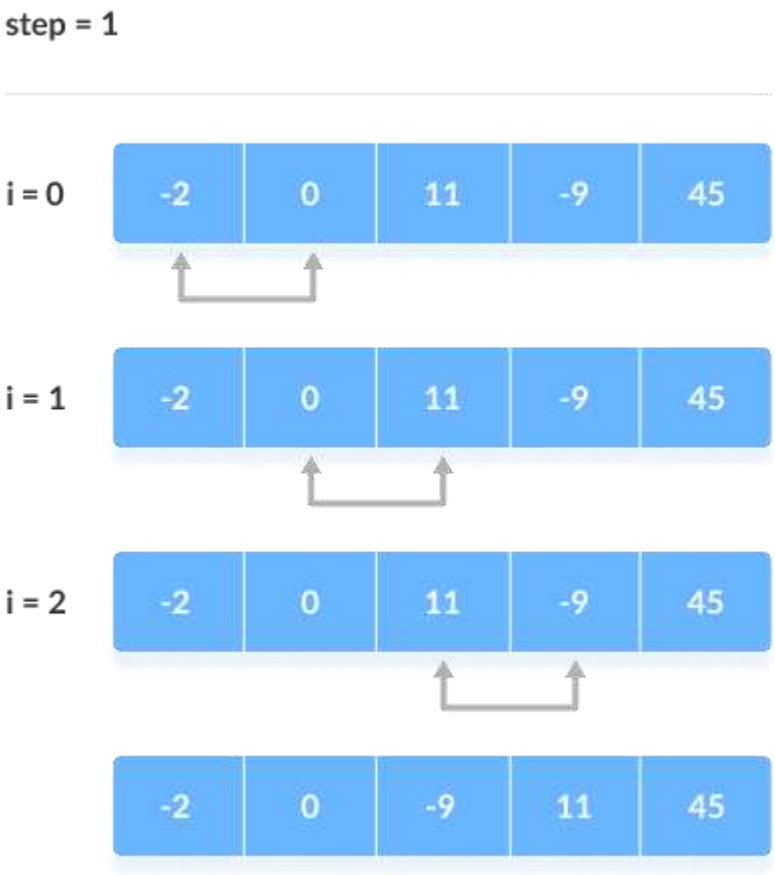
Một bước của thuật toán. Ở mỗi bước của thuật toán sẽ có 1 thêm 1 phần tử được sắp xếp và đưa về cuối dãy, ở các bước tiếp theo, các phần tử nãy sẽ không được xét lại nữa.

Xem xét bước đầu tiên của mảng [-2, 45, 0, 11, -9]

Bước đầu tiên :



Bước thứ 2. Đã có phần tử 45 ở bước trước được sắp đúng thứ tự nên sẽ không duyệt tới chỉ số trước của 45 nữa mà chỉ duyệt tới vị trí của số 11 trong dãy.



3. Thuật toán sắp xếp chèn (Insertion sort).

Tư tưởng: Ở mỗi bước của thuật toán sẽ cố gắng đưa phần tử ở vị trí hiện tại về đúng vị trí bằng cách chèn nó vào dãy các phần tử đứng trước nó sao cho đúng thứ tự.

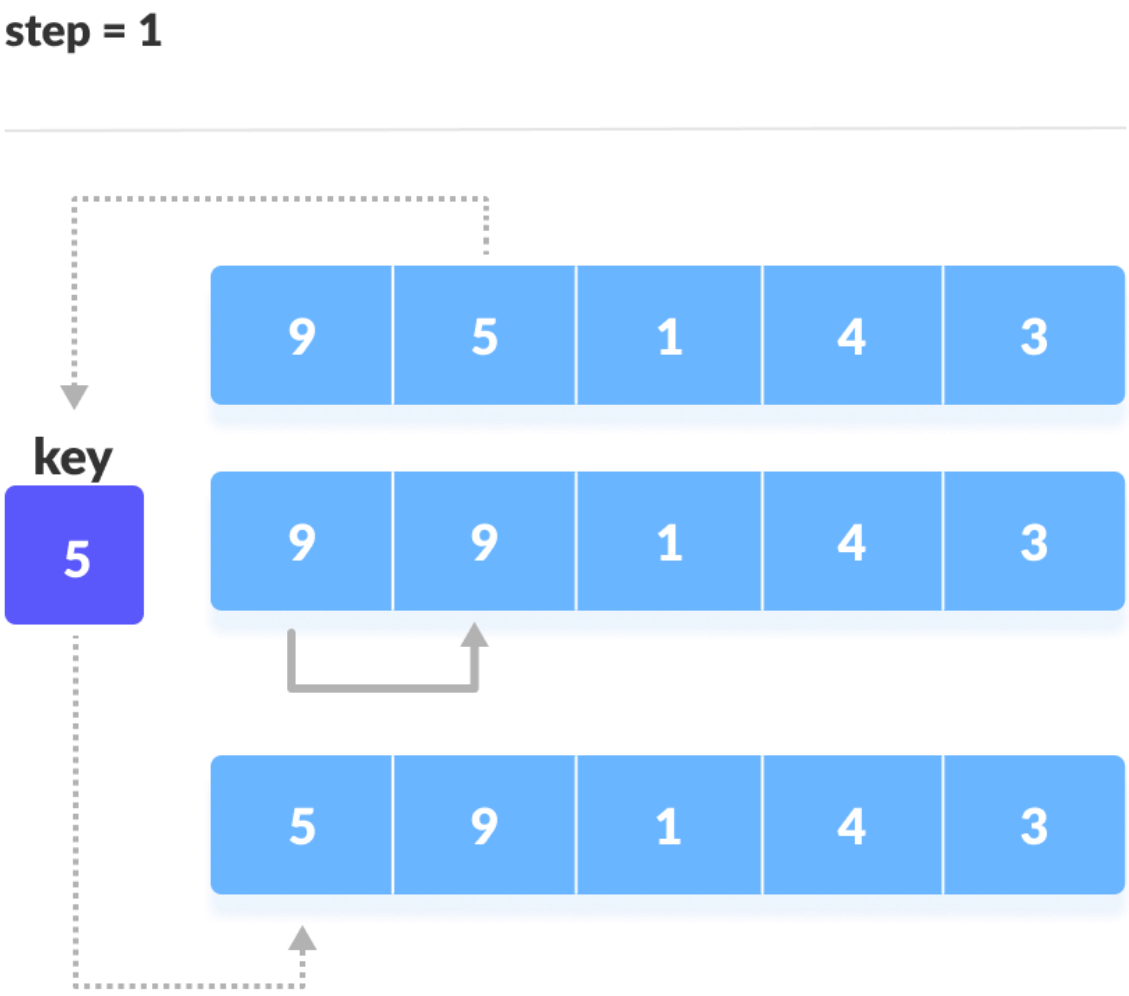
Độ phức tạp: $O(n^2)$.

Pseudocode

```
1
2  insertionSort(array)
3    mark first element as sorted
4    for each unsorted element X
5      'extract' the element X
6      for j <- lastSortedIndex down to 0
7        if current element j > X
8          move sorted element to the right by 1
9      break loop and insert X here
10 end insertionSort
```

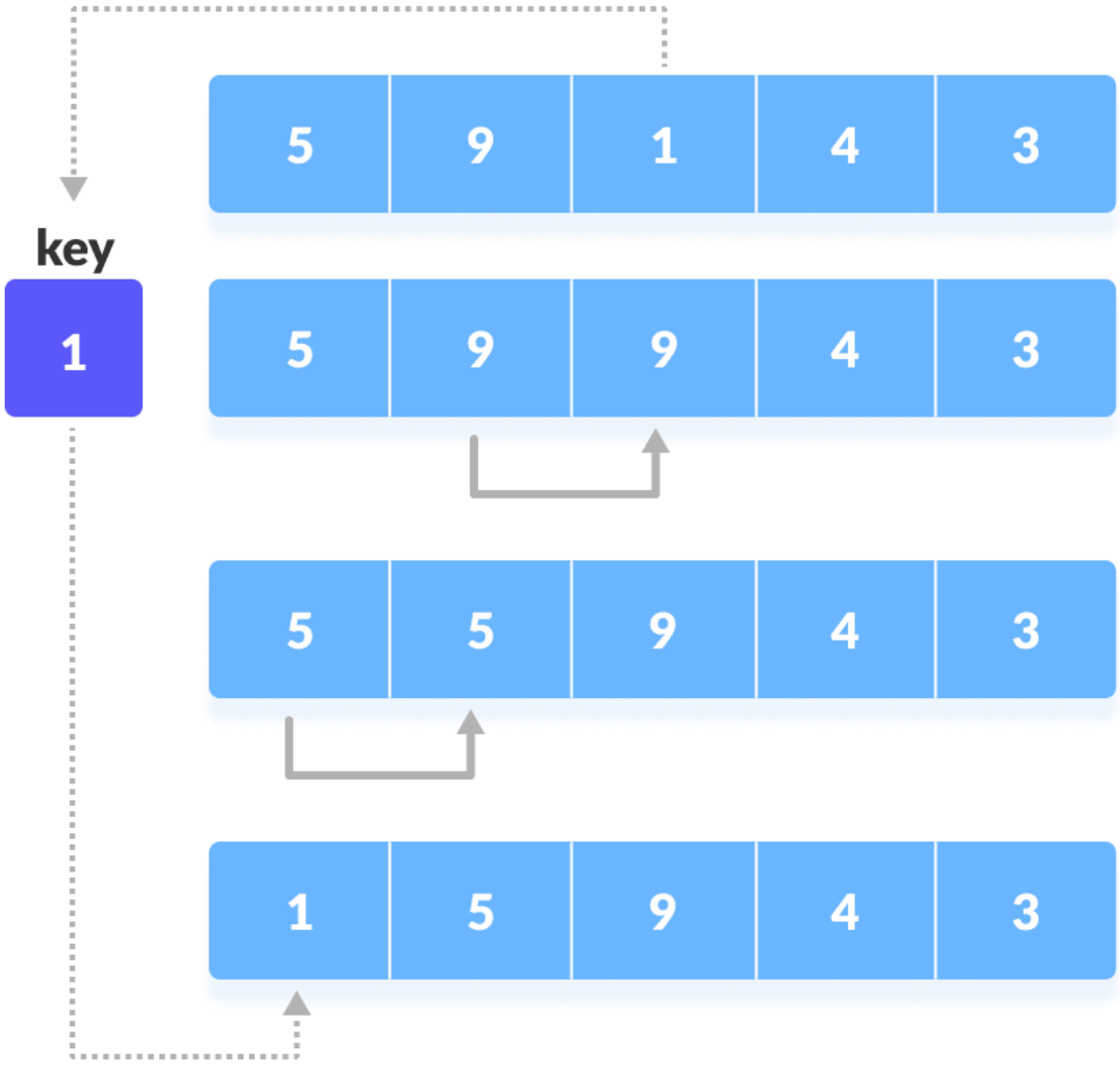
Mảng ban đầu

Bước 1.

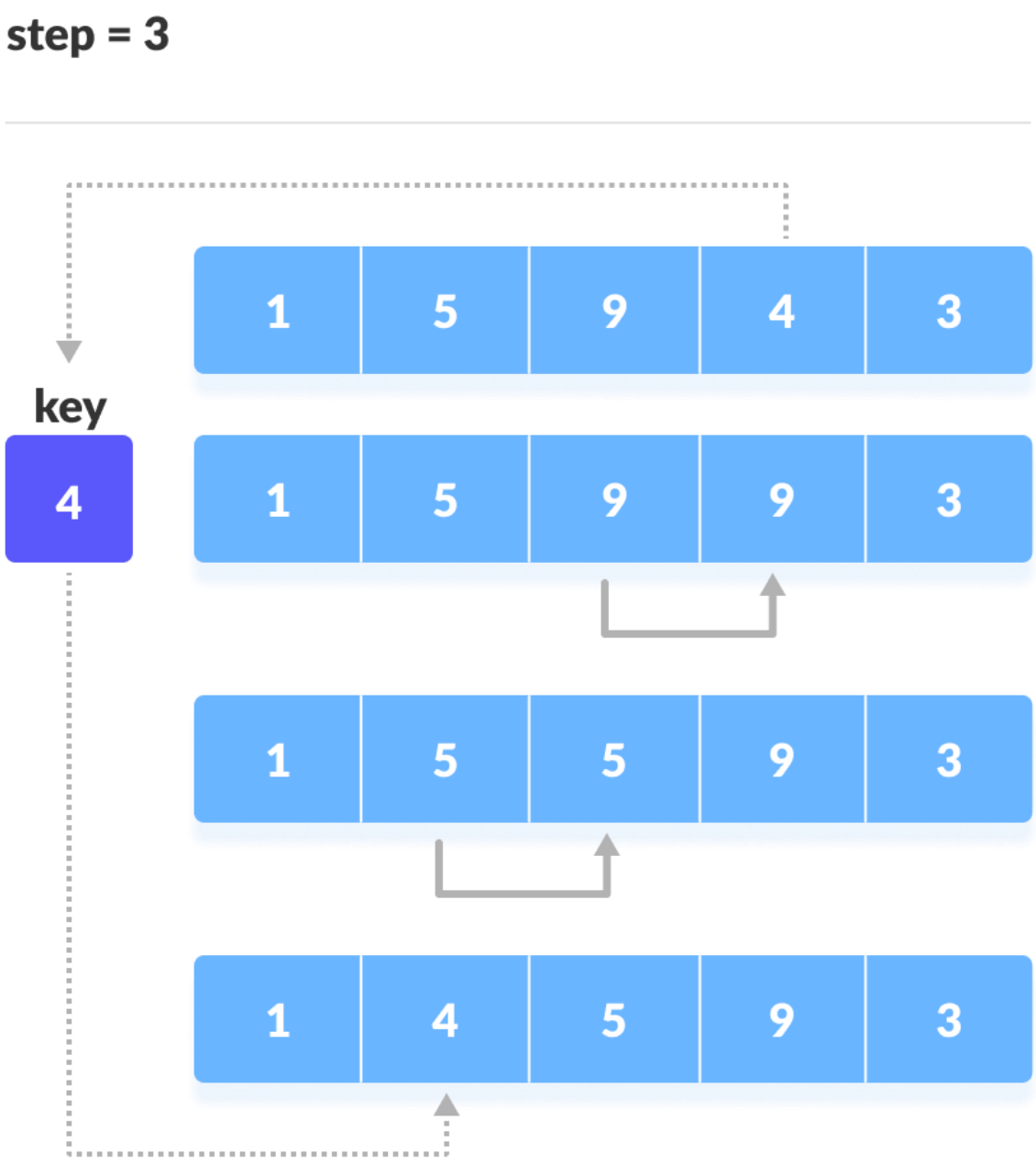


Bước 2.

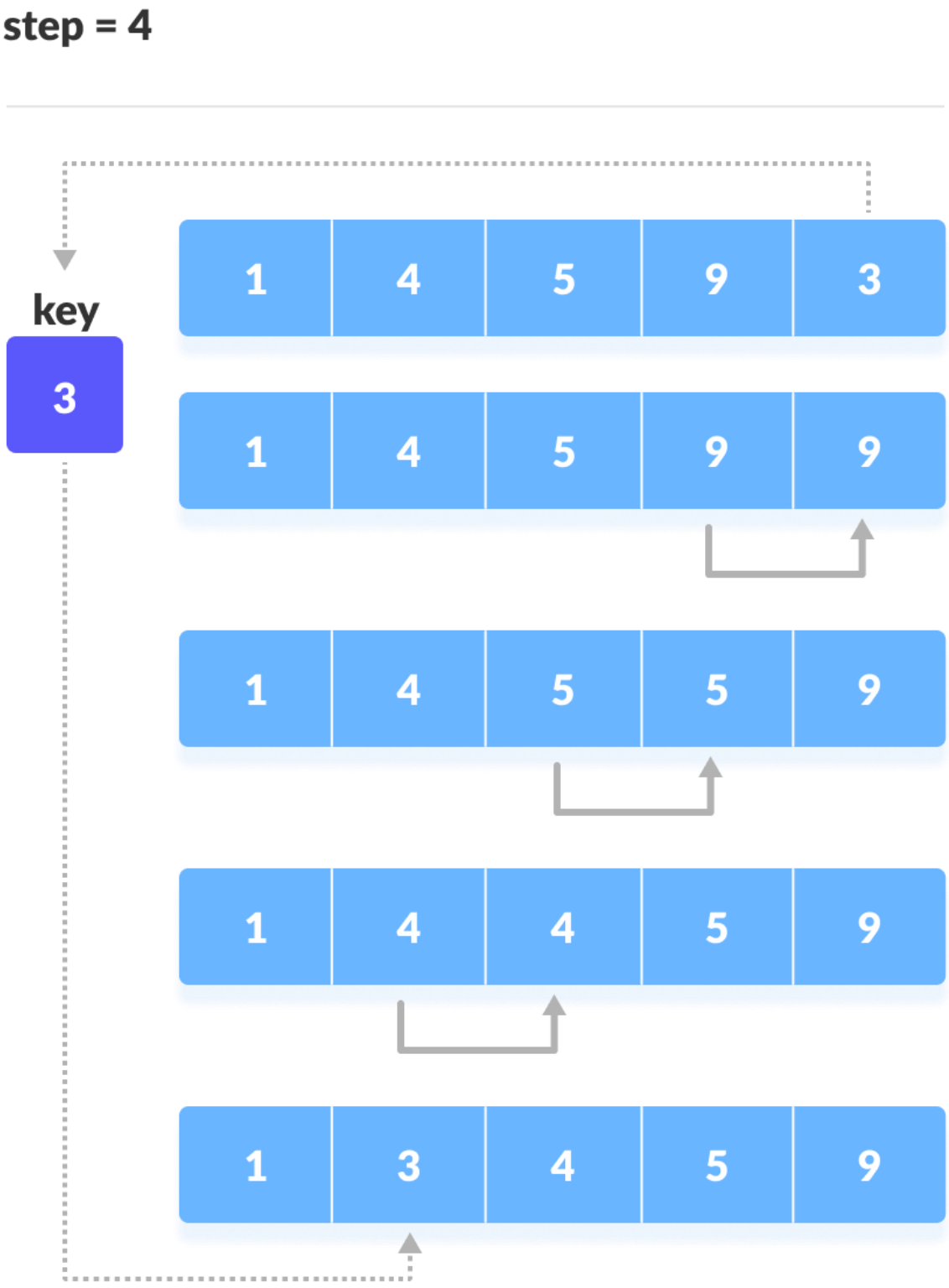
step = 2



Bước 3.



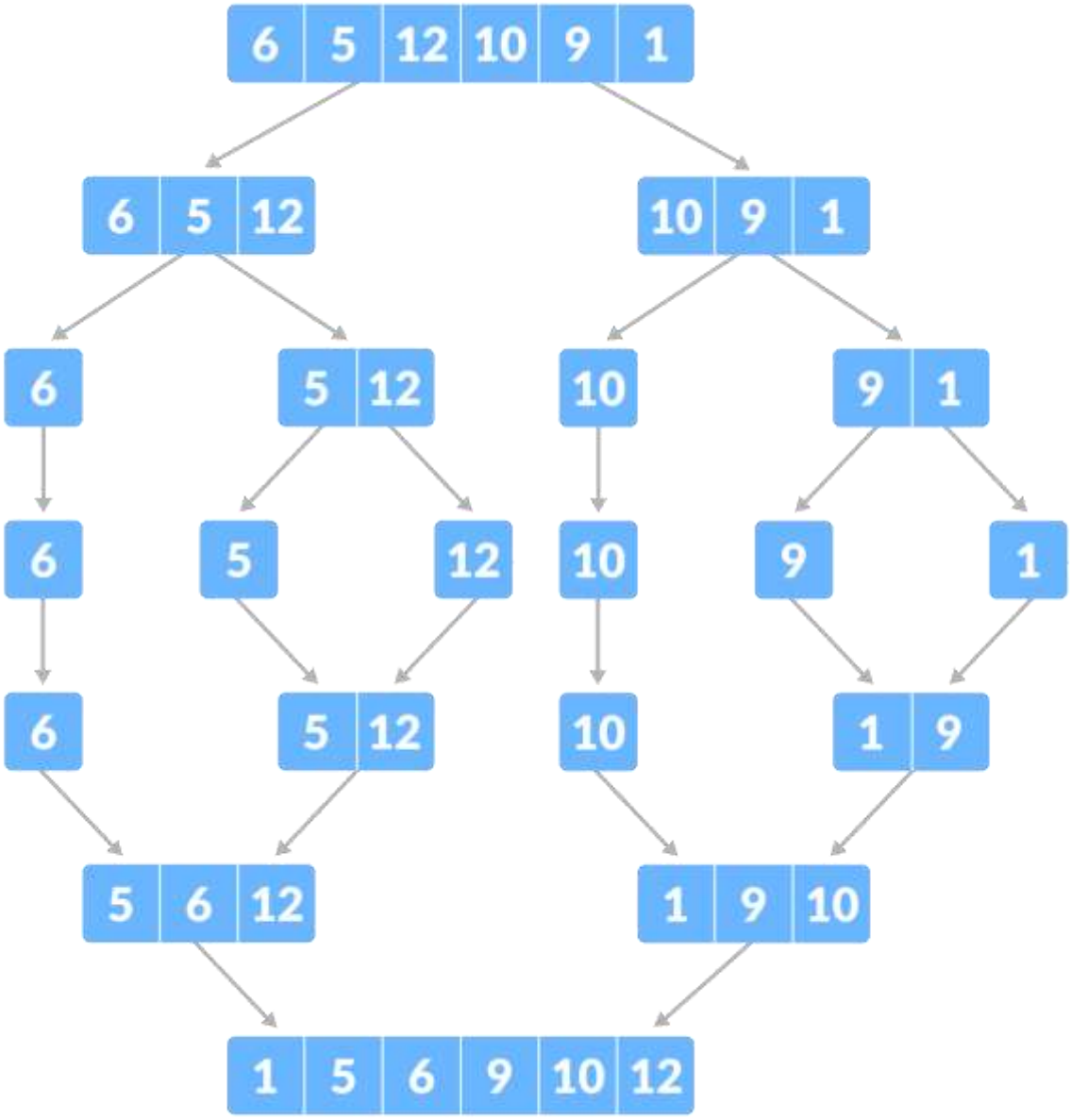
Bước 4.



4. Thuật toán sắp xếp trộn (Merge sort) và bài toán đếm số cặp nghịch thế trong mảng(Count inversions).

Tư tưởng : Thuật toán sắp xếp trộn sử dụng phương pháp chia và trị, chia dãy ban đầu thành các dãy con cho tới khi dãy chỉ còn 1 phần tử, sau đó thực hiện trộn 2 dãy con tăng dần thành 1 dãy tăng dần với độ phức tạp tuyến tính

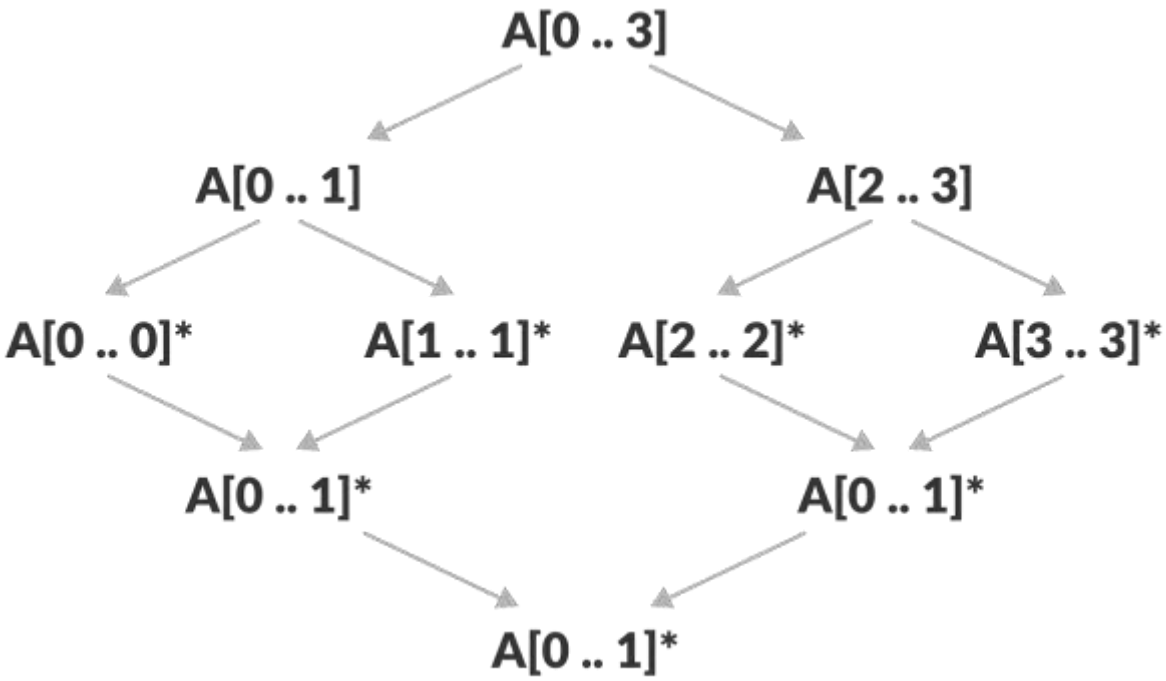
Độ phức tạp : O(nlogn).



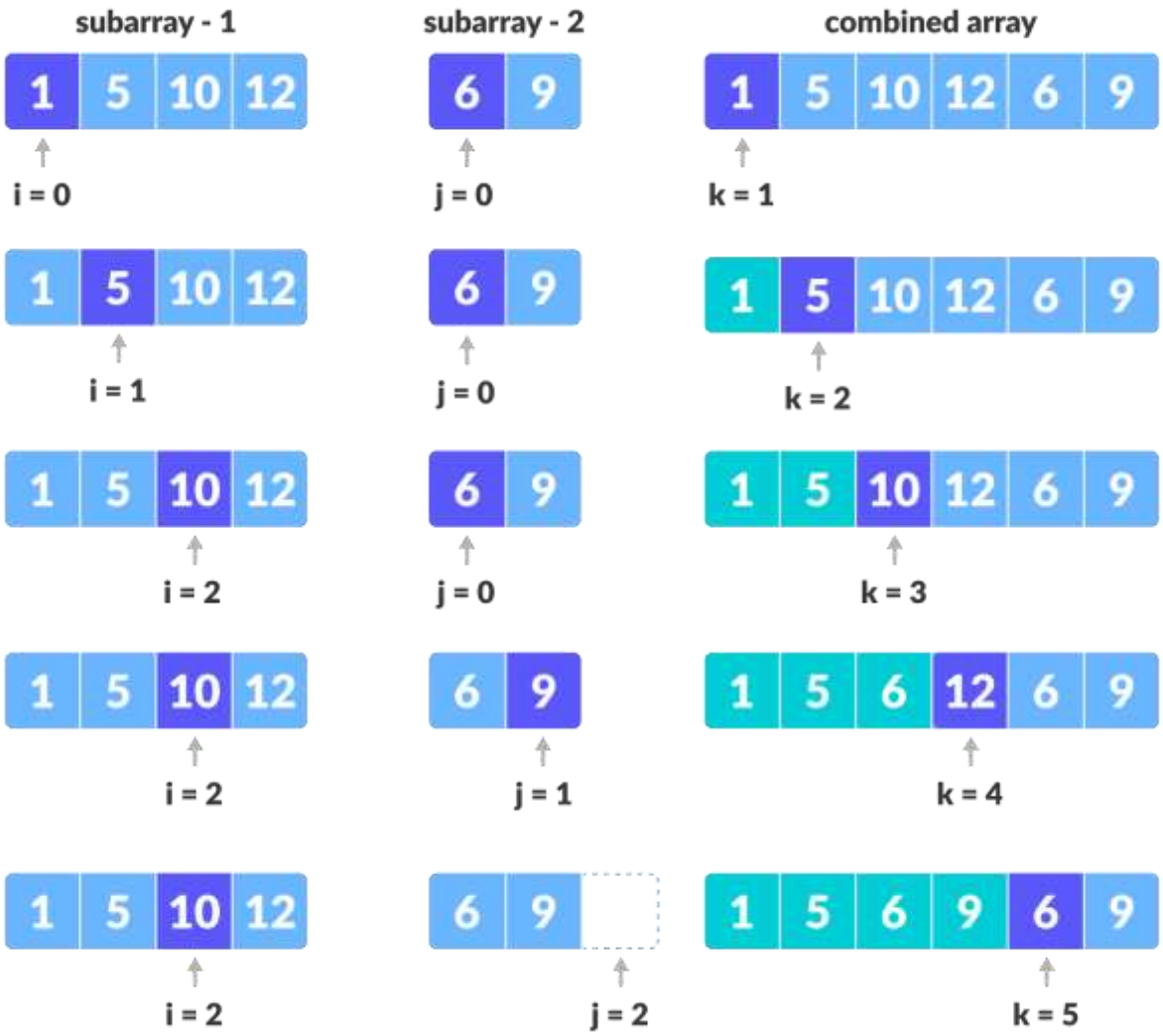
Pseudocode

```
MergeSort(A, l, r):
    if l >= r
        return
    m = (l+r)/2
    mergeSort(A, l, m)
    mergeSort(A, m+1, r)
    merge(A, l, m, r)
```

Quá trình thực hiện chia và trị của thuật toán



Thao tác trộn 2 dãy đã sắp xếp thành 1 dãy đã sắp xếp



5. Thuật toán sắp xếp Counting sort

Tư tưởng: Đếm phân phối tần suất các phần tử trong mảng theo thứ tự từ bé đến lớn

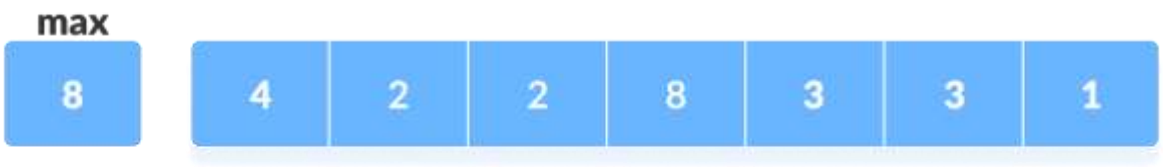
Độ phức tạp : O(n + k) trong đó k là số lớn nhất trong mảng.

Hạn chế : Chỉ áp dụng được với các mảng có phần tử là các số không âm và thường không vượt quá 10^7. Vì để thực hiện được thuật toán ta cần dùng một mảng để đếm số lần xuất hiện của các phần tử thông qua chỉ số của mảng.

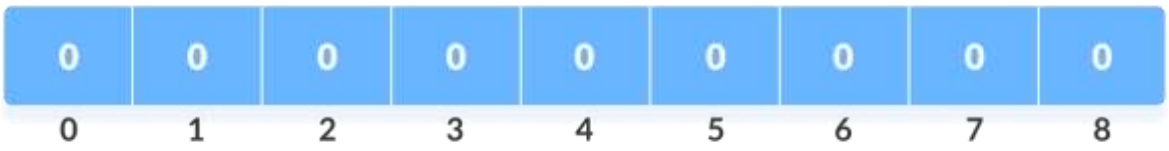
Pseudocode

```
countingSort(array, size)
  max <- find largest element in array
  initialize count array with all zeros
  for j <- 0 to size
    find the total count of each unique element and
    store the count at jth index in count array
  for i <- 1 to max
    find the cumulative sum and store it in count array itself
  for j <- size down to 1
    restore the elements to array
    decrease count of each element restored by 1
```

Tìm phần tử lớn nhất trong mảng:



Tạo một mảng để đếm tần suất các phần tử trong mảng



Duyệt qua mảng và tăng tần suất các phần tử trong mảng lên thông qua chỉ số của mảng đếm.

6. Thuật toán sắp xếp vun đống (Heap sort)

Tư tưởng : Sắp xếp vun đống hoạt động bằng cách hình dung các phần tử của mảng như một loại cây nhị phân hoàn chỉnh đặc biệt được gọi là heap

Độ phức tạp (O(nlogn))

Pseudocode

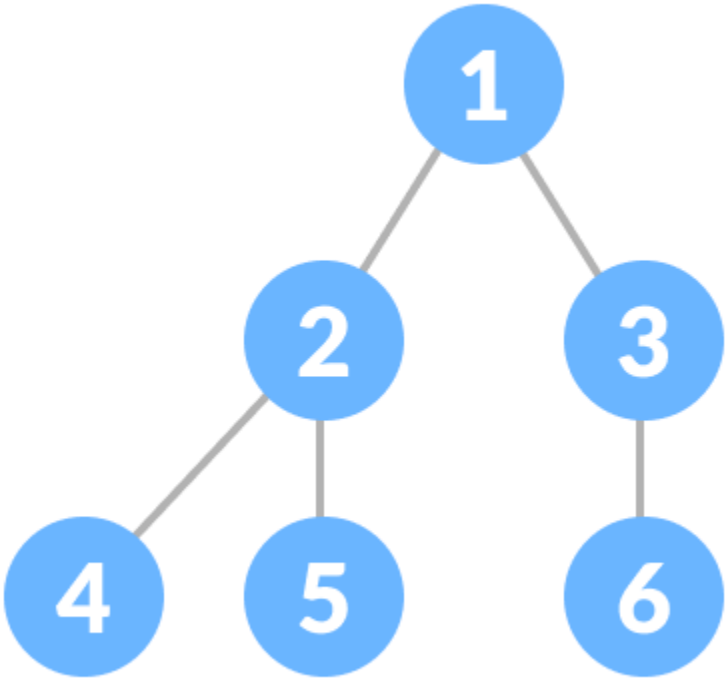
Định nghĩa cây nhị phân hoàn chỉnh : Complete Binary Tree

Cây nhị phân hoàn chỉnh là cây nhị phân trong đó tất cả các cấp (level) được lấp đầy hoàn toàn ngoại trừ có thể là cấp thấp nhất, và được điền từ bên trái.

Tất cả các phần tử lá phải nghiêng về bên trái

Nốt lá cuối cùng có thể không có anh chị em bên phải

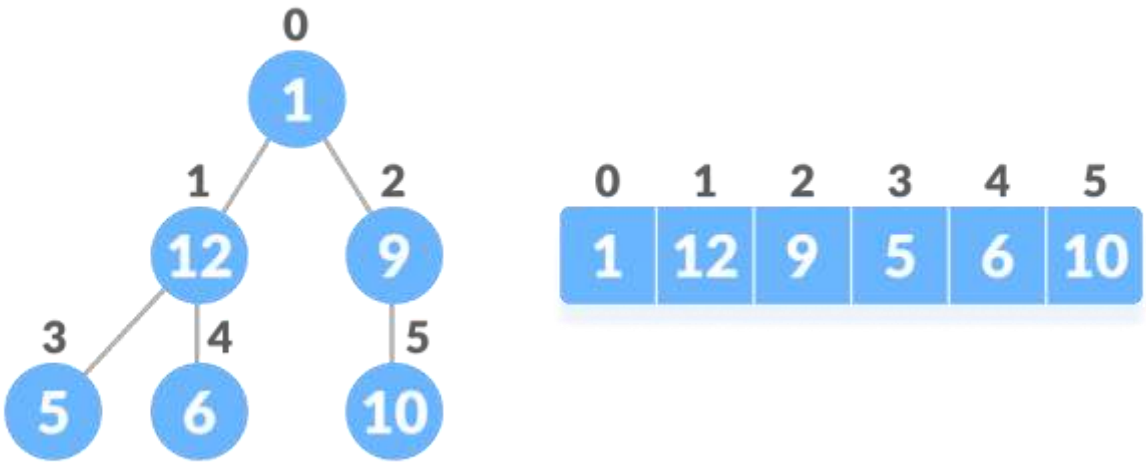
Một số ví dụ về cây nhị phân hoàn chỉnh



Xây dựng một cây nhị phân hoàn chỉnh tử mảng thông qua chỉ số của các phần tử.

Nếu chỉ số của một phần tử trong mảng là *I* thì nốt con bên trái của nó sẽ có chỉ số là *2I + 1* và nốt con bên phải có chỉ số *2I + 2*. Từ chỉ số *I* có thể tìm được ngay nốt con bên trái và nốt con bên phải, hoặc tử chỉ số của một phần tử *I* nào đó có thể tìm được chỉ số của nốt cha của nó là *(I-1)/2*

Ví dụ

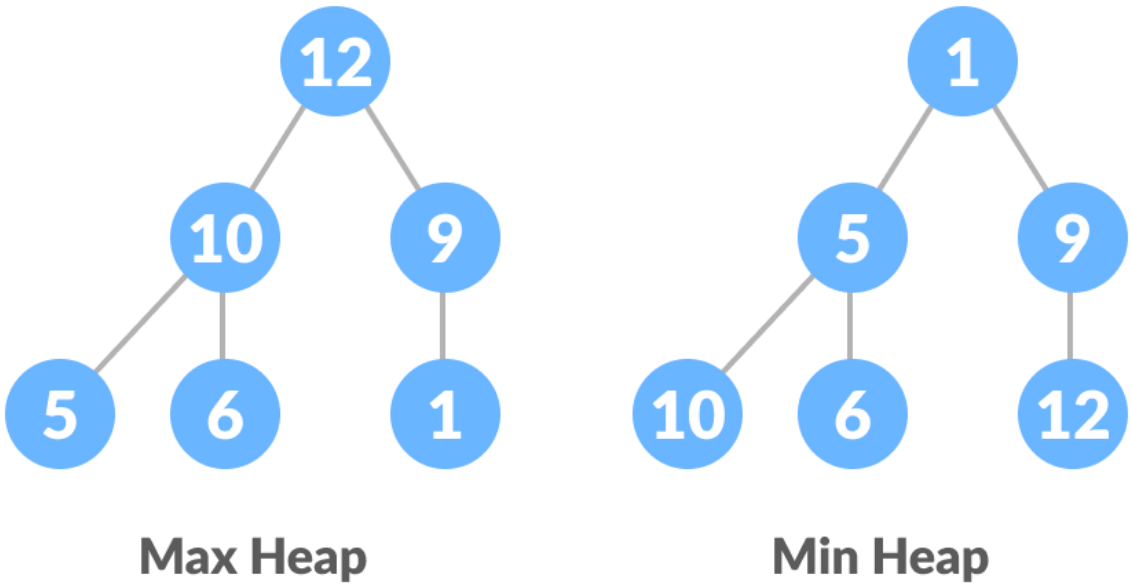


Cấu trúc dữ liệu Heap :

Heap là một cây nhị phân hoàn chỉnh

Tất cả các nút trong cây tuân theo thuộc tính đó là chúng lớn hơn phần tử con của chúng, tức là phần tử lớn nhất nằm ở gốc. Trong trường hợp nốt cha lớn hơn các nốt con của nó ta có Max-Heap, ngược lại nếu nốt cha nhỏ hơn 2 nốt con của nó ta có Min-Heap.

Ví dụ

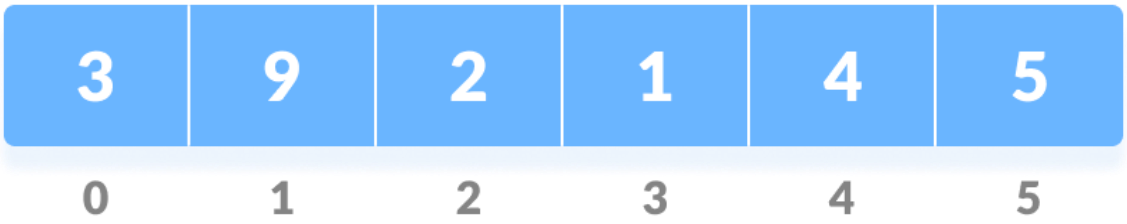


Heapify : Thao tác heapify với nốt có chỉ số i trong mảng

```
void heapify(int arr[], int n, int i) {
  int largest = i;
  int left = 2 * i + 1;
  int right = 2 * i + 2;

  if (left < n && arr[left] > arr[largest])
    largest = left;

  if (right < n && arr[right] > arr[largest])
    largest = right;
    if (largest != i) {
      swap(&arr[i], &arr[largest]);
      heapify(arr, n, largest);
  }
}
```

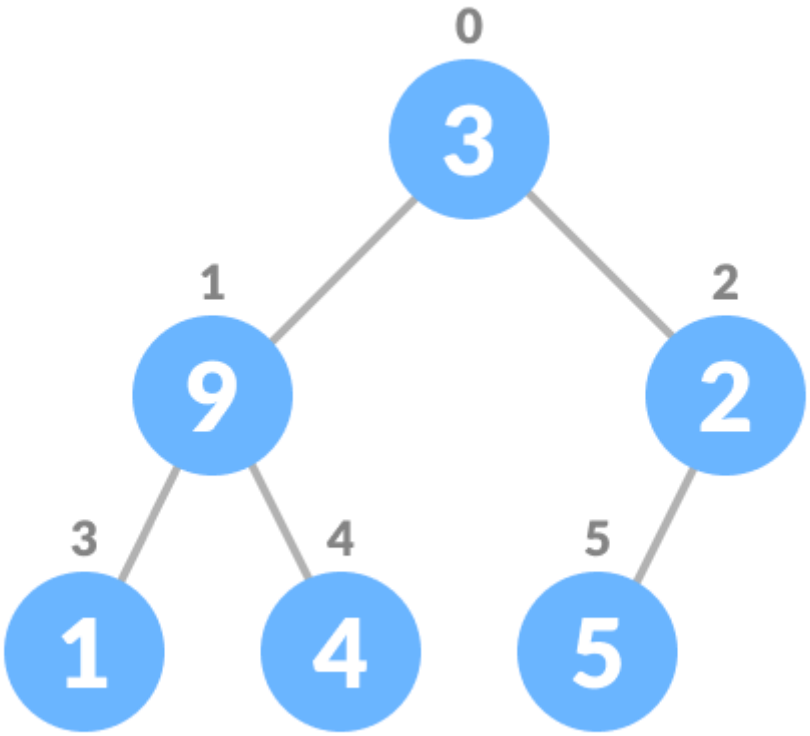
Bước 1 :

Mảng đầu vào :



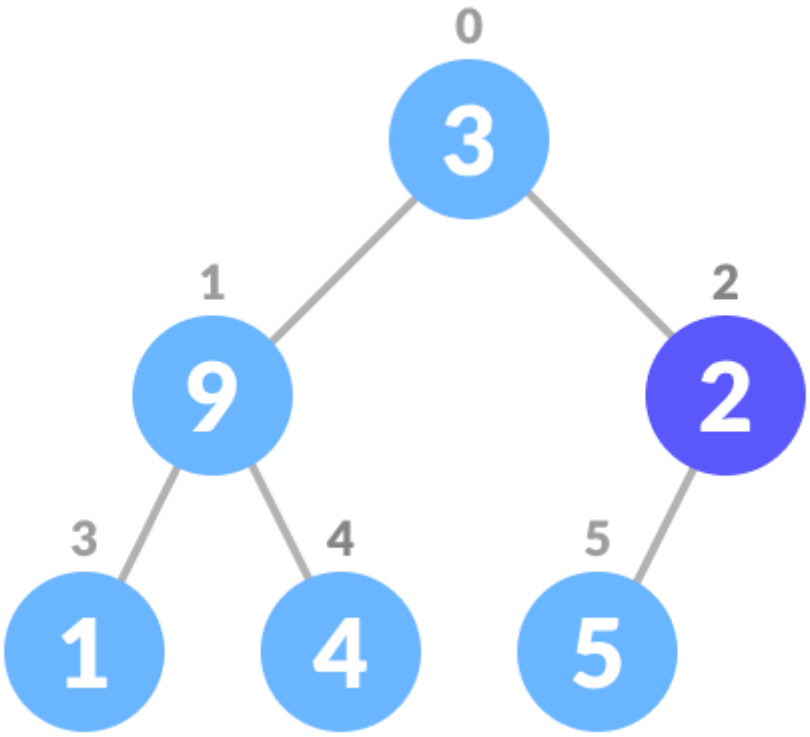
Bước 2 :

Tạo cây nhị phân hoàn chỉnh từ mảng đầu vào



Bước 3 :

Bắt đầu từ các nốt không phải là nốt lá



Bước 4:

Ta coi phần tử ở chỉ số i này là phần tử lớn nhất giữa 3 phần tử : Phần tử ở nốt gốc là chính nó, nốt con bên trái, nốt con bên phải

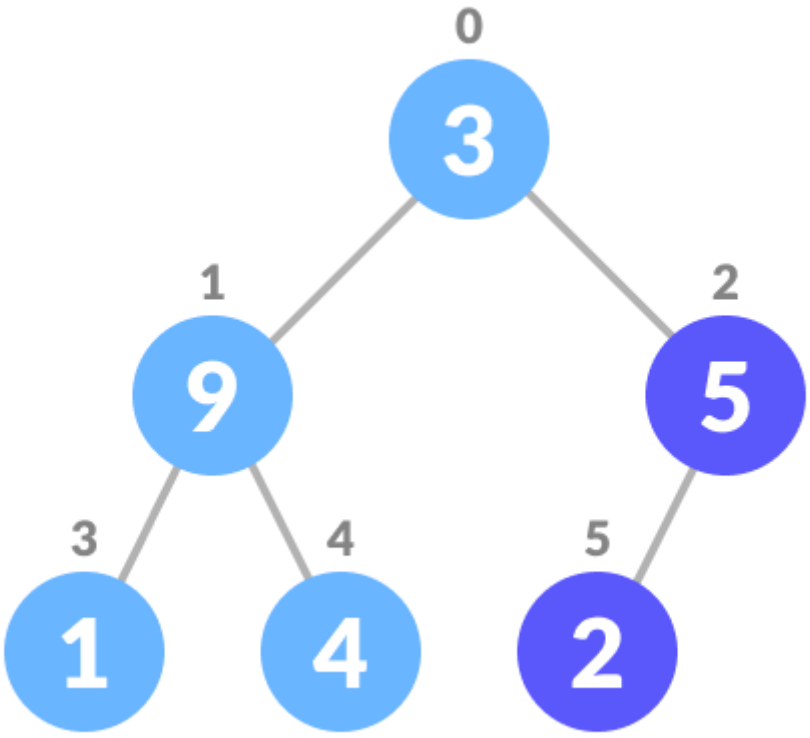
largest = i

Bước 5:

Lần lượt so sánh phần tử ở nút gốc với nút con bên trái, nếu nút con bên trái lớn hơn nút gốc thì cập nhật largest = 2 * i + 1, tiếp tục duyệt qua nút con bên phải, nếu nút con bên phải lớn hơn phần tử đang được coi là lớn nhất có chỉ số largest thì cập nhật largest = 2 * i + 2, vậy sau 2 lần so sánh ta tìm được chỉ số của phần tử lớn nhất trong 3 phần tử : Nút cha có chỉ số i, nút con bên trái có chỉ số 2 * i + 1, nút con bên phải có chỉ số 2 * i + 2. Trong trường hợp nút i không có con, hoặc không có con bên phải thao tác trên được thực hiện bình thường

Bước 6:

Hoán vị phần tử có chỉ số largest với nút gốc có chỉ số i



Bước 7 :

Lặp lại các bước từ 3 tới 7 cho tới khi subtree gốc i cũng đã được heapfied

Để xây dựng Heap ta gọi hàm Heapify với mọi nốt không phải nốt lá

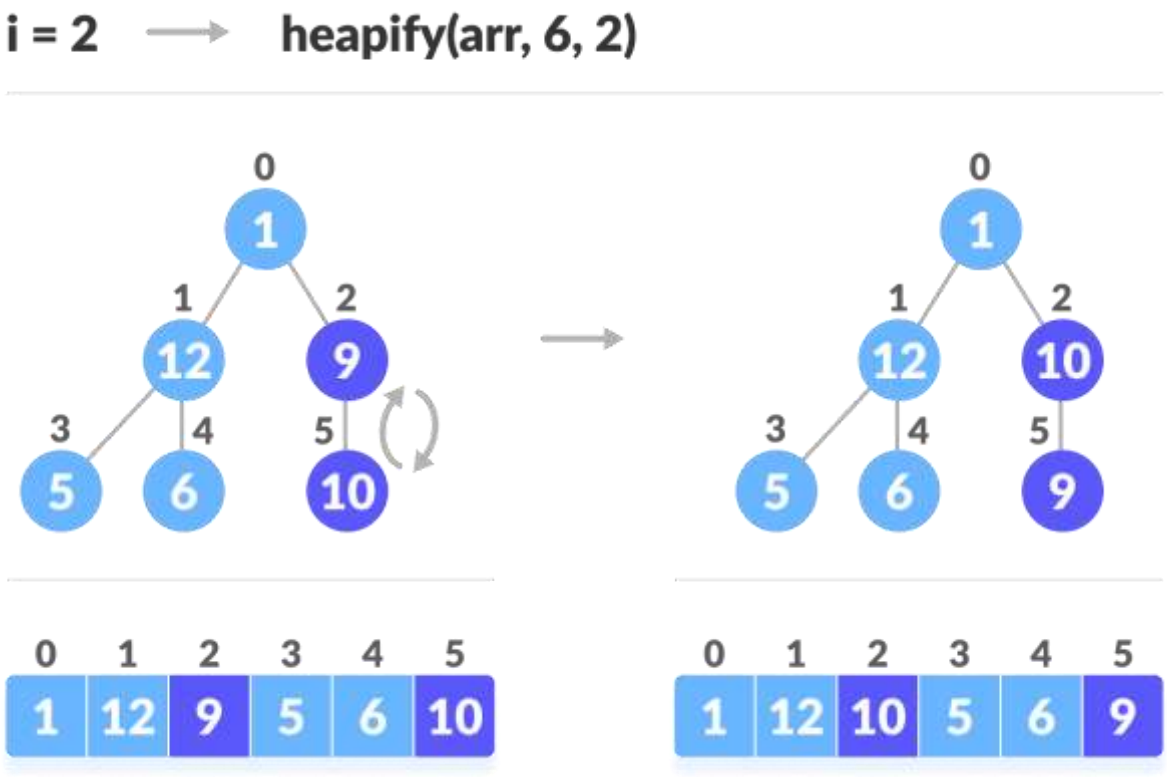
```
1   for (int i = n / 2 - 1; i >= 0; i--)
2       heapify(arr, n, i);
```

Build Max-heap

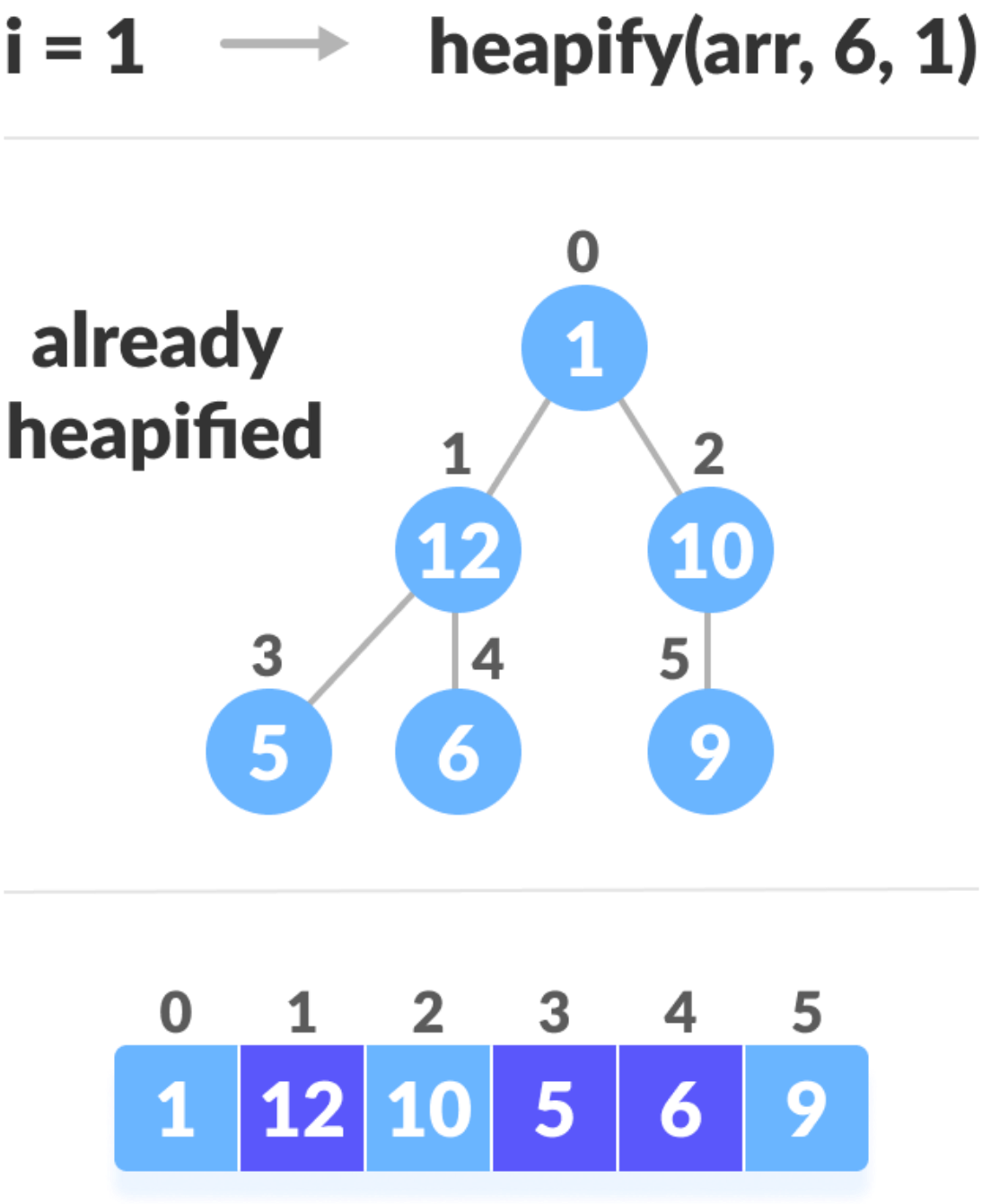
n = 6

i = 6/2 - 1 = 2 # loop runs from 2 to 0

Gọi hàm heapify với nốt ở chỉ số 2

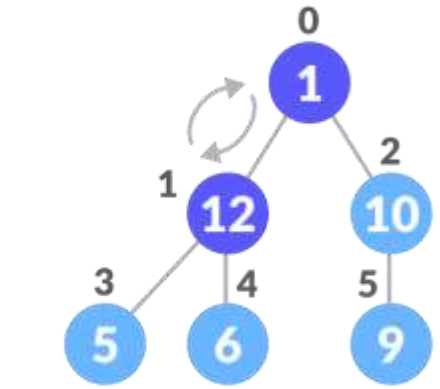


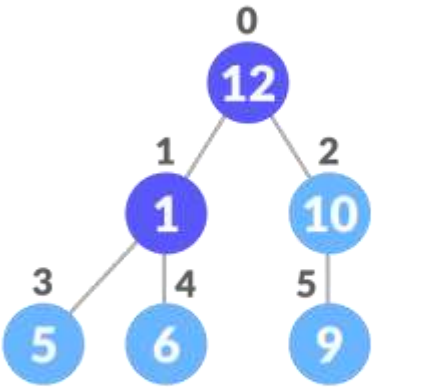
Gọi hàm heapify với nốt ở chỉ số 1

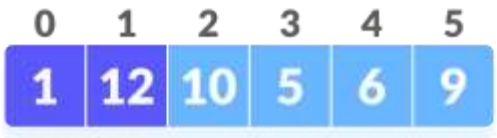


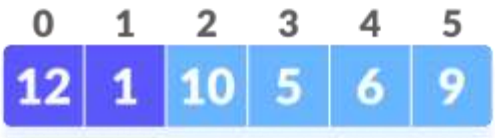
Gọi hàm heapify với nốt ở chỉ số 0

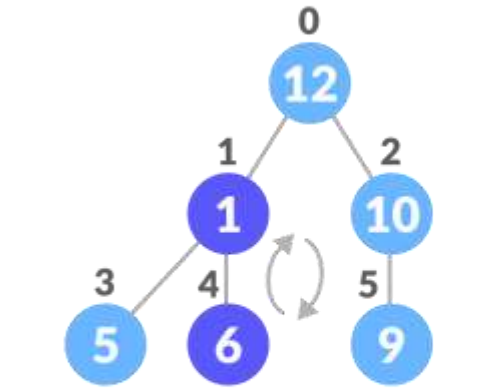
i = 0 → heapify(arr, 6, 0)

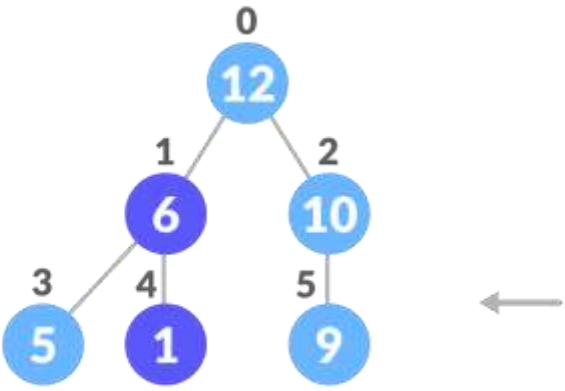


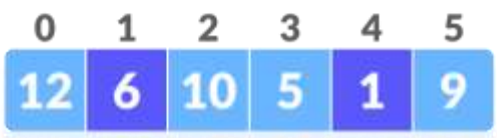


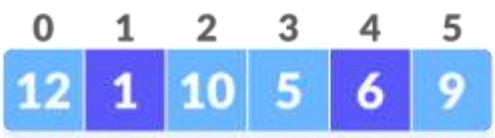












Heap sort :

Bước 1 : Vì cây thỏa mãn thuộc tính Max-Heap, nên mục lớn nhất được lưu trữ tại nút gốc.

Bước 2 : Hoán đổi: Loại bỏ phần tử gốc và đặt ở cuối mảng

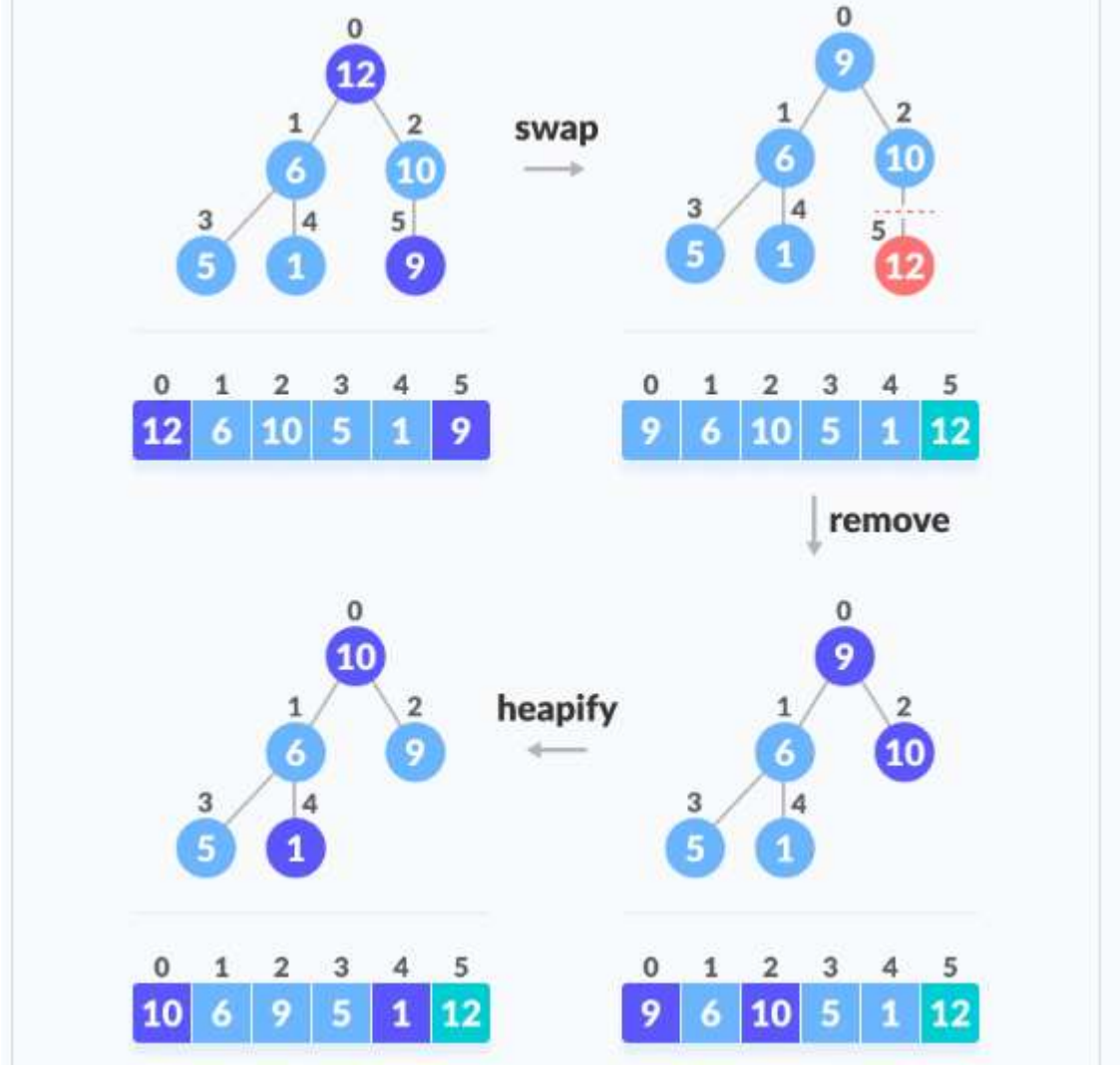
Bước 3 : Giảm kích thước của heap đi 1.

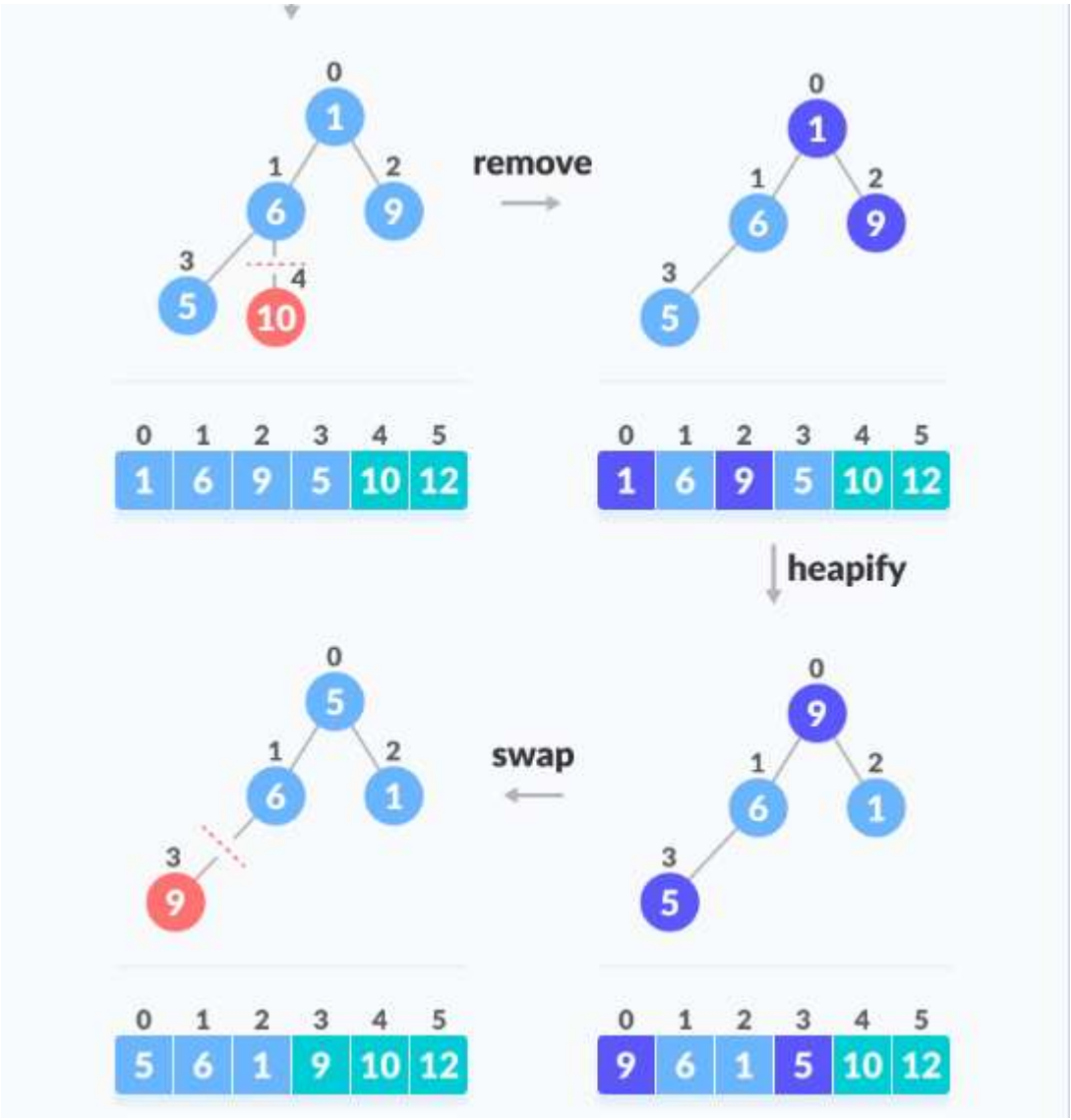
Bước 4 : Heapify: Heapify lại phần tử gốc để chúng ta có phần tử cao nhất ở gốc.

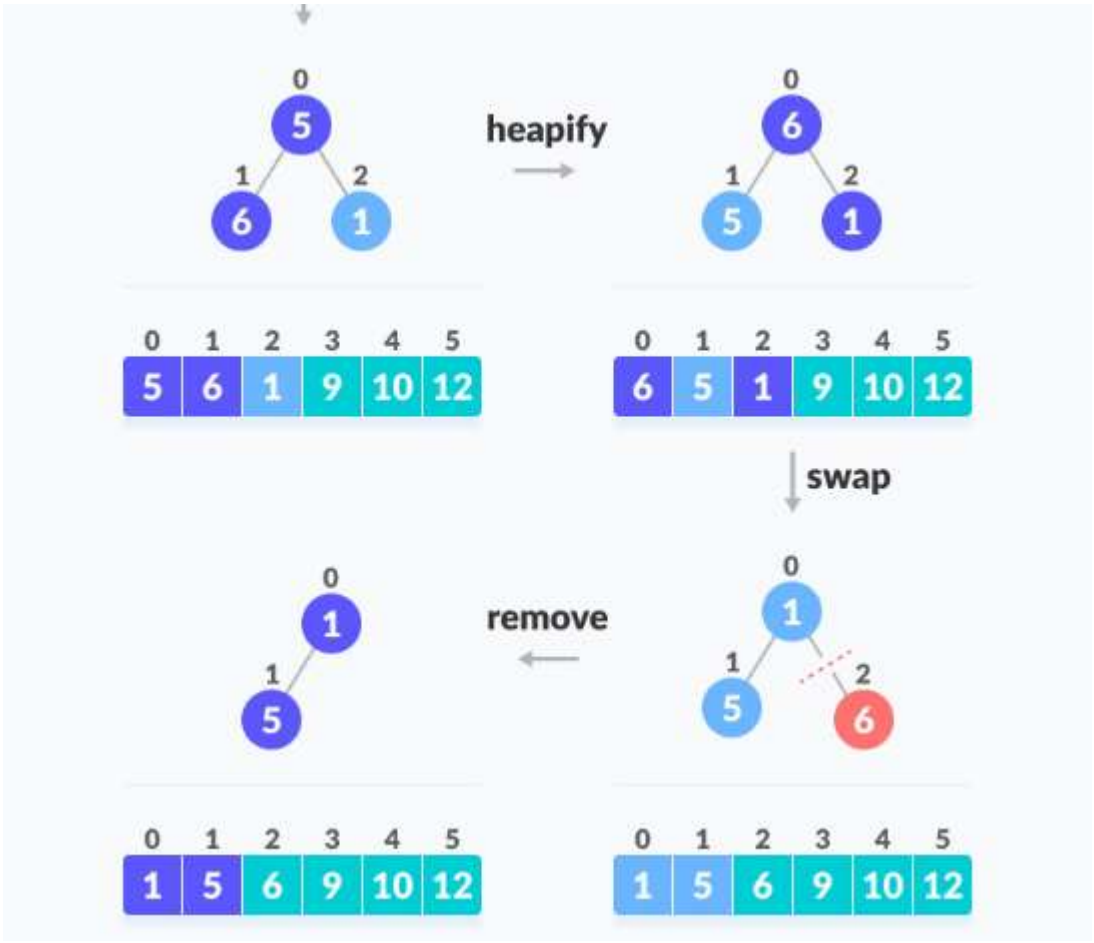
Bước 5 :Quá trình này được lặp lại cho đến khi tất cả các mục của danh sách được sắp xếp.

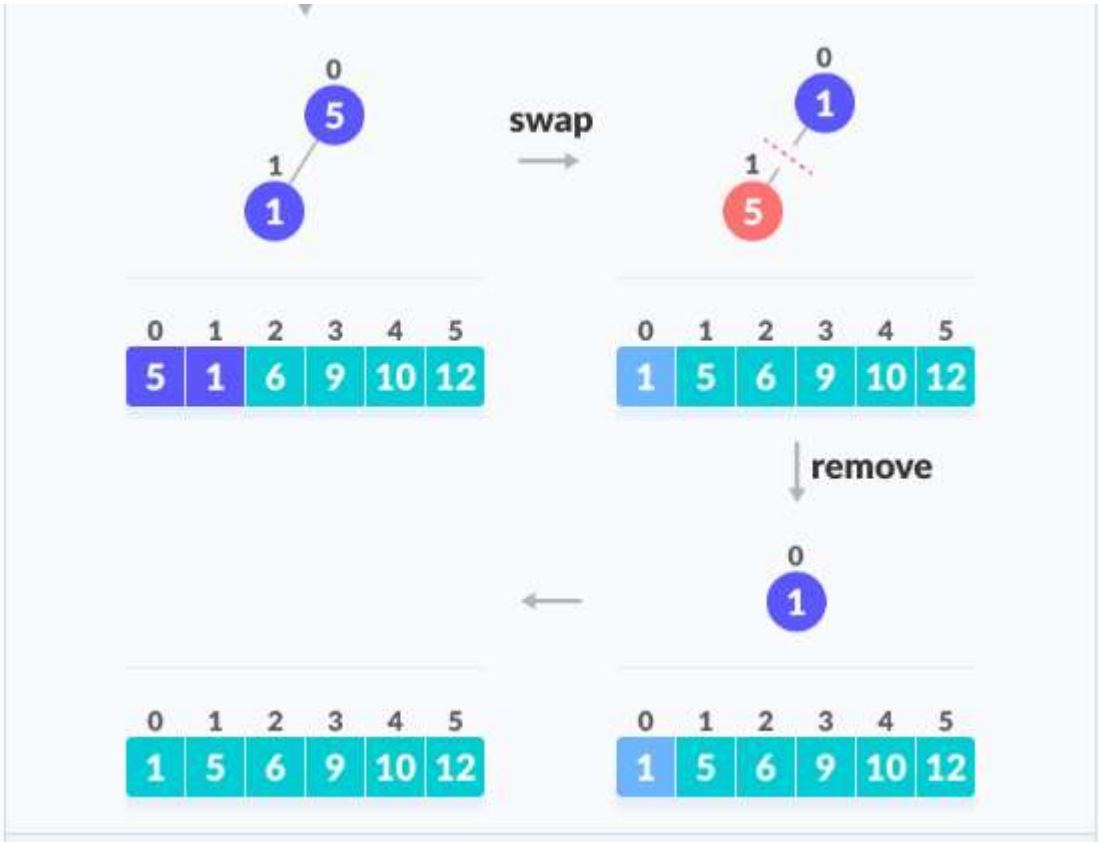
```
void heapsort(int a[], int n){
   //Build max-heap
   for (int i = n / 2 - 1; i >= 0; i--)
      heapify(arr, n, i);
   //Heap sort
   for (int i = n - 1; i >= 0; i--) {
      swap(arr[0], arr[i]);

      // Heapify root element to get highest element at root again
      heapify(arr, i, 0);
    }
}
```

Ví dụ


0
12
1
6
2
10
3
5
4
1
5
9
swap
0
9
1
6
2
10
3
5
4
1
5
12
0 1 2 3 4 5
12 6 10 5 1 9
0 1 2 3 4 5
9 6 10 5 1 12
remove
0
10
1
6
2
9
3
5
4
1
heapify
0
9
1
6
2
10
3
5
4
1
0 1 2 3 4 5
10 6 9 5 1 12
0 1 2 3 4 5
9 6 10 5 1 12


remove
heapify
swap
0 1 2 3 4 5
1 6 9 5 10 12
0 1 2 3 4 5
1 6 9 5 10 12
0 1 2 3 4 5
5 6 1 9 10 12
0 1 2 3 4 5
9 6 1 5 10 12


heapify
swap
remove
0 1 2 3 4 5
5 6 1 9 10 12
6 5 1 9 10 12
1 5 6 9 10 12


swap
remove
0 1 2 3 4 5
5 1 6 9 10 12
1 5 6 9 10 12

7. Thuật toán Quicksort

Độ phức tạp : O(nlogn)

Tư tưởng:

Divide: Partition (rearrange) the array $A[p..r]$ into two (possibly empty) subarrays $A[p..q-1]$ and $A[q+1..r]$ such that each element of $A[p..q-1]$ is less than or equal to $A[q]$, which is, in turn, less than or equal to each element of $A[q+1..r]$. Compute the index q as part of this partitioning procedure.

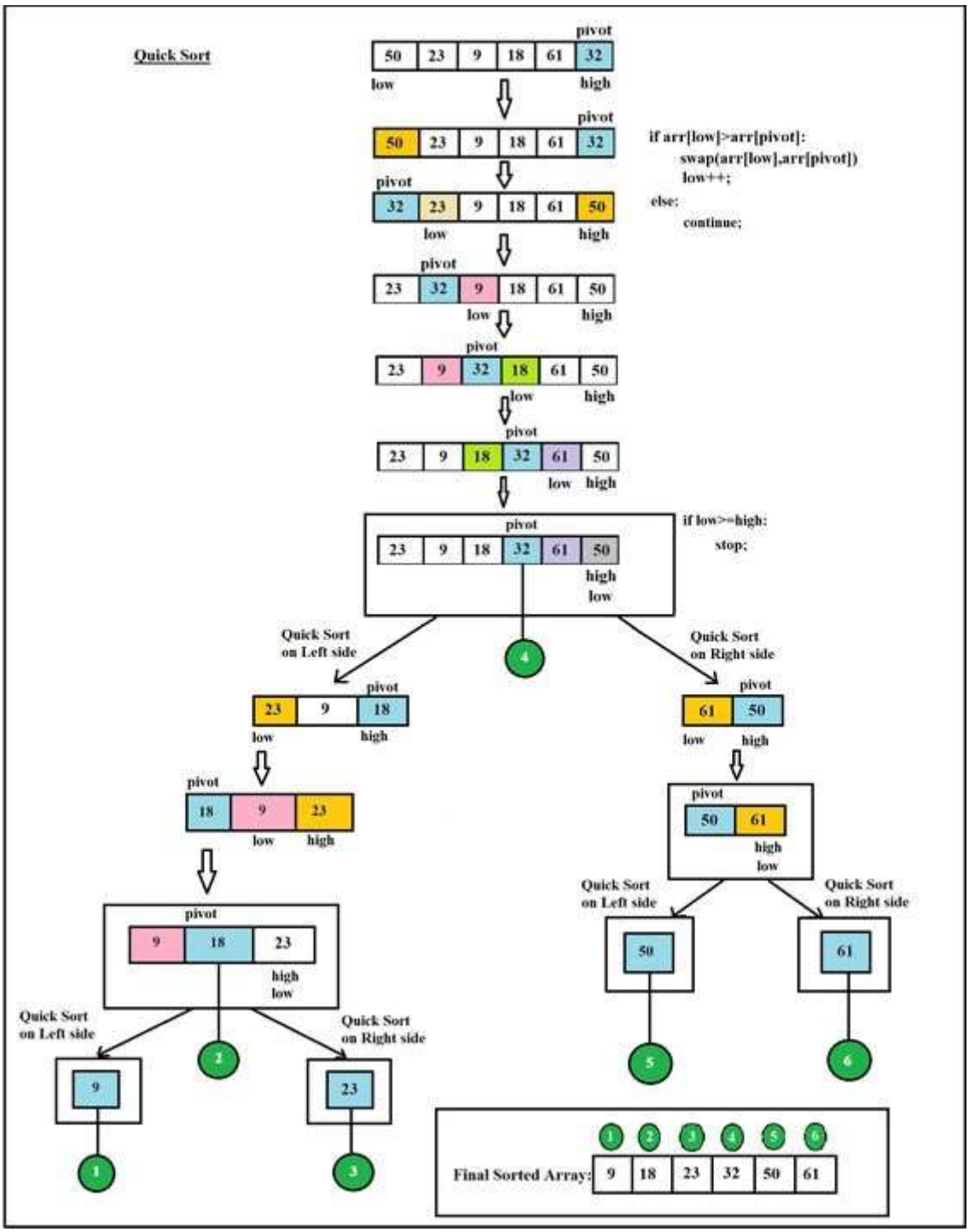
Conquer: Sort the two subarrays $A[p..q-1]$ and $A[q+1..r]$ by recursive calls to quicksort.

Combine: Because the subarrays are already sorted, no work is needed to combine them: the entire array $A[p..r]$ is now sorted.

Pseudocode

The following procedure implements quicksort:

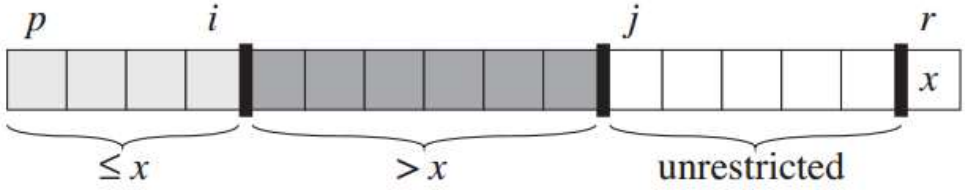
```
QUICKSORT(A, p, r)
1  if p < r
2      q = PARTITION(A, p, r)
3      QUICKSORT(A, p, q − 1)
4      QUICKSORT(A, q + 1, r)
```

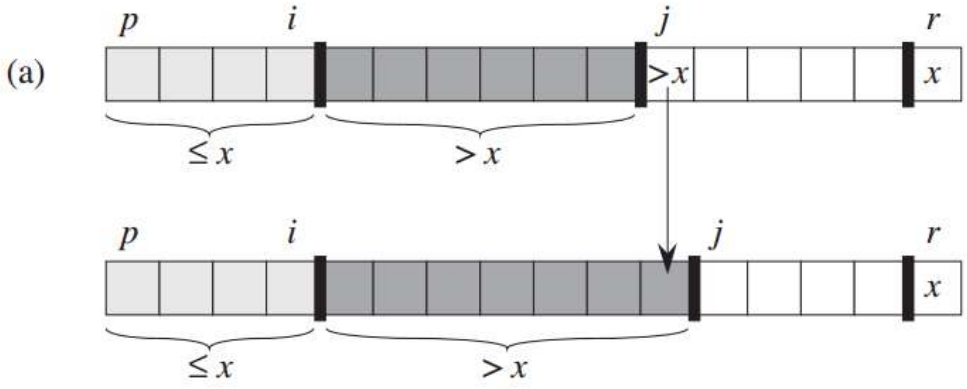


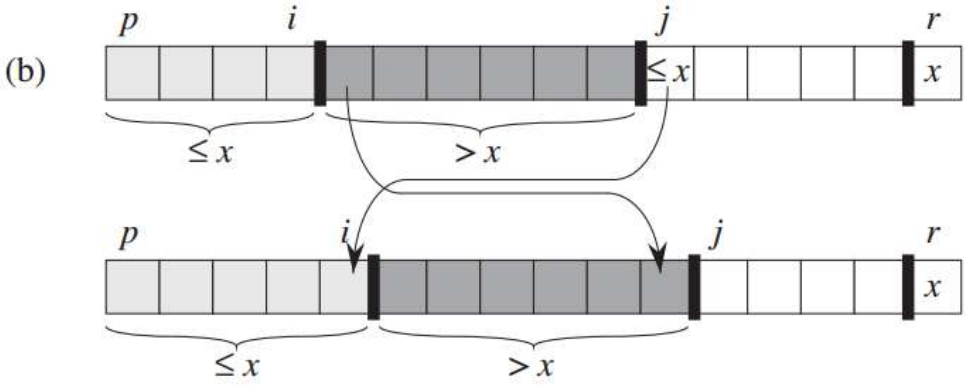
Lomuto partition (CLRS).

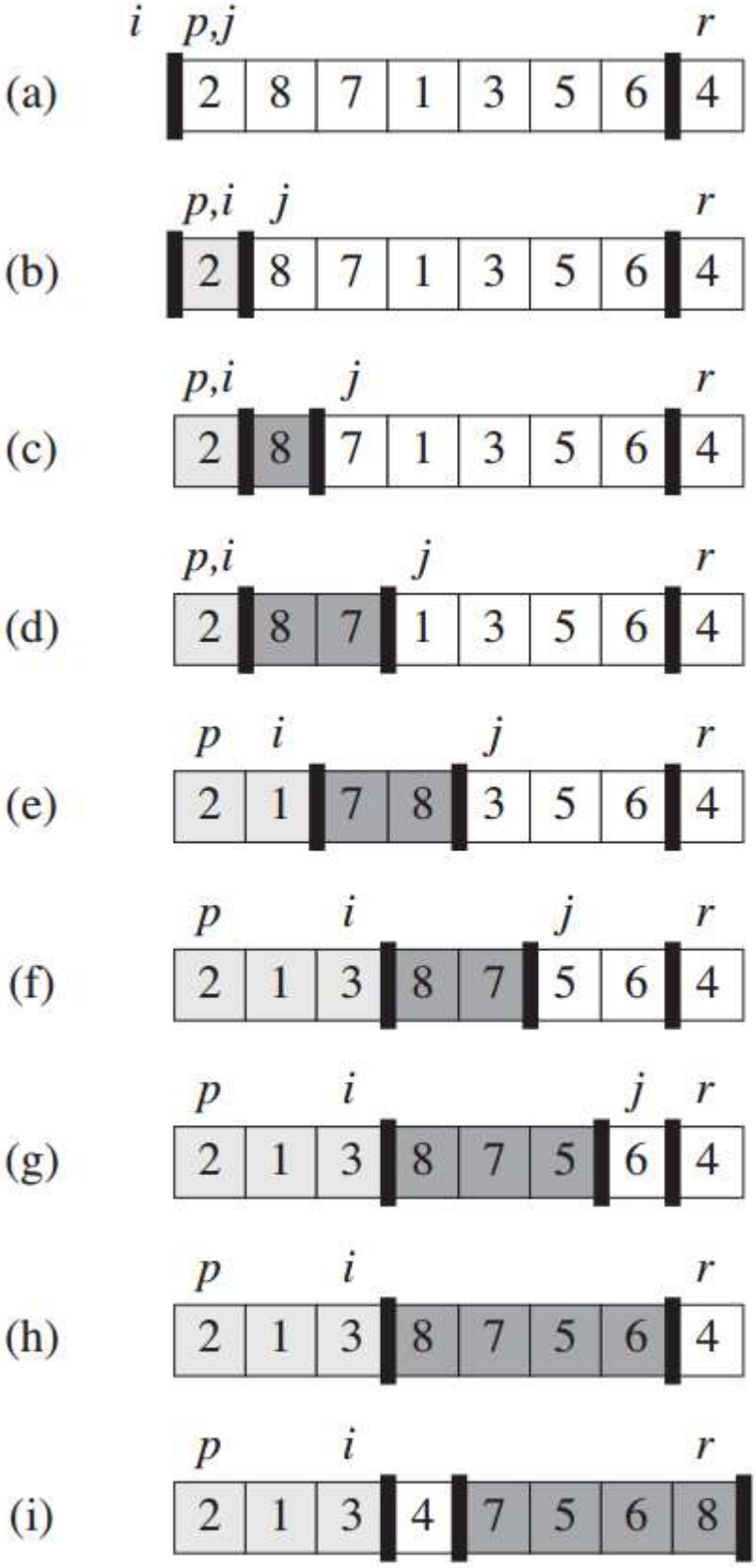
```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

Ví dụ về Lomuto Partition








i p,j r
(a) 2 8 7 1 3 5 6 4
p,i j r
(b) 2 8 7 1 3 5 6 4
p,i j r
(c) 2 8 7 1 3 5 6 4
p,i j r
(d) 2 8 7 1 3 5 6 4
p i j r
(e) 2 1 7 8 3 5 6 4
p i j r
(f) 2 1 3 8 7 5 6 4
p i j r
(g) 2 1 3 8 7 5 6 4
p i r
(h) 2 1 3 8 7 5 6 4
p i r
(i) 2 1 3 4 7 5 6 8

Hoare Partition

```
HOARE-PARTITION(A, p, r)
 1  x = A[p]
 2  i = p − 1
 3  j = r + 1
 4  while TRUE
 5       repeat
 6            j = j − 1
 7       until A[j] ≤ x
 8       repeat
 9            i = i + 1
10       until A[i] ≥ x
11       if i < j
12            exchange A[i] with A[j]
13       else return j
```