SẮP XẾP (SORTING)

Nội dung

- Tổng quan
- Các phương pháp sắp xếp thông dụng

Tổng quan

- Tại sao phải sắp xếp?
  - Để có thể sử dụng thuật toán tìm nhị phân
  - Để thực hiện thao tác nào đó được nhanh hơn
- Định nghĩa bài toán sắp xếp
  - Sắp xếp là quá trình xử lý một danh sách các phần tử (hoặc các mẫu tin) để đặt chúng theo một thứ tự thỏa mãn một tiêu chuẩn nào đó dựa trên nội dung thông tin lưu giữ tại mỗi phần tử

Các phương pháp sắp xếp thông dụng

- Đổi chỗ trực tiếp (Interchange sort)
- Nổi bọt (Bubble sort)
- Chèn trực tiếp (Insertion sort)
- Chọn trực tiếp (Selection sort)
- Sắp xếp nhanh (Quick sort)
- Sắp xếp trộn (Merge sort)
- Vun đống (Heap sort)

Interchange Sort

- Khái niệm nghịch thế:
  - Xét một mảng các số a[0], a[1], … a[n-1]
  - Nếu có i<j và a[i] > a[j], thì ta gọi đó là một nghịch thế
- Mảng chưa sắp xếp sẽ có nghịch thế
- Mảng đã có thứ tự sẽ không chứa nghịch thế

$$a[0] \leq a[1] \leq \ldots \leq a[n-1]$$

Interchange Sort – Ý tưởng

- Nhận xét:
  - Để sắp xếp một dãy số, ta có thể xét các nghịch thế có trong dãy và làm triệt tiêu dần chúng đi
- Ý tưởng:
  - Xuất phát từ đầu dãy, tìm tất cả nghịch thế chứa phần tử này, triệt tiêu chúng bằng cách đổi chỗ phần tử này với phần tử tương ứng trong cặp nghịch thế
  - Lặp lại xử lý trên với các phần tử tiếp theo trong dãy

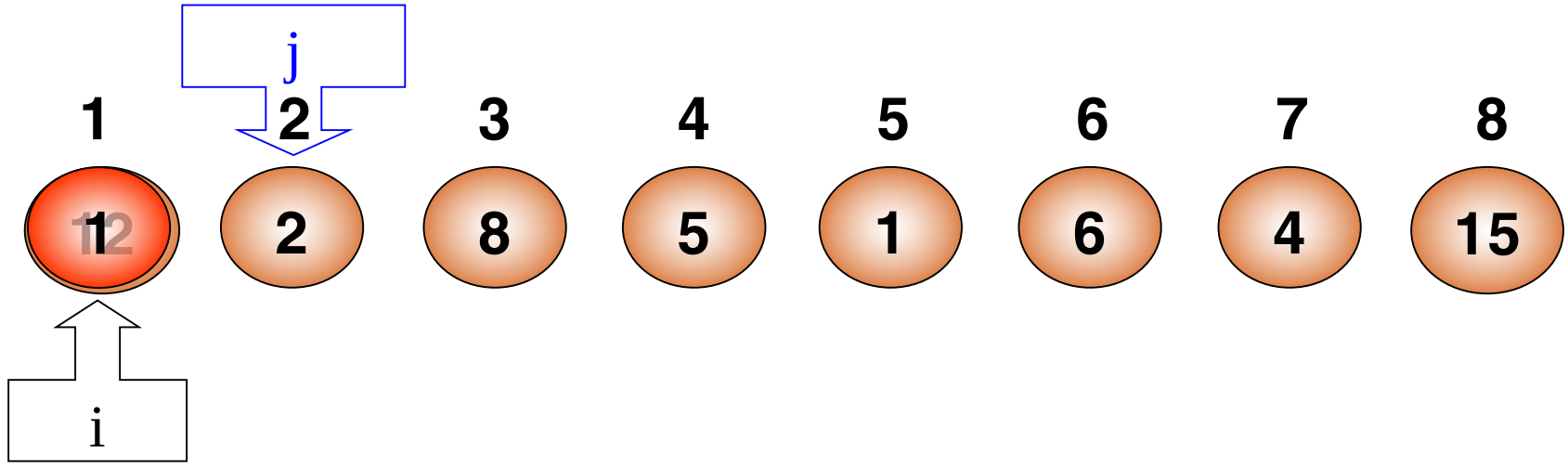
Interchange Sort – Thuật toán

// input: dãy (a, n)

// output: dãy (a, n) đã được sắp xếp

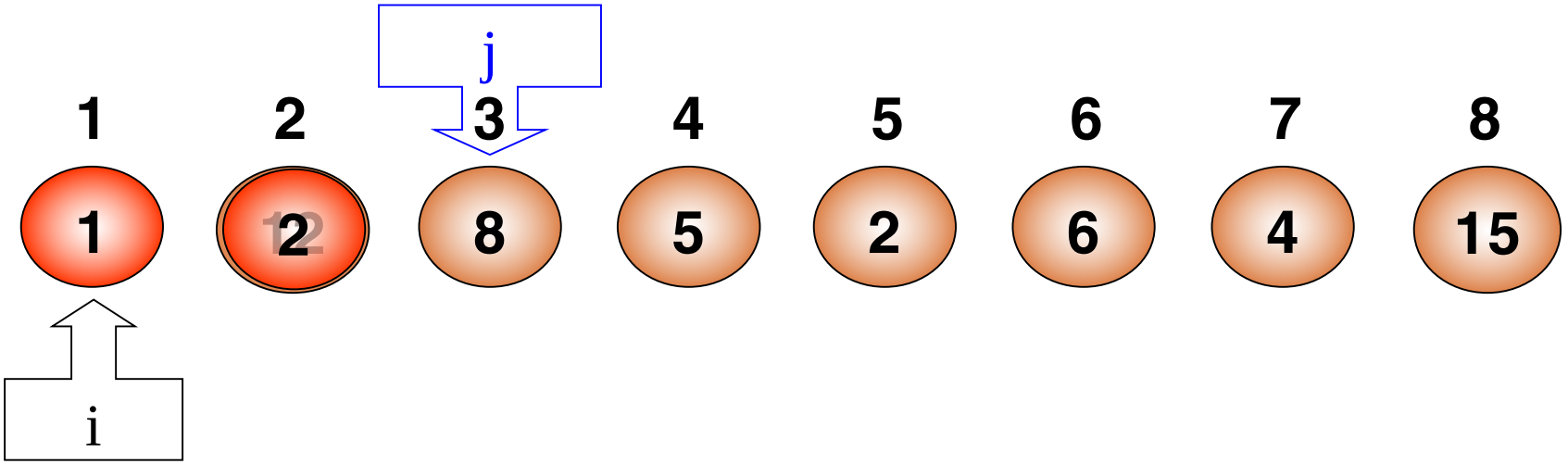
- Bước 1: i = 0; *// bắt đầu từ đầu dãy*
- Bước 2: j = i+1;
- Bước 3: Trong khi j < n thực hiện:
  - Nếu a[i]>a[j] thì đổi chỗ a[i], a[j]
  - j = j+1;
- Bước 4: i = i+1;
  - Nếu (i < n-1): Lặp lại Bước 2
  - Ngược lại: Dừng

Interchange Sort – Ví dụ



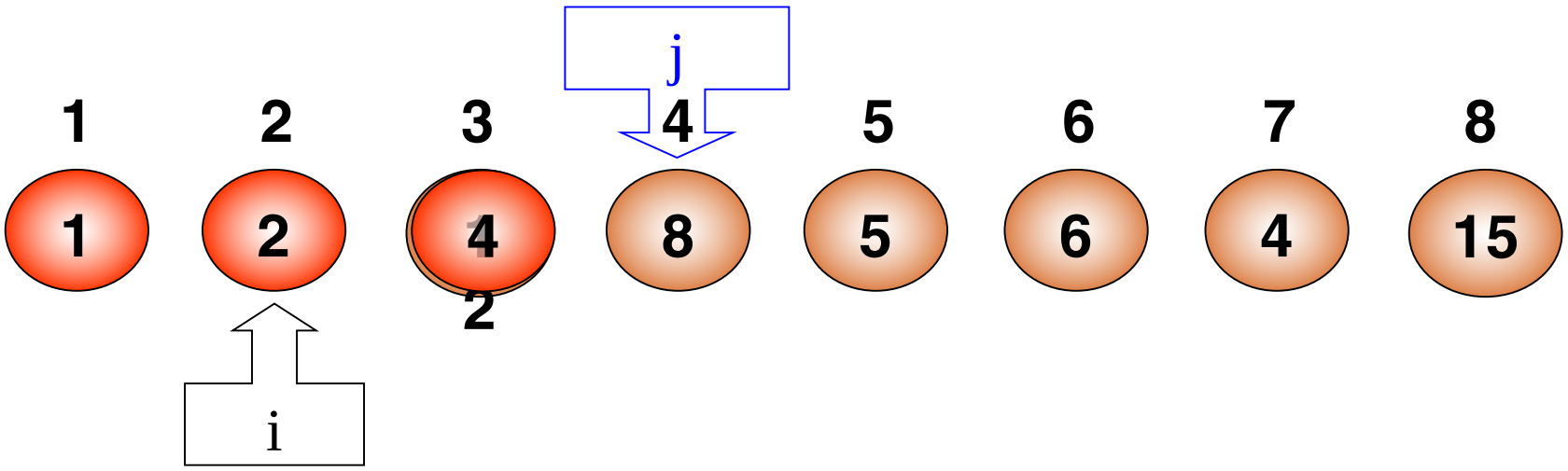
Nếu a[i] > a[j] thì đổi chỗ a[i], a[j]

Interchange Sort – Ví dụ



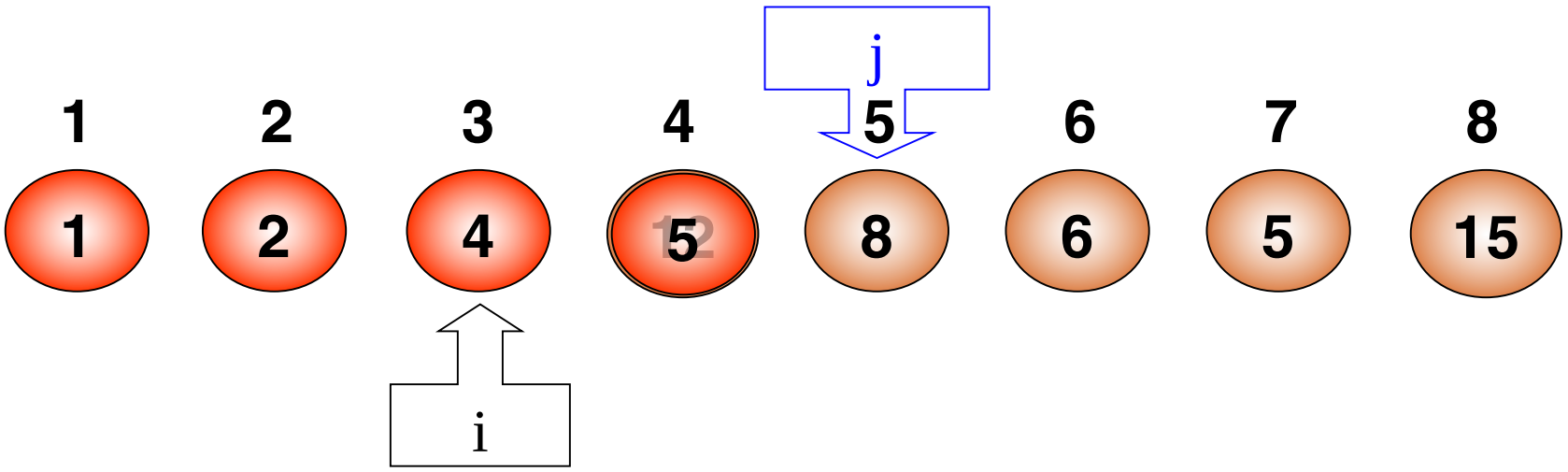
Nếu a[i] > a[j] thì đổi chỗ a[i], a[j]

Interchange Sort – Ví dụ



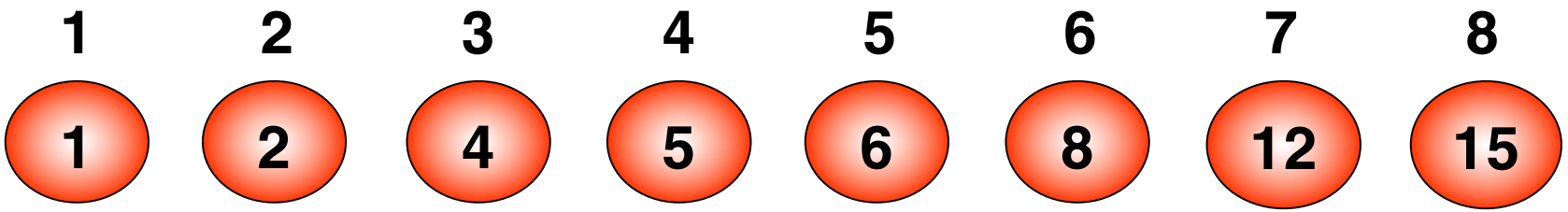
Nếu a[i] > a[j] thì đổi chỗ a[i], a[j]

Interchange Sort – Ví dụ



Nếu a[i] > a[j] thì đổi chỗ a[i], a[j]

Interchange Sort – Ví dụ



Nếu a[i] > a[j] thì đổi chỗ a[i], a[j]

Interchange Sort - Cài đặt


```
void InterchangeSort(int a[], int n)
{
  for (int i=0 ; i<n-1 ; i++)
      for (int j=i+1; j < n ; j++)
         if(a[i]>a[j])
           Swap(a[i], a[j]);
}
void Swap(int &a, int &b)
{
      int  temp = a;
             a = b;
             b = temp;
}
```

Interchange Sort - Đánh giá giải thuật

- Số lượng các phép so sánh xảy ra không phụ thuộc vào tình trạng của dãy số ban đầu
- Số lượng phép hoán vị thực hiện tùy thuộc vào kết quả so sánh

Trường hợp	Số lần so sánh	Số lần hoán vị
Tốt nhất	$\sum_{i=1}^{n-1}(n-i+1)=\frac{n(n-1)}{2}$	0
Xấu nhất	$\frac{n(n-1)}{2}$	$\sum_{i=1}^{n-1}(n-i+1)=\frac{n(n-1)}{2}$

Bubble Sort – Ý tưởng

- Xuất phát từ cuối (đầu) dãy, đổi chỗ các cặp phần tử kế cận để đưa phần tử nhỏ (lớn) hơn trong cặp phần tử đó về vị trí đúng đầu (cuối) dãy hiện hành, sau đó sẽ không xét đến nó ở bước tiếp theo
- Ở lần xử lý thứ i có vị trí đầu dãy là i
- Lặp lại xử lý trên cho đến khi không còn cặp phần tử nào để xét

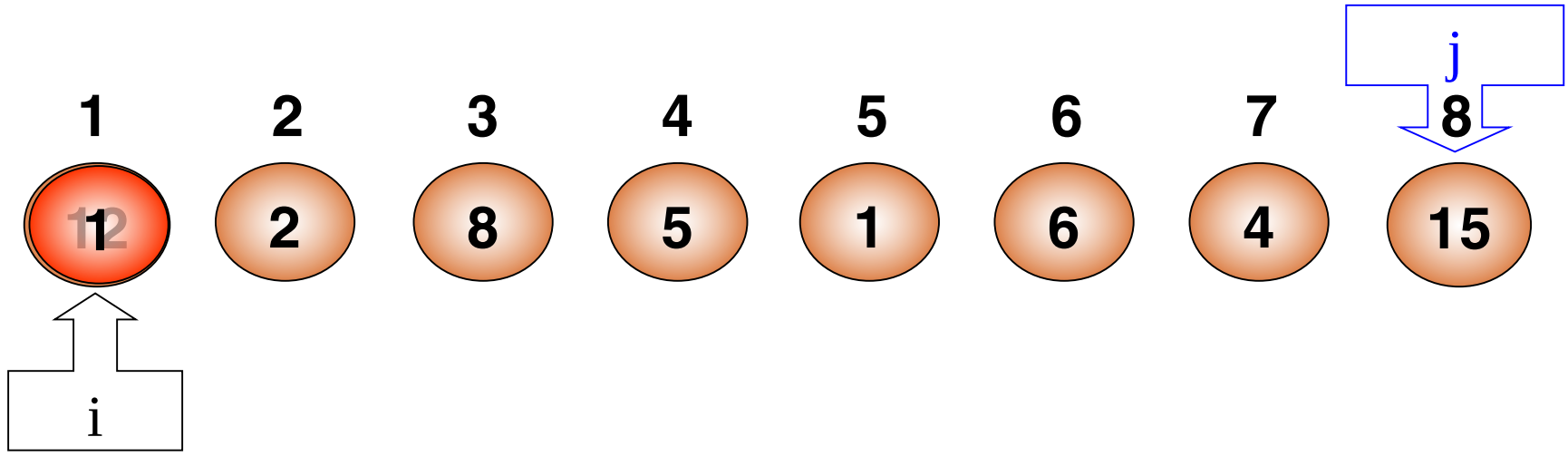
Bubble Sort – Thuật toán

// input: dãy (a, n)

// output: dãy (a, n) đã được sắp xếp

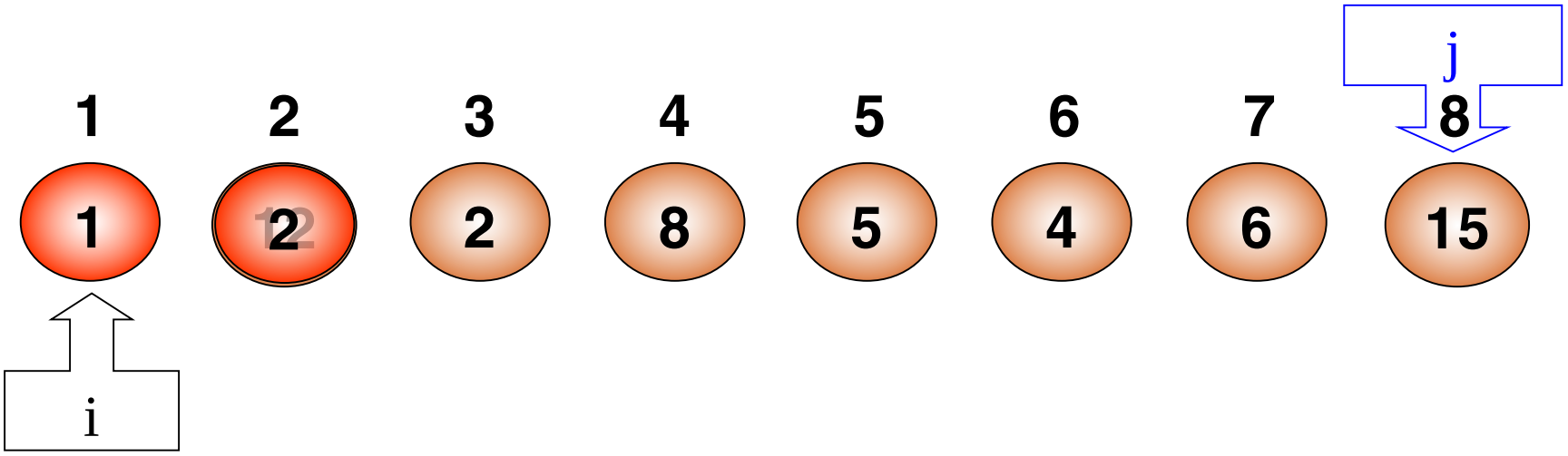
- Bước 1: i = 0;
- Bước 2: j = n-1; *//Duyệt từ cuối dãy ngược về vị trí i*
  - Trong khi (j > i) thực hiện:
    - Nếu a[j]<a[j-1] thì đổi chỗ a[j], a[j-1]
    - j = j-1;
- Bước 3: i = i+1; *// lần xử lý kế tiếp*
  - Nếu i = n: Dừng *// Hết dãy*
  - Ngược lại: Lặp lại Bước 2

Bubble Sort – Ví dụ



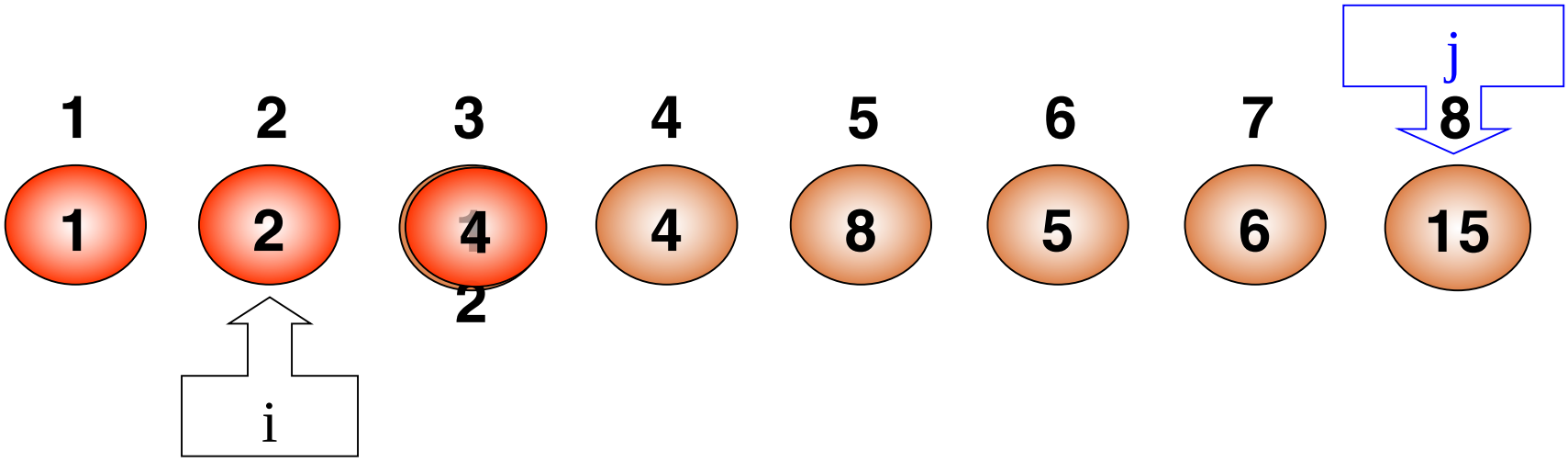
Nếu a[j]<a[j-1] thì đổi chỗ a[j], a[j-1]

Bubble Sort – Ví dụ



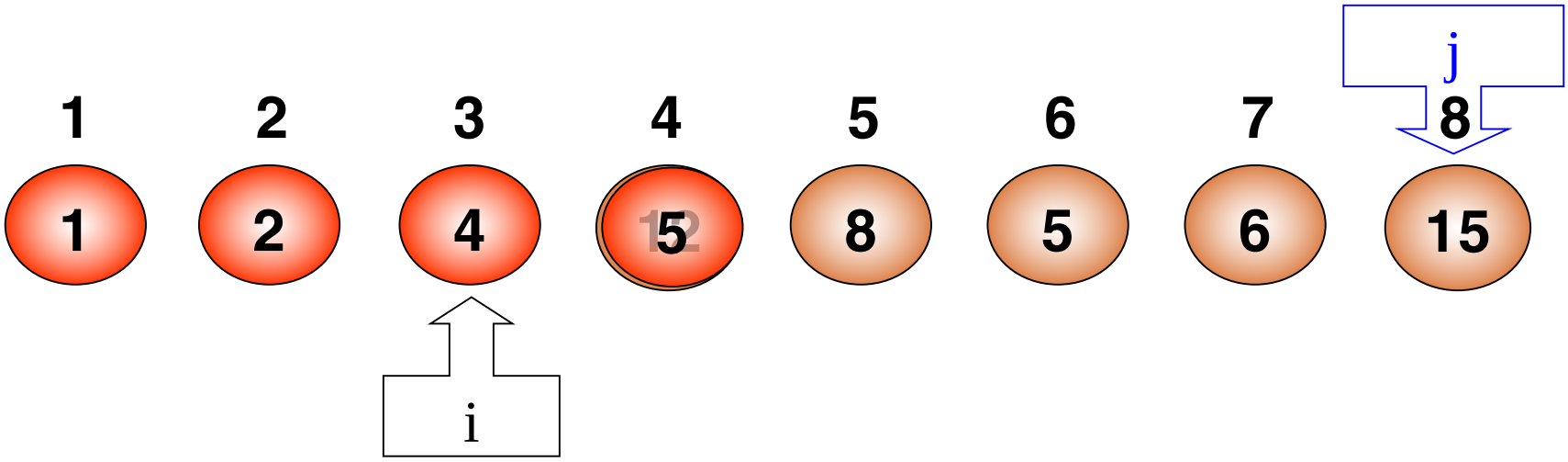
Nếu a[j]<a[j-1] thì đổi chỗ a[j], a[j-1]

Bubble Sort – Ví dụ



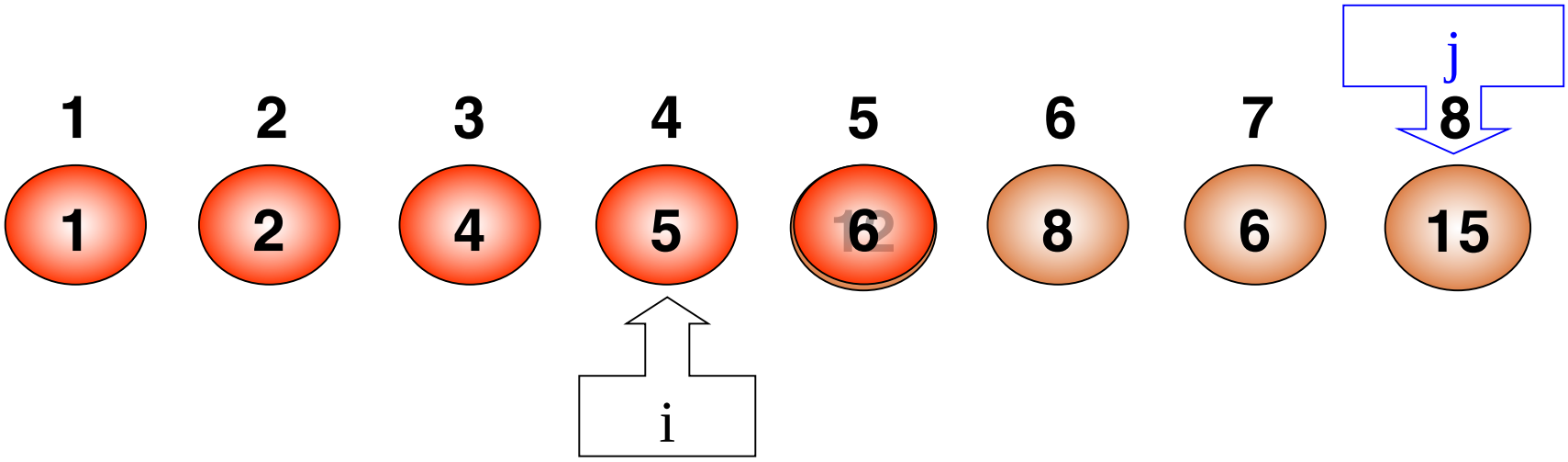
Nếu a[j]<a[j-1] thì đổi chỗ a[j], a[j-1]

Bubble Sort – Ví dụ



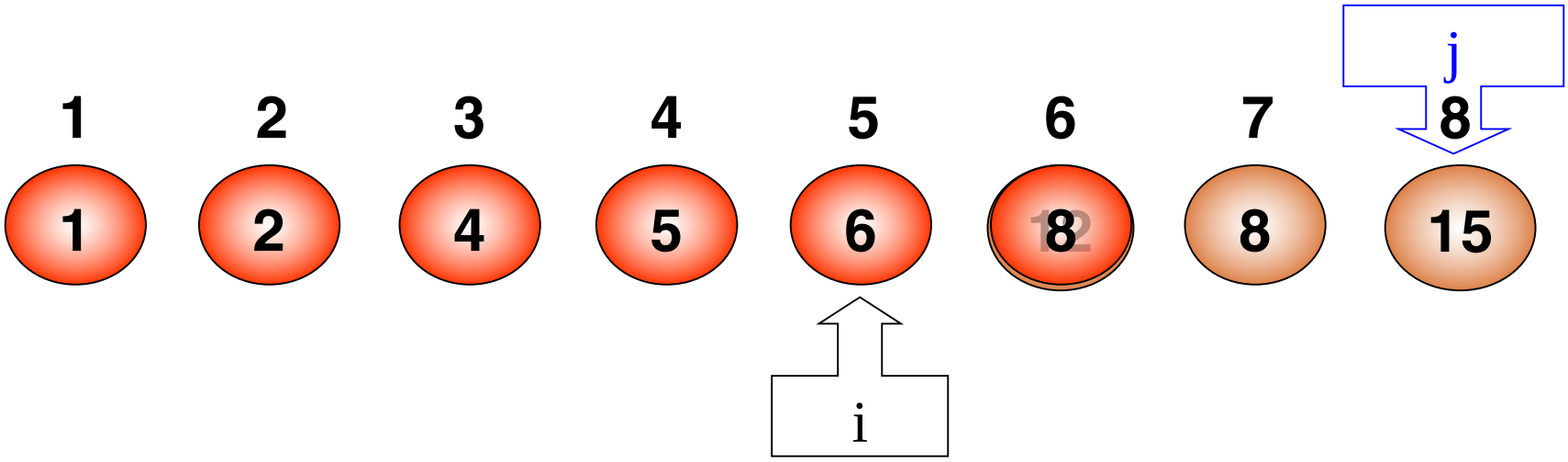
Nếu a[j]<a[j-1] thì đổi chỗ a[j], a[j-1]

Bubble Sort – Ví dụ



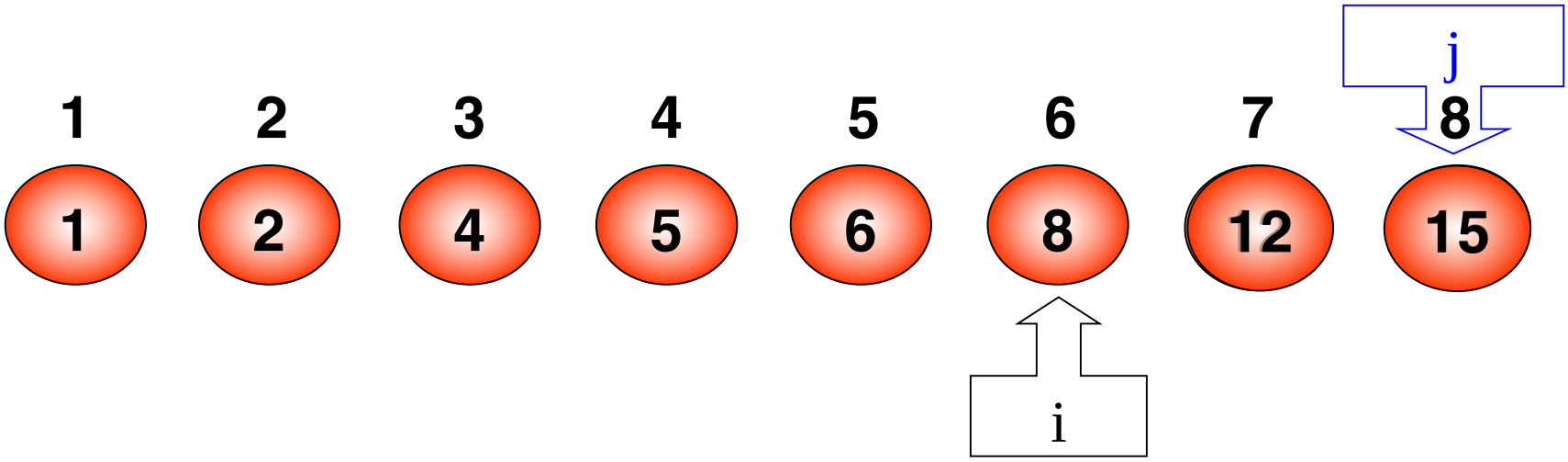
Nếu a[j]<a[j-1] thì đổi chỗ a[j], a[j-1]

Bubble Sort – Ví dụ



Nếu a[j]<a[j-1] thì đổi chỗ a[j], a[j-1]

Bubble Sort – Ví dụ



Nếu a[j]<a[j-1] thì đổi chỗ a[j], a[j-1]

Bubble Sort - Cài đặt


```
void BubbleSort(int a[], int n)
{
  for (int i=0; i<n-1; i++)
      for (int j=n-1; j>i; j--)
         if(a[j] < a[j-1])
            Swap(a[j], a[j-1]);
}
```

Bubble Sort - Đánh giá giải thuật

- Số lượng các phép so sánh xảy ra không phụ thuộc vào tình trạng của dãy số ban đầu
- Số lượng phép hoán vị thực hiện tùy thuộc vào kết quả so sánh

Trường hợp	Số lần so sánh	Số lần hoán vị
Tốt nhất	$\sum_{i=1}^{n-1}(n-i+1) = \frac{n(n-1)}{2}$	0
Xấu nhất	$\frac{n(n-1)}{2}$	$\sum_{i=1}^{n-1}(n-i+1) = \frac{n(n-1)}{2}$

Bubble Sort - Đánh giá giải thuật

- Khuyết điểm:
  - Không nhận diện được tình trạng dãy đã có thứ tự hay có thứ tự từng phần (nếu chỉ có 1 cặp chưa đúng vị trí vẫn phải chạy hết vòng lặp)
  - Các phần tử nhỏ được đưa về vị trí đúng rất nhanh, trong khi các phần tử lớn lại được đưa về vị trí đúng rất chậm

Insertion Sort – Ý tưởng

- Nhận xét:
  - Mọi dãy **a[0] , a[1] ,..., a[n-1]** luôn có i-1 phần tử đầu tiên **a[0] , a[1] ,... ,a[i-2]** đã có thứ tự ($2 \leq i$)
- Ý tưởng chính:
  - Tìm cách chèn phần tử **ai** vào vị trí thích hợp của đoạn đã được sắp để có dãy mới **a[0] , a[1] ,... ,a[i-1]** trở nên có thứ tự
  - Vị trí này chính là pos thỏa :

$$\mathbf{a[pos-1] \leq a[i] < a[pos]} \quad (1 \leq pos \leq i)$$

Insertion Sort – Ý tưởng

Chi tiết hơn:

- *Dãy ban đầu* ***a[0] , a[1] ,..., a[n-1]****, xem như đã có đoạn gồm một phần tử* ***a[0]*** *đã được sắp*
- *Thêm* ***a[1]*** *vào đoạn* ***a[0]*** *sẽ có đoạn* ***a[0] a[1]*** *được sắp*
- *Thêm* ***a[2]*** *vào đoạn* ***a[0] a[1]*** *để có đoạn* ***a[0] a[1] a[2]*** *được sắp*
- *Tiếp tục cho đến khi thêm xong* ***a[n-1]*** *vào đoạn* ***a[0] a[1] …a[n-1]*** *sẽ có dãy* ***a[0] a[1]…… A[n-1]*** *được sắp*

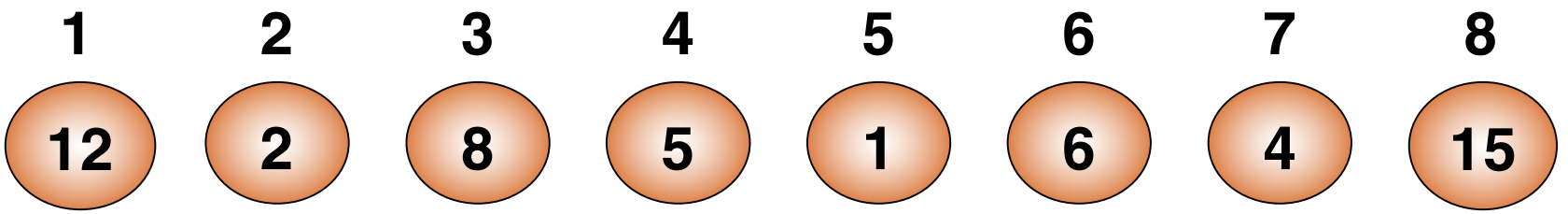
Insertion Sort – Thuật toán

// input: dãy (a, n)

// output: dãy (a, n) đã được sắp xếp

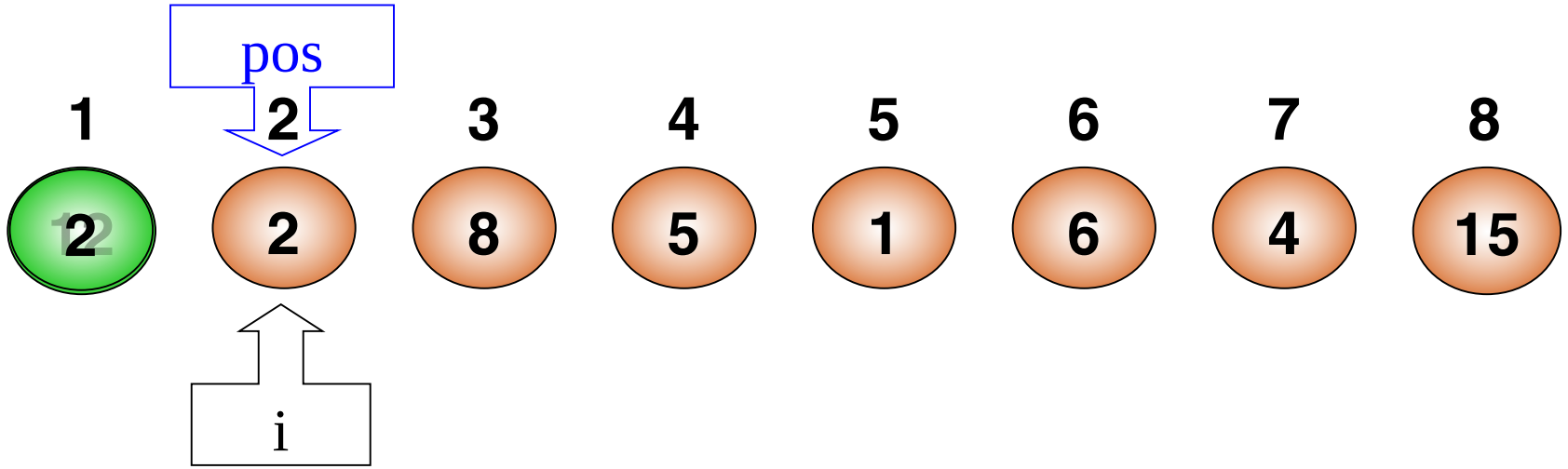
- Bước 1: i = 2; *// giả sử có đoạn* a[0] *đã được sắp*
- Bước 2: x = a[i]; //Tìm vị trí pos thích hợp trong đoạn a[0]
  //đến a[i] để chèn x vào
- Bước 3: Dời chỗ các phần tử từ a[pos] đến a[i-1] sang phải 1 vị trí để dành chỗ cho x
- Bước 4: a[pos] = x; *// có đoạn* a[0]..a[i] *đã được sắp*
- Bước 5: i = i+1;
  Nếu i ≤ n: Lặp lại Bước 2
  Ngược lại: Dừng

Insertion Sort – Ví dụ



Insertion Sort – Ví dụ

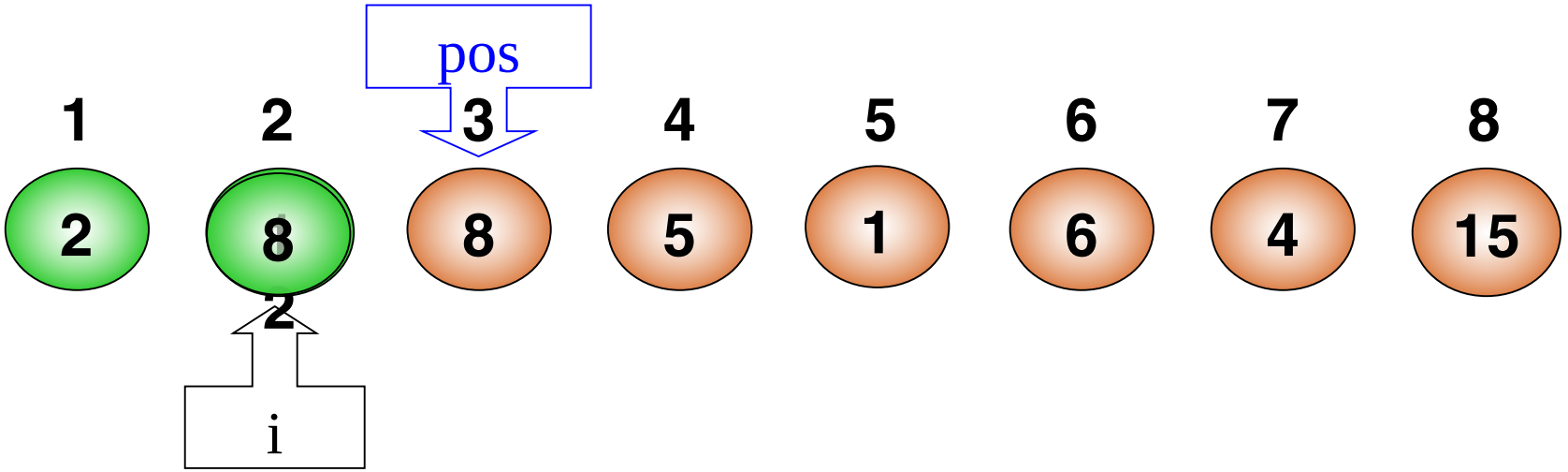
Chèn a[1] vào (a[0], a[1])



X

Insertion Sort – Ví dụ

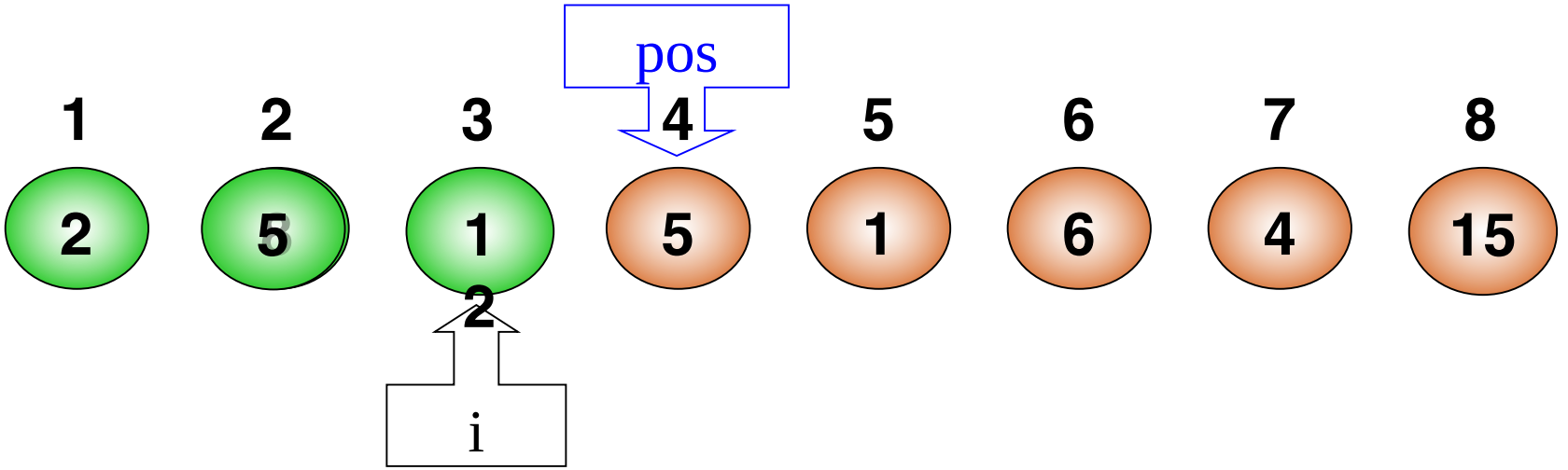
Chèn a[2] vào (a[0] … a[2])



X

Insertion Sort – Ví dụ

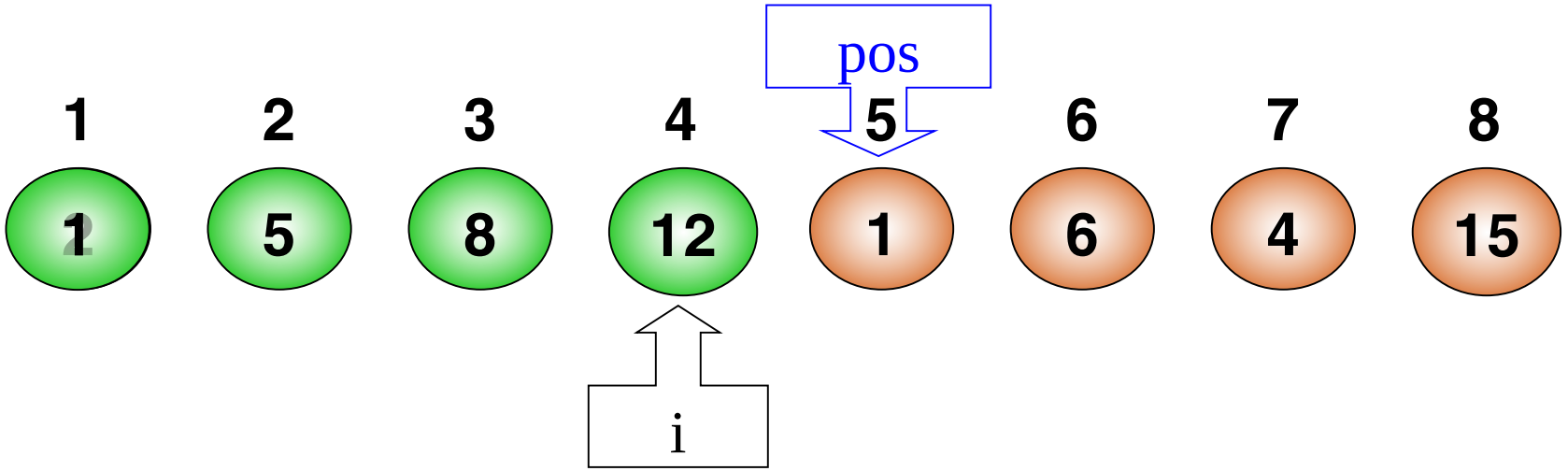
Chèn a[3] vào (a[0] … a[3])



X

Insertion Sort – Ví dụ

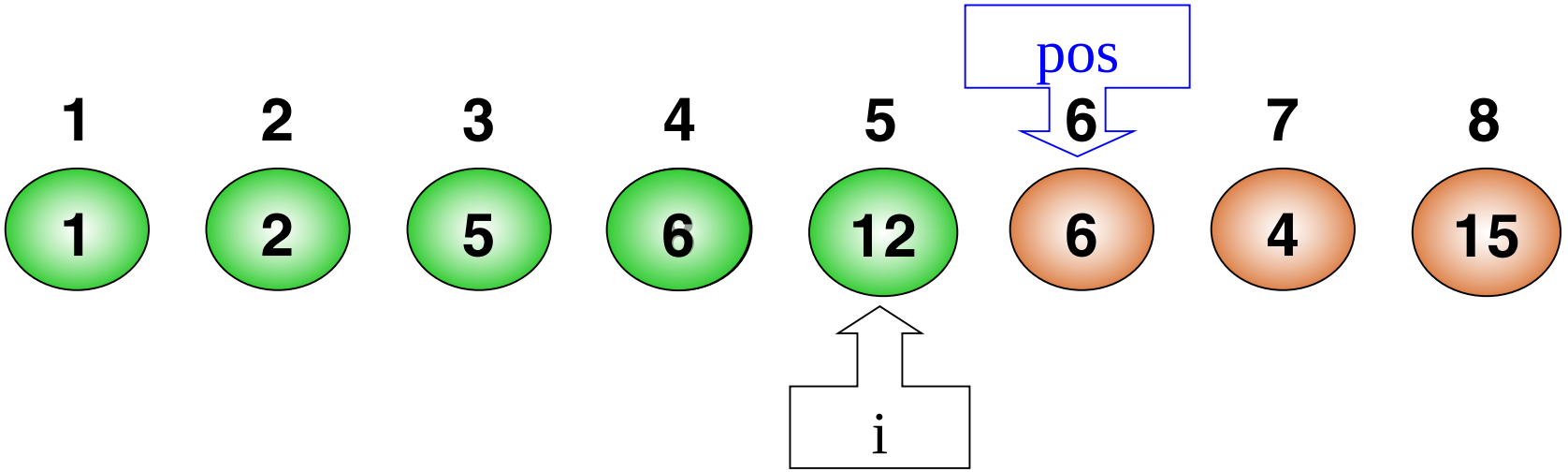
Chèn a[4] vào (a[0] … a[4])



X

Insertion Sort – Ví dụ

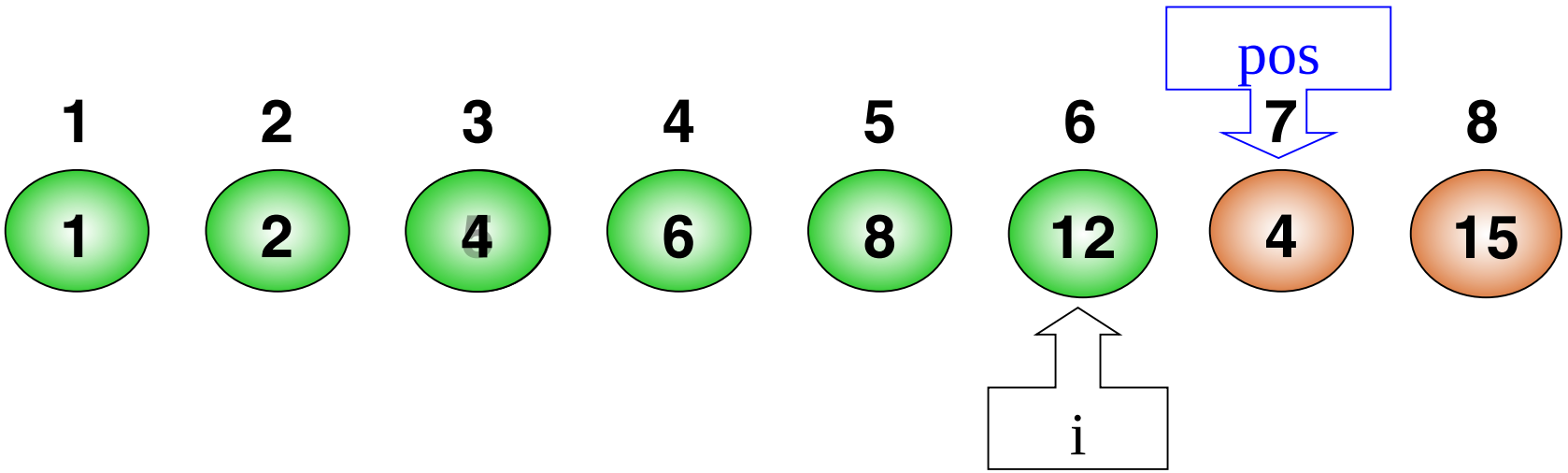
Chèn a[5] vào (a[0]… a[5])



X

Insertion Sort – Ví dụ

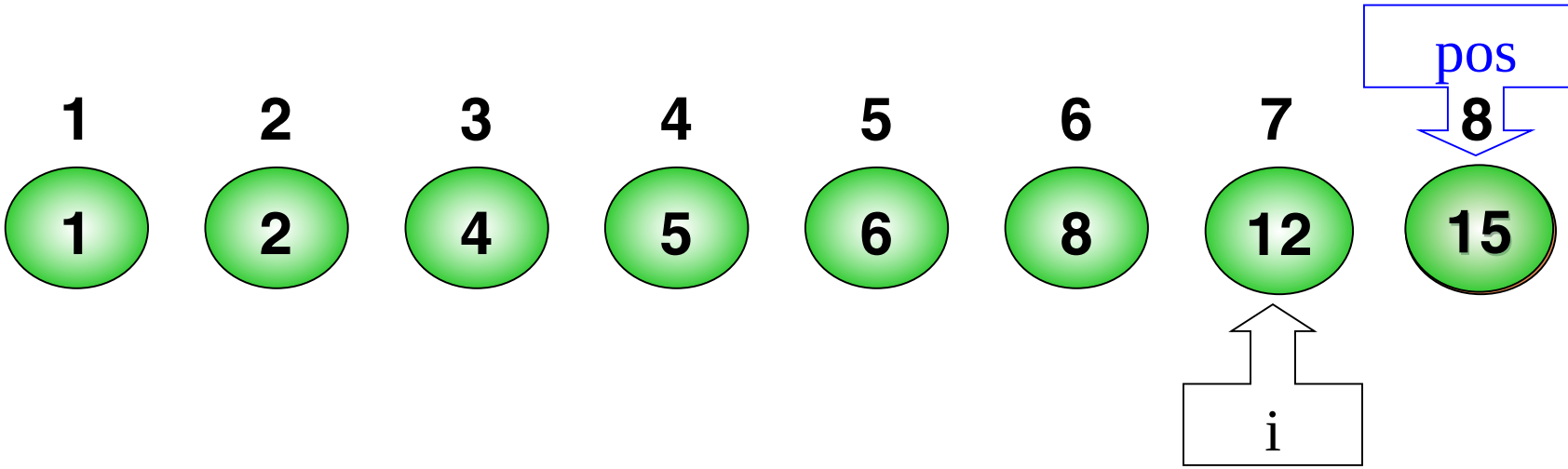
Chèn a[6] vào (a[0] … a[6])



X

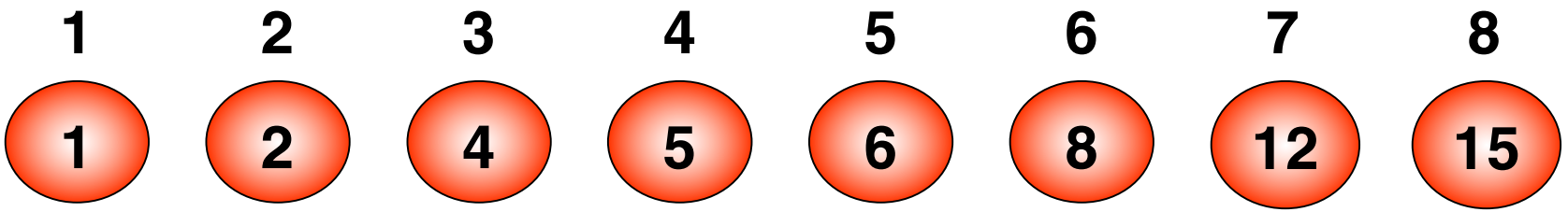
Insertion Sort – Ví dụ

Chèn a[7] vào (a[0] … a[7])



X

Insertion Sort – Ví dụ



Insertion Sort – Cài đặt


```
void InsertionSort(int a[], int n){
  int pos, x;
  for(int i=0; i<n; i++) //đoạn a[0] đã sắp
  {
      x = a[i+1];
      pos = i;
      while(pos>=0 && a[pos]>x)
      {     a[pos+1] = a[pos];
            pos--;
      }
      a[pos+1] = x;
  }
}
```

Insertion Sort – Nhận xét

- Khi tìm vị trí thích hợp để chèn a[i] vào đoạn a[0] đến a[i-1], do đoạn đã được sắp nên có thể sử dụng giải thuật tìm nhị phân để thực hiện việc tìm vị trí pos

➔ giải thuật sắp xếp chèn nhị phân *Binary Insertion Sort*

- *Lưu ý:* Chèn nhị phân chỉ làm giảm số lần so sánh, không làm giảm số lần dời chỗ

Insertion Sort – Đánh giá giải thuật

- Các phép so sánh xảy ra trong mỗi vòng lặp tìm vị trí thích hợp pos. Mỗi lần xác định vị trí pos đang xét không thích hợp ➔ dời chỗ phần tử a[pos-1] đến vị trí pos
- Giải thuật thực hiện tất cả N-1 vòng lặp tìm pos, do số lượng phép so sánh và dời chỗ này phụ thuộc vào tình trạng của dãy số ban đầu, nên chỉ có thể ước lược trong từng trường hợp như sau:

Trường hợp	Số phép so sánh	Số phép gán
Tốt nhất	$\sum_{i=1}^{n-1} 1 = n-1$	$\sum_{i=1}^{n-1} 2 = 2(n-1)$
Xấu nhất	$\sum_{i=1}^{n-1} (i-1) = \frac{n(n-1)}{2}$	$\sum_{i=1}^{n-1} (i+1) = \frac{n(n+1)}{2} - 1$

Selection Sort – Ý tưởng

- Nhận xét
  - Mảng có thứ tự thì a[i]=min(a[i], a[i+1], …, a[n-1])
- Ý tưởng: mô phỏng một trong những cách sắp xếp tự nhiên nhất trong thực tế:
  - Chọn phần tử nhỏ nhất trong n phần tử ban đầu, đưa phần tử này về vị trí đúng là đầu dãy hiện hành
  - Xem dãy hiện hành chỉ còn n-1 phần tử của dãy ban đầu, bắt đầu từ vị trí thứ 2; lặp lại quá trình trên cho dãy hiện hành... đến khi dãy hiện hành chỉ còn 1 phần tử

Selection Sort – Thuật toán

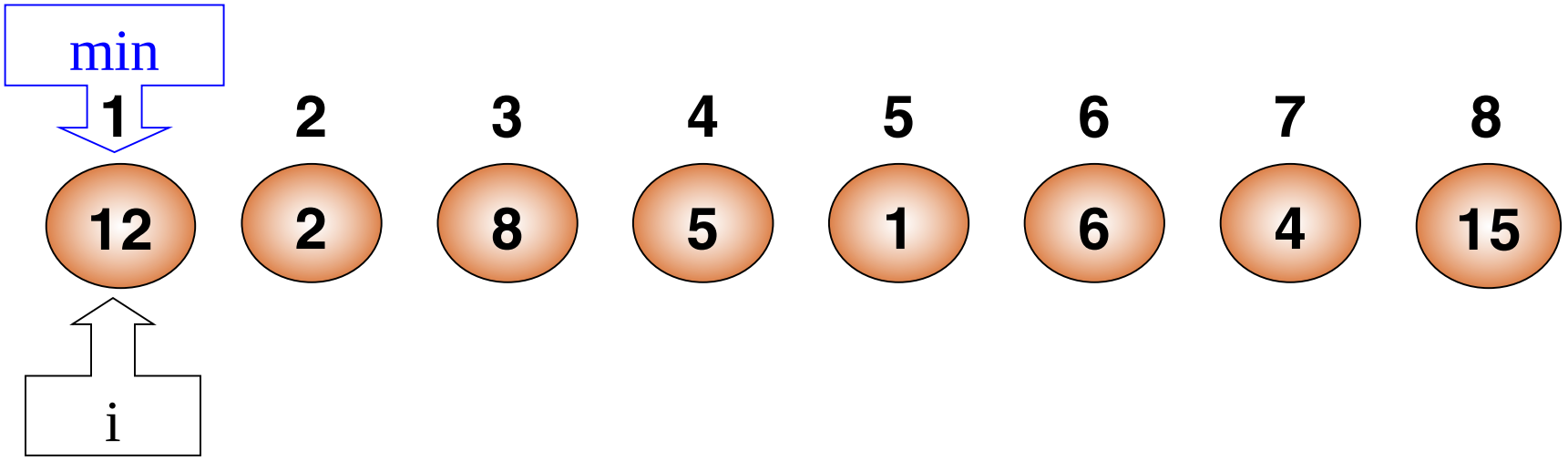
// input: dãy (a, n)
// output: dãy (a, n) đã được sắp xếp

- Bước 1 : i = 0
- Bước 2 : Tìm phần tử a[min] nhỏ nhất trong dãy hiện hành từ a[i] đến a[n-1]
- Bước 3 : Nếu min ≠ i: Đổi chỗ a[min] và a[i]
- Bước 4 : Nếu i < n:
  - i =i+1
  - Lặp lại Bước 2

  Ngược lại: Dừng. *//n phần tử đã nằm đúng vị trí*

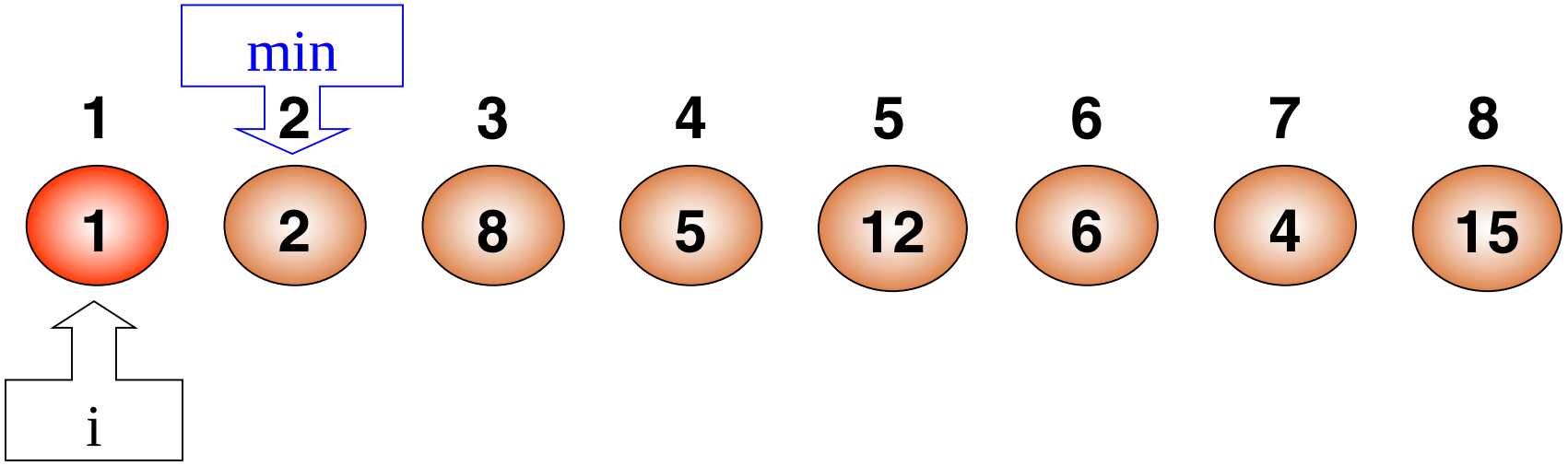
Selection Sort – Ví dụ

Find MinPos(1, 8) **Swap(a[i], a[min])**



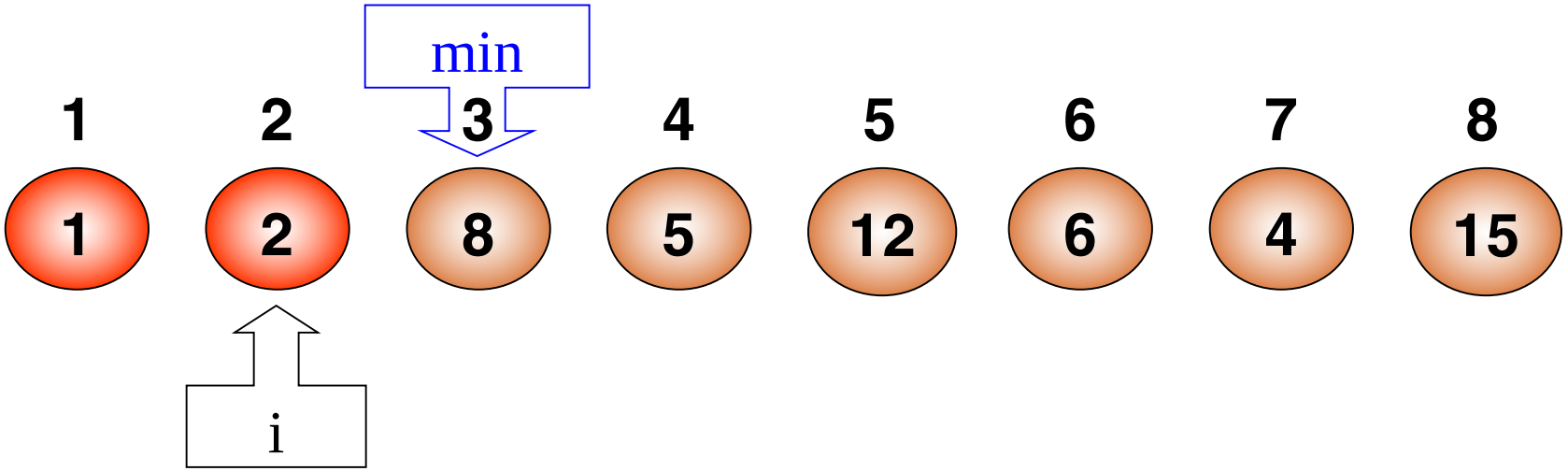
Selection Sort – Ví dụ

Find MinPos(2, 8) **Swap(a[i], a[min])**



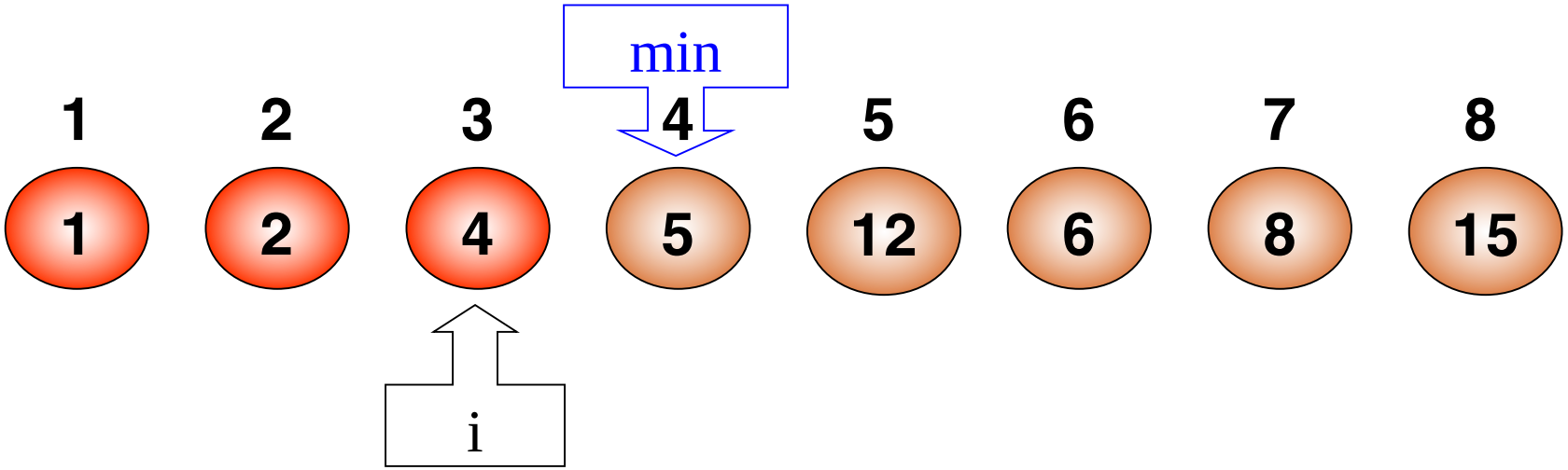
Selection Sort – Ví dụ

Find MinPos(3, 8) **Swap(a[i], a[min])**



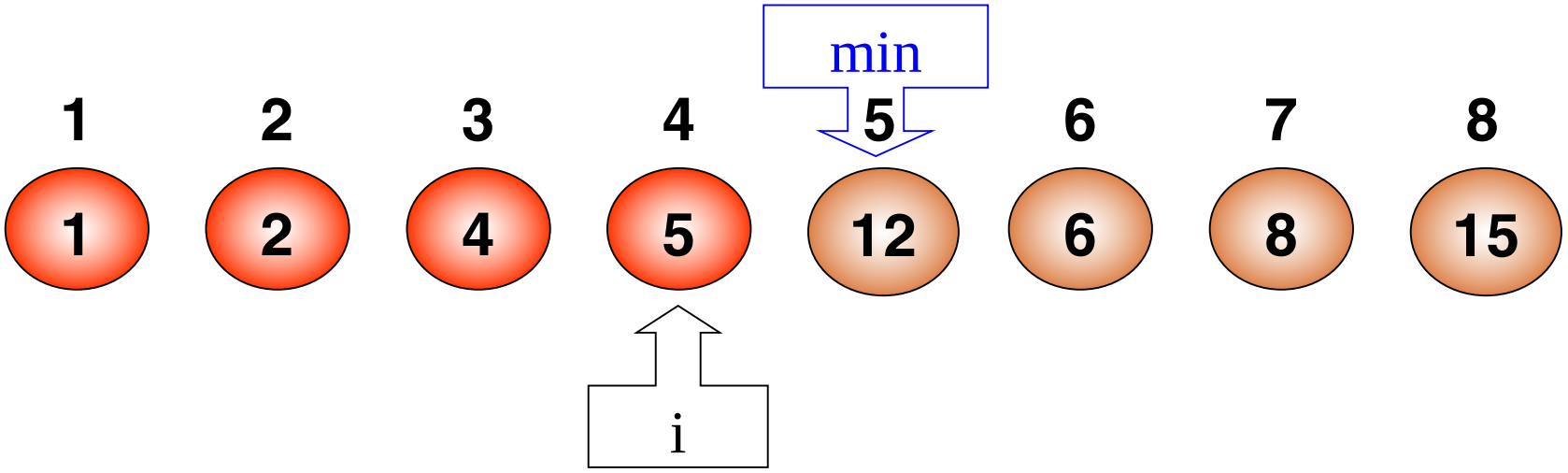
Selection Sort – Ví dụ

Find MinPos(4, 8) **Swap(a[i], a[min])**



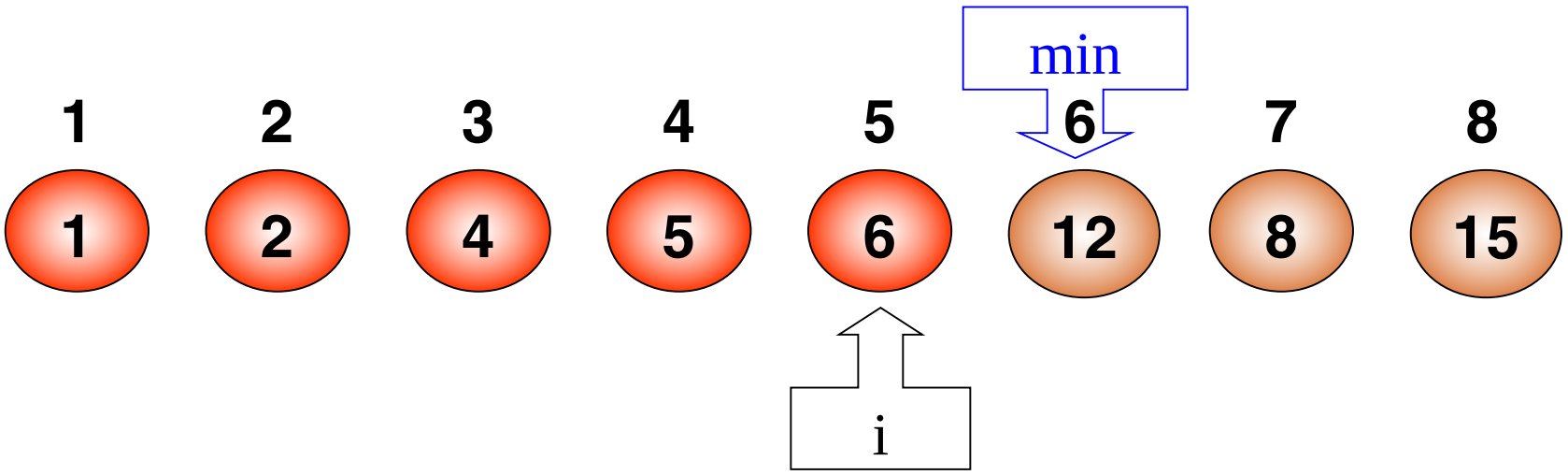
Selection Sort – Ví dụ

Find MinPos(5, 8) **Swap(a[i], a[min])**



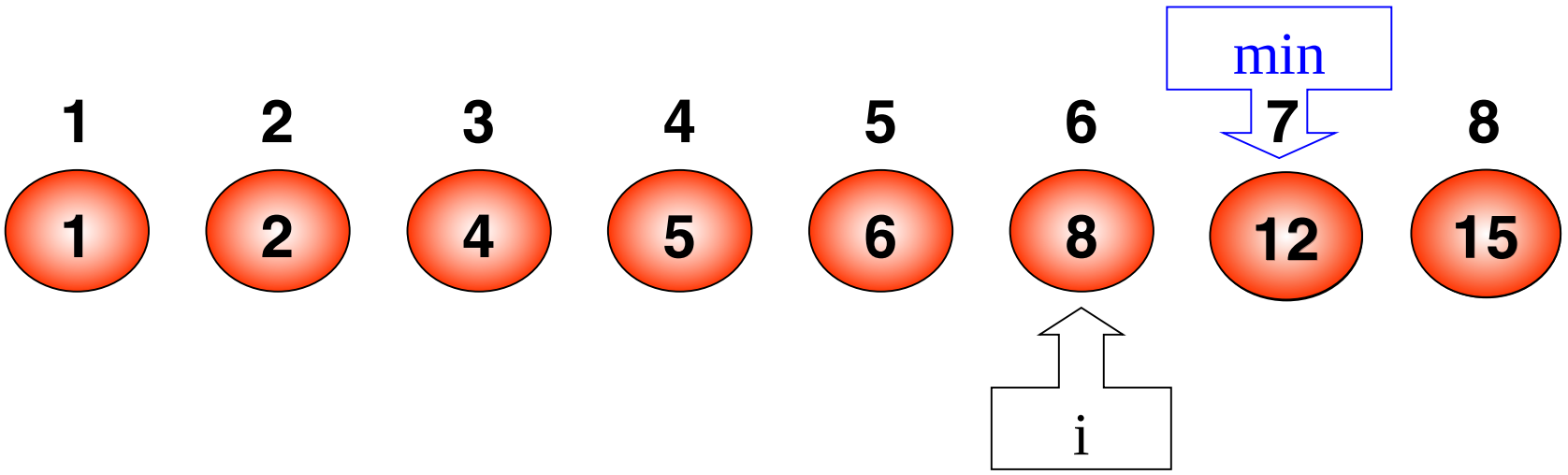
Selection Sort – Ví dụ

Find MinPos(6, 8) **Swap(a[i], a[min])**



Selection Sort – Ví dụ

Find MinPos(7, 8) **Swap(a[i], a[min])**



Selection Sort – Cài đặt


```
void SelectionSort(int a[], int n )
{
  int min; // chỉ số phần tử nhỏ nhất trong dãy hiện hành
  for (int  i=0; i<n-1; i++)
  {
      min = i;
      for(int j = i+1; j<n; j++)
          if (a[j] < a[min])
              min = j; // ghi nhận vị trí phần tử nhỏ nhất
      if (min != i)
          Swap(a[min], a[i]);
  }
}
```

Selection Sort – Đánh giá giải thuật

- Ở lượt thứ i, cần (n-i) lần so sánh để xác định phần tử nhỏ nhất hiện hành
- Số lượng phép so sánh không phụ thuộc vào tình trạng của dãy số ban đầu
- Trong mọi trường hợp, số lần so sánh là:

$$\sum_{i=1}^{n-1}(n-i)=\frac{n(n-1)}{2}$$

Trường hợp	Số lần so sánh	Số phép gán
Tốt nhất	n(n-1)/2	0
Xấu nhất	n(n-1)/2	3n

Quick Sort – Ý tưởng

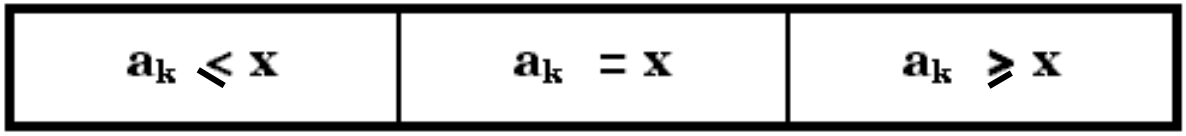
- Một vài hạn chế của thuật toán Đổi chỗ trực tiếp:
  - Mỗi lần đổi chỗ chỉ thay đổi 1 cặp phần tử trong nghịch thế; các trường hợp như: $i < j < k$ và $a_i > a_j > a_k$ (*) chỉ cần thực hiện 1 lần đổi chỗ (a_i, a_k): thuật toán không làm được
  - Độ phức tạp của thuật toán $O(N^2)$ ➔ khi N đủ lớn thuật toán sẽ rất chậm
- Ý tưởng: phân chia dãy thành các đoạn con → tận dụng được các phép đổi chỗ dạng (*) và làm giảm độ dài dãy khi sắp xếp → cải thiện đáng kể độ phức tạp của thuật toán

Quick Sort – Ý tưởng

- Giải thuật QuickSort sắp xếp dãy **a[0], a[1] ..., a[n-1]** dựa trên việc phân hoạch dãy ban đầu thành 3 phần:
  - Phần 1: Gồm các phần tử có giá trị không lớn hơn **x**
  - Phần 2: Gồm các phần tử có giá trị bằng **x**
  - Phần 3: Gồm các phần tử có giá trị không bé hơn **x**

  với x là giá trị của một phần tử tùy ý trong dãy ban đầu.
- Sau khi thực hiện phân hoạch, dãy ban đầu được phân thành 3 đoạn:
  1. a[k] $\leq$ x , với k = 1 .. j
  2. a[k] = x , với k = j+1 .. i-1
  3. a[k] $\geq$ x , với k = i..n-1

Quick Sort – Ý tưởng



- Đoạn thứ 2 đã có thứ tự
- Nếu các đoạn 1 và 3 chỉ có 1 phần tử thì chúng cũng đã có thứ tự, khi đó dãy con ban đầu đã được sắp
- Ngược lại, nếu các đoạn 1 và 3 có nhiều hơn 1 phần tử thì dãy con ban đầu chỉ có thứ tự khi các đoạn 1, 3 được sắp
- Để sắp xếp các đoạn 1 và 3, ta lần lượt tiến hành việc phân hoạch từng dãy con theo cùng phương pháp phân hoạch dãy ban đầu vừa trình bày …

Quick Sort – Giải thuật

// input: dãy con (a, left, right)

// output: dãy con (a, left, right) được sắp tăng dần

- Bước 1: Nếu left = right *// dãy có ít hơn 2 phần tử*
  Kết thúc; *// dãy đã được sắp xếp*
- Bước 2: Phân hoạch dãy a[left] … a[right] thành các đoạn: a[left].. a[j], a[j+1].. a[i-1], a[i].. a[right]

  // Đoạn 1 ≤ x

  // Đoạn 2: a[j+1].. a[i-1] = x

  // Đoạn 3: a[i].. a[right] ≥ x
- Bước 3: Sắp xếp đoạn 1: a[left].. a[j]
- Bước 4: Sắp xếp đoạn 3: a[i].. a[right]

Quick Sort – Phân hoạch dãy

// input: dãy con a[left], ..., a[right]

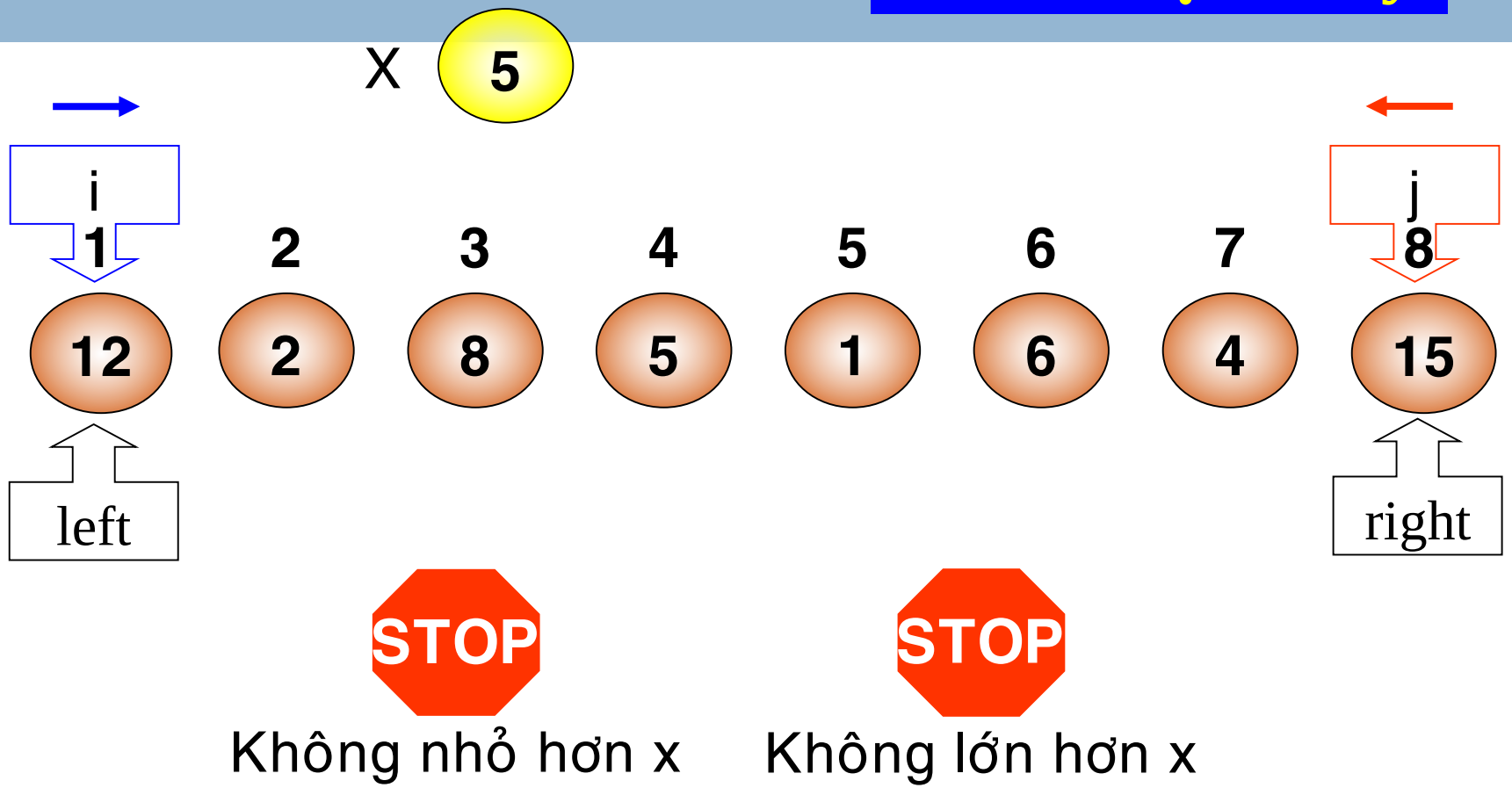
// output: dãy con chia thành 3 đoạn: đoạn 1 ≤ đoạn 2 ≤ đoạn 3

- Bước 1: Chọn tùy ý một phần tử a[p] trong dãy con là giá trị mốc:
  x = a[p];
- Bước 2: Duyệt từ 2 đầu dãy để phát hiện và hiệu chỉnh cặp phần tử a[i], a[j] vi phạm điều kiện
  - Bước 2.1: i = left; j = right;
  - Bước 2.2: Trong khi (a[i]<x) i++;
  - Bước 2.3: Trong khi (a[j]>x) j--;
  - Bước 2.4: Nếu i<= j // a[i] ≥ x ≥ a[j] mà a[j] đứng sau a[i]
    - Hoán vị (a[i], a[j]); i++; j--;
  - Bước 2.5: Nếu i < j: Lặp lại Bước 2.2 //chưa xét hết mảng

  //Hết duyệt

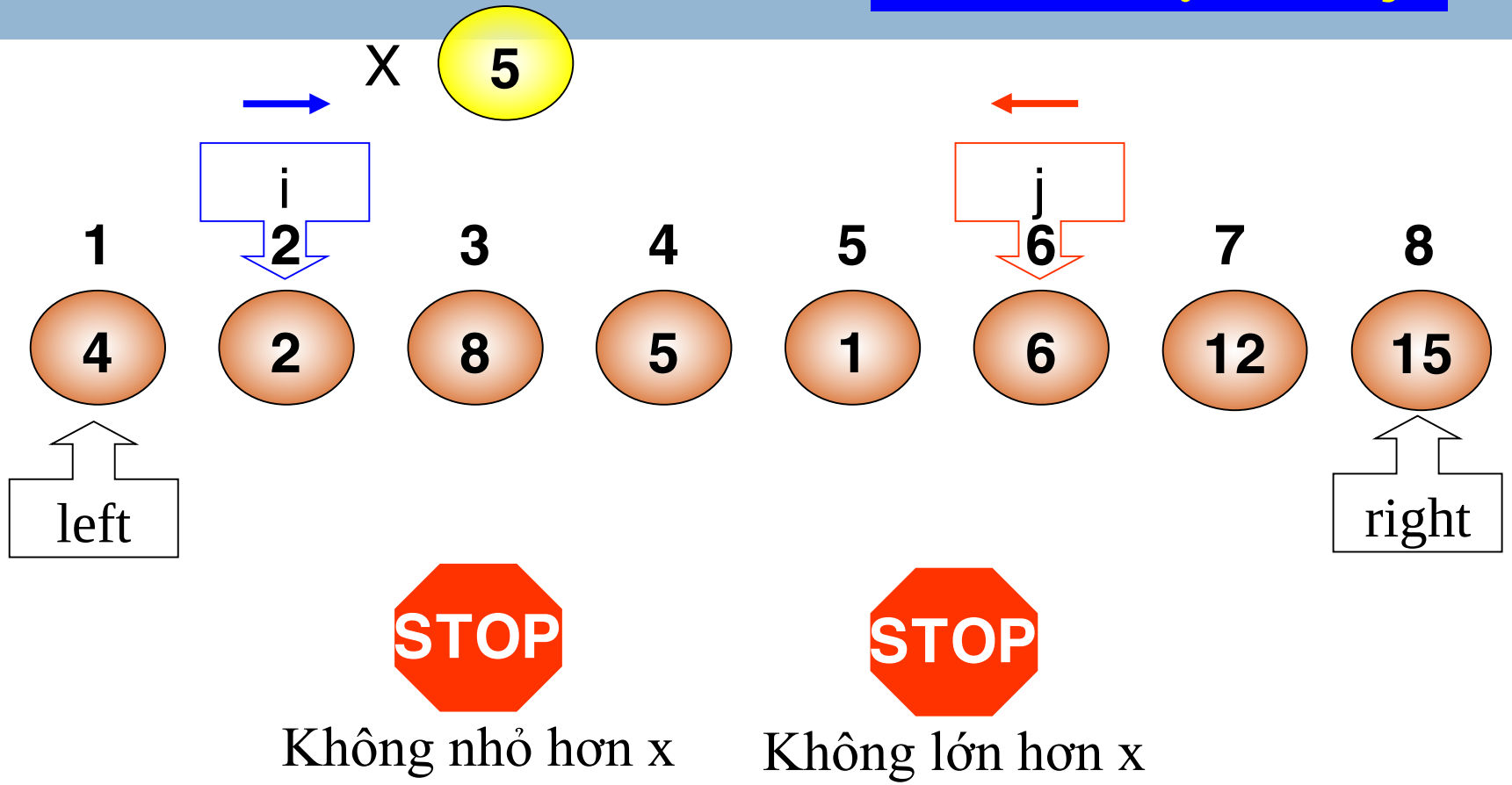
Quick Sort – Ví dụ

Phân hoạch dãy

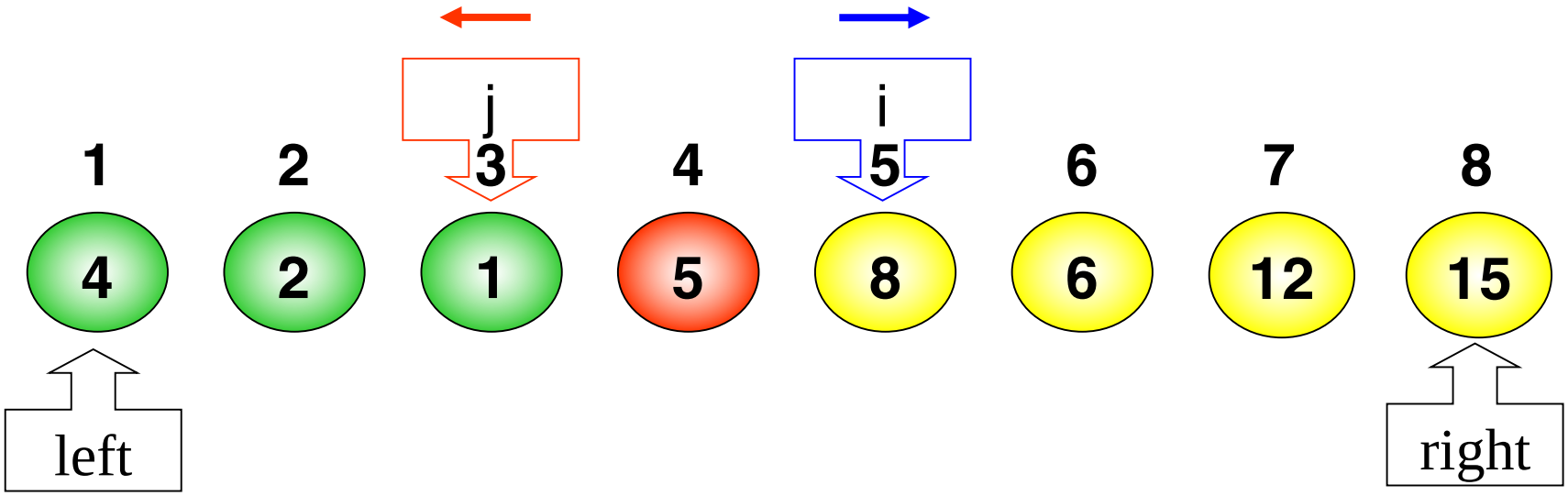


Quick Sort – Ví dụ

Phân hoạch dãy

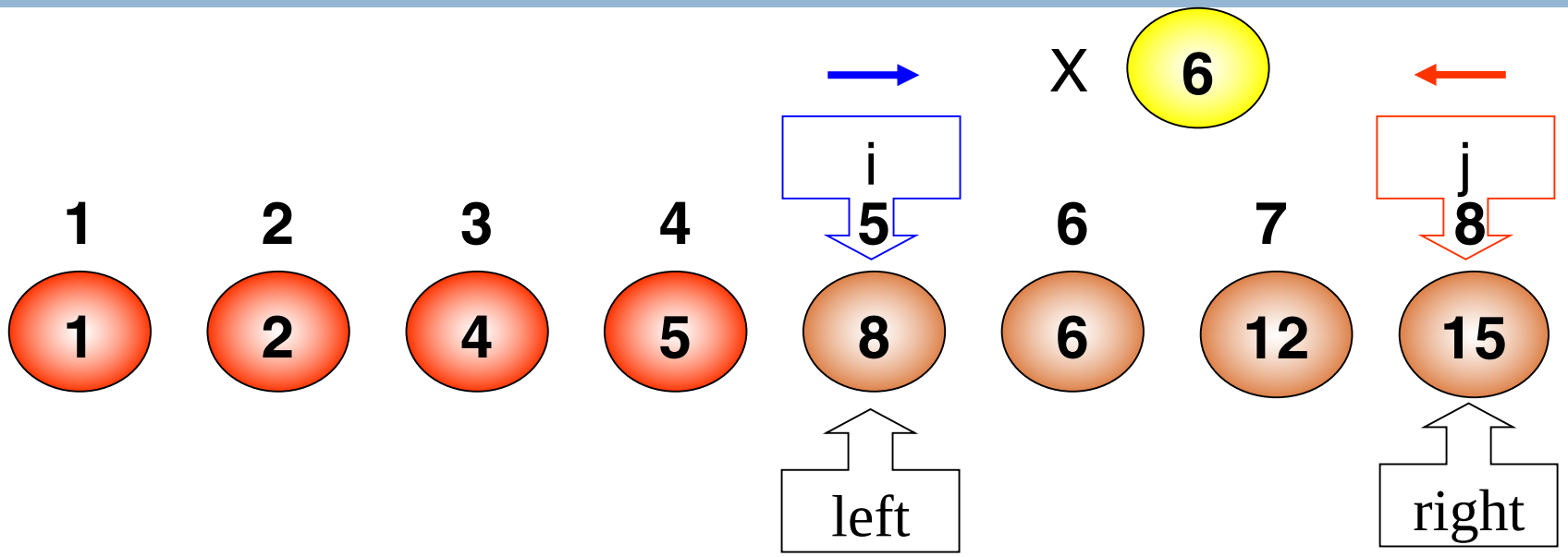


Quick Sort – Ví dụ



Quick Sort – Ví dụ

Phân hoạch dãy



Sắp xếp đoạn 3

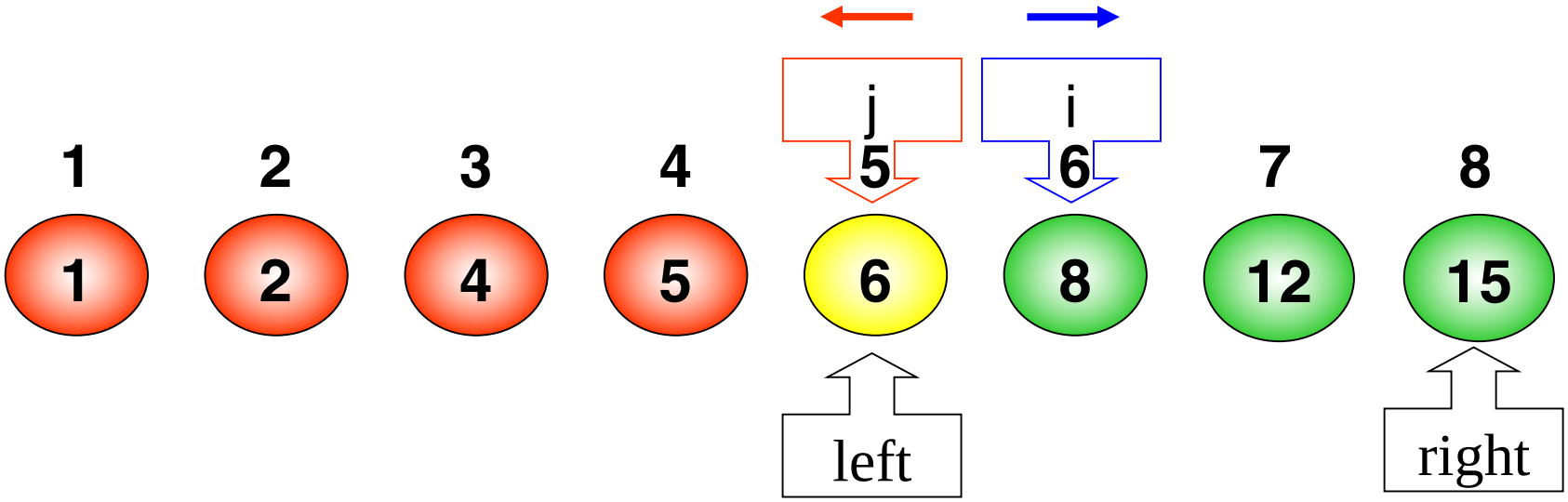
STOP

Không nhỏ hơn x

STOP

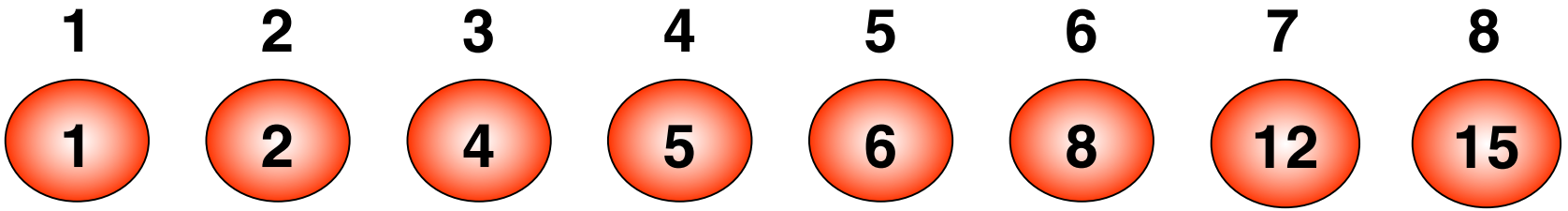
Không lớn hơn x

Quick Sort – Ví dụ



Sắp xếp đoạn 3

Quick Sort – Ví dụ



Quick Sort – Cài đặt


```
void QuickSort(int a[], int left, int right)
{
   int  i, j, x;
   if (left ≥ right)   return;
   x = a[(left+right)/2]; // chọn phần tử giữa làm giá trị mốc
   i = left; j = right;
    do{
        while(a[i] < x) i++;
        while(a[j] > x) j--;
        if(i <= j) {
                Swap(a[i], a[j]);
                i++ ; j--;
        }
   } while(i < j);
   if(left<j) QuickSort(a, left, j);
   if(i<right) QuickSort(a, i, right);
}
```

Quick Sort – Đánh giá giải thuật

- Về nguyên tắc, có thể chọn giá trị mốc x là một phần tử tùy ý trong dãy, nhưng để đơn giản, phần tử có vị trí giữa thường được chọn, khi đó p = (l +r)/ 2
- Giá trị mốc x được chọn sẽ có tác động đến hiệu quả thực hiện thuật toán vì nó quyết định số lần phân hoạch
  - Số lần phân hoạch sẽ ít nhất nếu ta chọn được x là phần tử trung vị (median), nhiều nhất nếu x là cực trị của dãy
  - Tuy nhiên do chi phí xác định phần tử median quá cao nên trong thực tế người ta không chọn phần tử này mà chọn phần tử nằm chính giữa dãy làm mốc với hy vọng nó có thể gần với giá trị median

Quick Sort – Đánh giá giải thuật

Hiệu quả phụ thuộc vào việc chọn giá trị mốc:

- Trường hợp tốt nhất: mỗi lần phân hoạch đều chọn phần tử median làm mốc, khi đó dãy được phân chia thành 2 phần bằng nhau và cần $\log_2(n)$ lần phân hoạch thì sắp xếp xong
- Nếu mỗi lần phân hoạch chọn phần tử có giá trị cực đại (hay cực tiểu) là mốc ➔ dãy sẽ bị phân chia thành 2 phần không đều: một phần chỉ có 1 phần tử, phần còn lại gồm (n-1) phần tử, do vậy cần phân hoạch n lần mới sắp xếp xong

Quick Sort – Đánh giá giải thuật

- Độ phức tạp thuật toán

Trường hợp	Độ phức tạp
Tốt nhất	O(NlogN)
Trung bình	O(NlogN)
Xấu nhất	$O(N^{2)}$

Bài tập

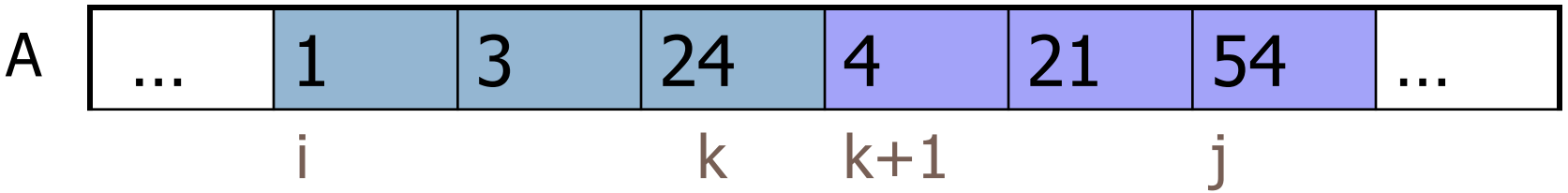
- Mô tả quá trình sắp xếp dãy số sau đây bằng thuật toán sắp xếp QuickSort
- 45,24, 432, 23, 876, 34, 789, 4

Thuật toán MergeSort

Ý tưởng:

- Giả sử ta có hai dãy **A[i],..,A[k]** và **A[k+1],..,A[j]** và hai dãy này đã được sắp.
- Thực hiện trộn hai dãy trên để được dãy **A[i],..,A[j]** cũng được sắp
- Do hai dãy **A[i],..,A[k]** và dãy **A[k+1],..,A[j]** đã được sắp nên việc trộn hai dãy thành một dãy được sắp là rất đơn giản.
- Vậy trộn như thế nào?

Ví dụ: Trộn hai dãy sau



Thuật toán trộn

- Sử dụng hai biến left, right, t và sử dụng mảng phụ *B[i],..,B[j]*. left xuất phát từ i, right xuất phát từ k+1, t xuất phát tử i trên mảng phụ B.
- Nếu *A[left].key<A[right].key* thì *B[t]←A[left], t ←t+1* và *left←left+1*
- Nếu *A[left].key≥A[right].key* thì *B[t]←A[right], t ←t+1 và right←right+1*
- Quá trình trên được thực hiện cho đến khi *left>k* hoặc *right>j* thì dừng lại.

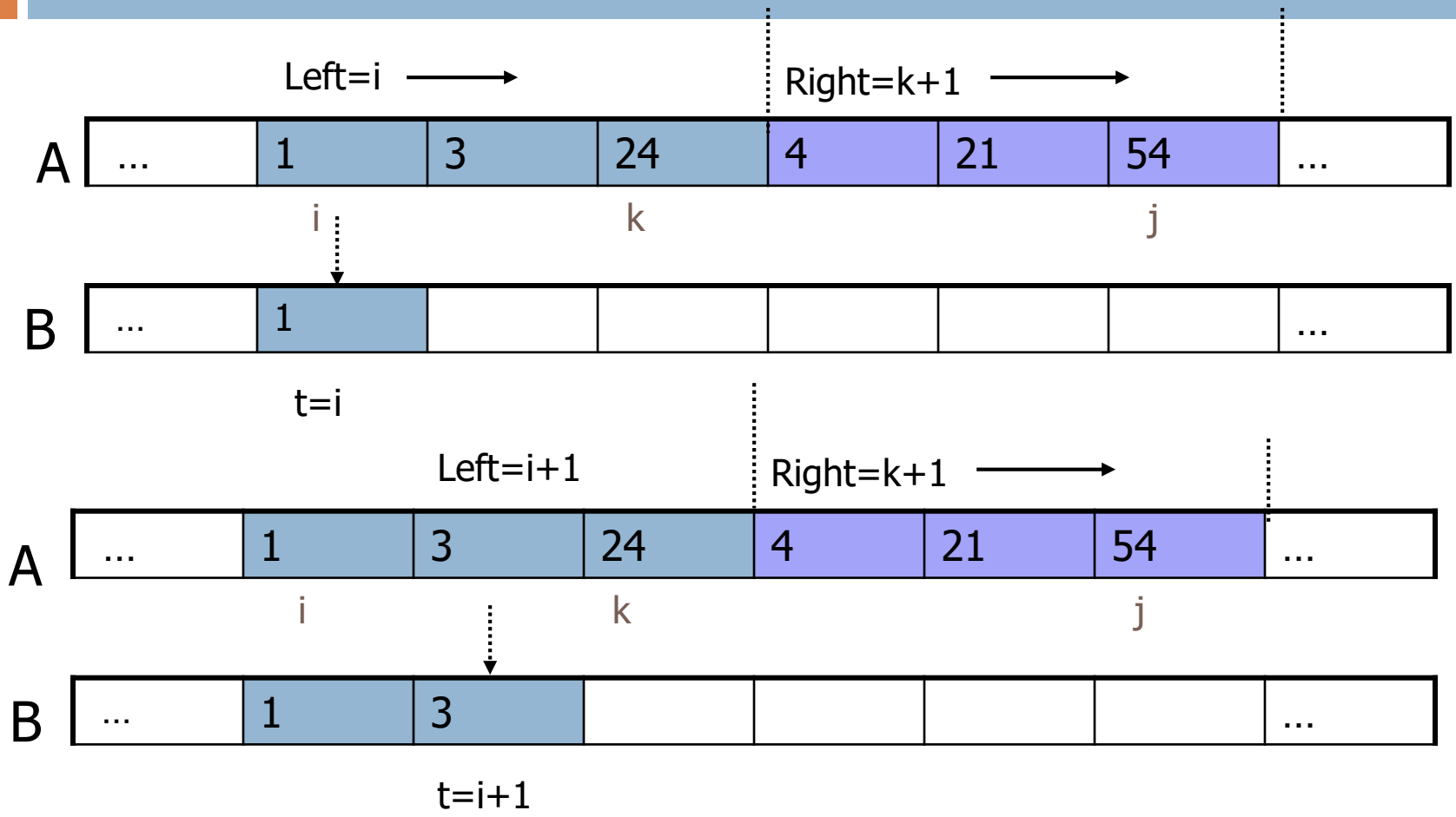
Thuật toán trộn (tiếp)

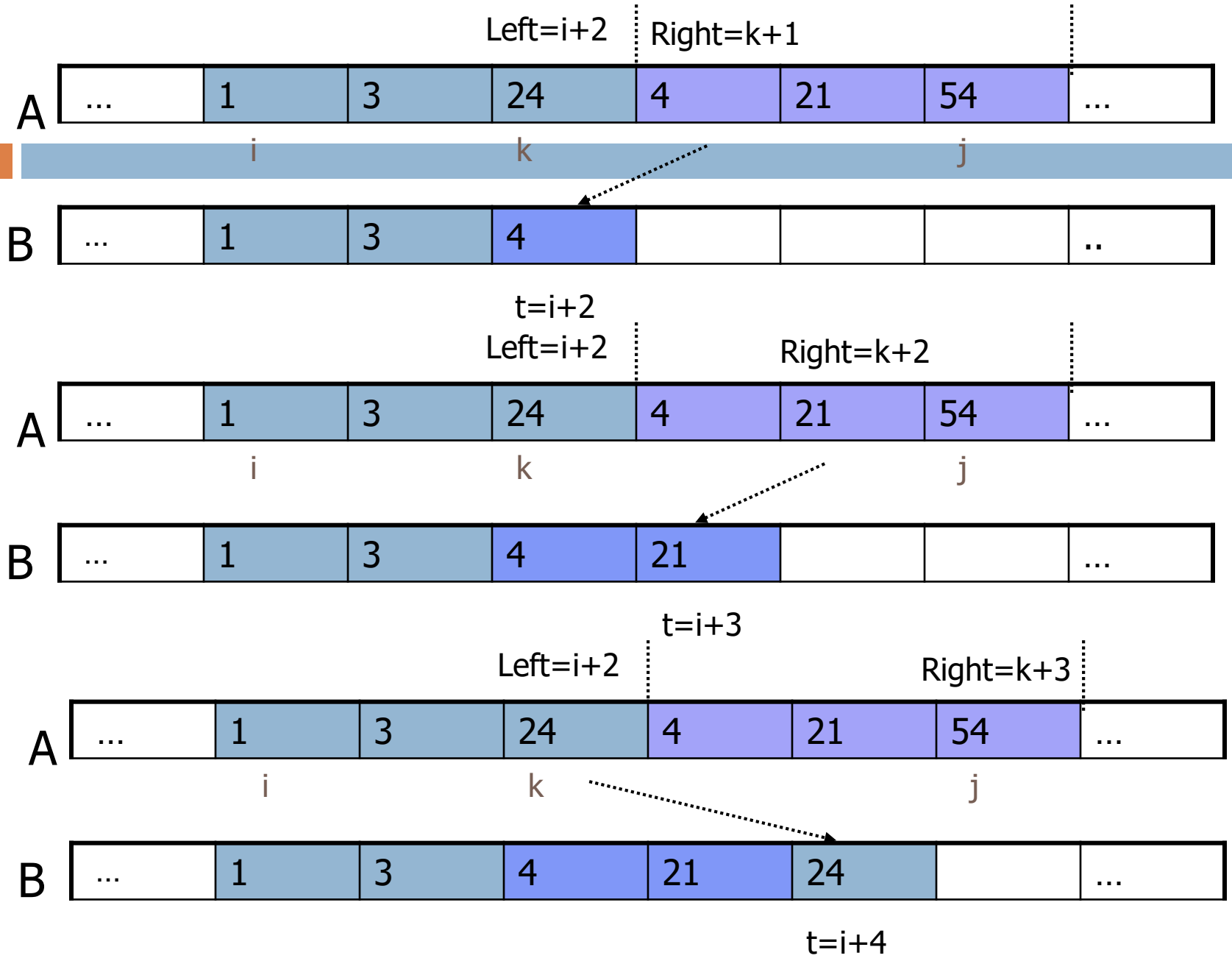
- Nếu *left>k* thì *B[t]←A[right],..,B[j]←A[j].*

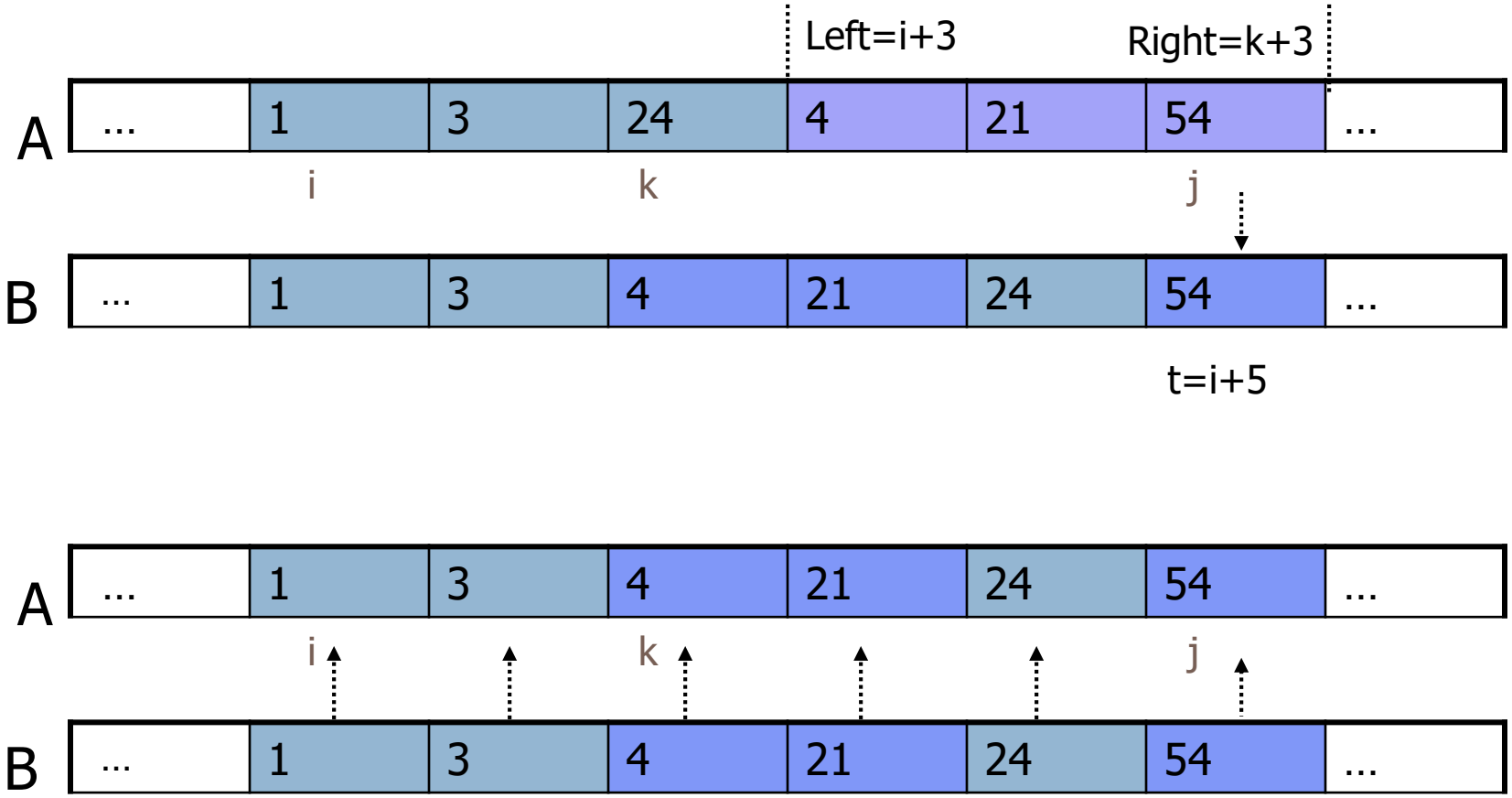
- Nếu right>j thì B[t]←A[left], B[t+1]←A[letf+1],.., B[t+k-left]←A[k].

- Gán A[i] ←B[i], .., A[j] ←B[j]

Quá trình trộn dãy




Left=i+2
Right=k+1
A
...
1
3
24
4
21
54
...
i
k
j
B
...
1
3
4
..
t=i+2
Left=i+2
Right=k+2
A
...
1
3
24
4
21
54
...
i
k
j
B
...
1
3
4
21
...
t=i+3
Left=i+2
Right=k+3
A
...
1
3
24
4
21
54
...
i
k
j
B
...
1
3
4
21
24
...
t=i+4


Left=i+3
Right=k+3
A
...
1
3
24
4
21
54
...
i
k
j
B
...
1
3
4
21
24
54
...
t=i+5
A
...
1
3
4
21
24
54
...
i
k
j
B
...
1
3
4
21
24
54
...

Thuật toán giả mã

Algorithm *Merge(array A, int i, int k, int j)*
Input: Hai dãy A[i],..,A[k] và A[k+1],..,A[j] đã được sắp và các số nguyên i, j
Output: Dãy A[i],..,A[j] cũng được sắp

```
left ←i; right←k+1; t ←i;
While (left≤k) and (right≤j)  do
    if A[left].key<A[right].key then
            B[t] ← A[left];
            left ←left+1;
            t ←t+1;
    else
             B[t] ← A[right];
            right ←right+1;
            t ←t+1 ;            //kết thúc while
If left>k then
    for r←right to j do
            B[t] ← A[r];
            t++;
else
    for r← left to k do
            B[t] ←A[r];
           t++;
for r ←i to j do
  A[r] ← B[r] ;
```

Thuật toán

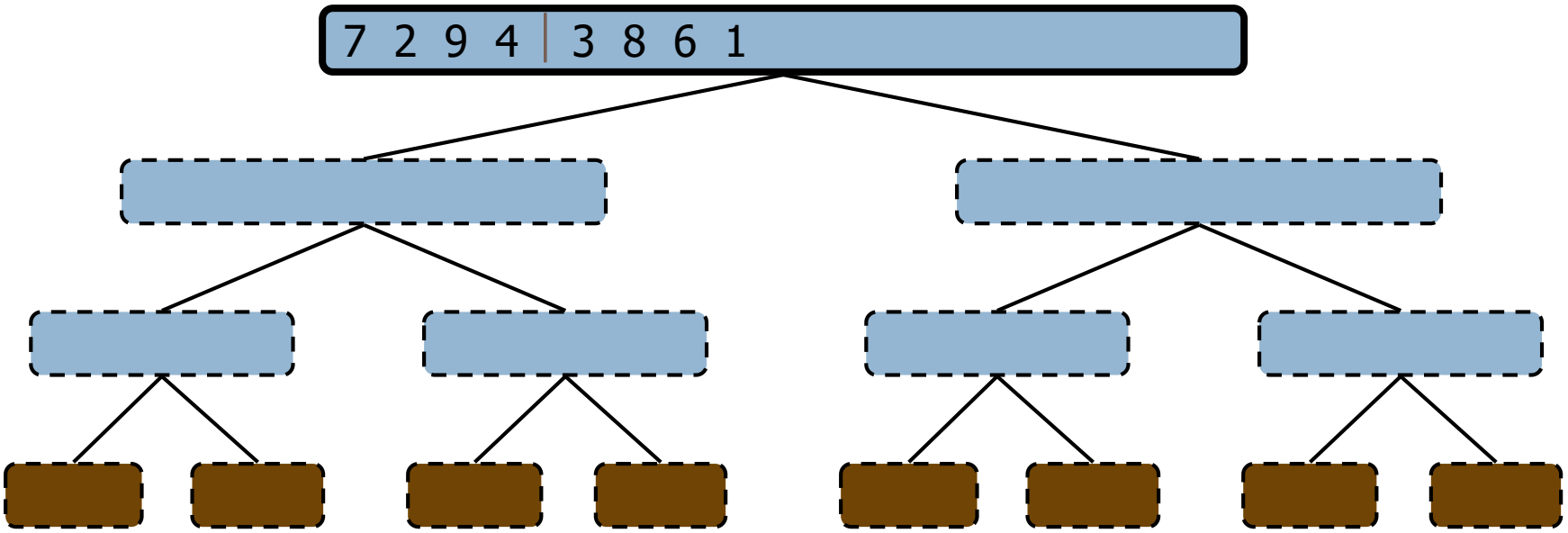
- Để sắp xếp dãy A[1],..,A[n] ta thực hiện như sau:
- Chia dãy trên thành hai dãy:A[1],..,A[k] và dãy A[k+1],..,A[n], trong đó k=(n+1)/2
- Thực hiện sắp xếp 2 dãy A[1],..,A[k] và A[k+1],..,A[n] độc lập cũng theo thuật toán Mergesort.
- Thực hiện trộn hai dãy:A[1],..,A[k] và dãy A[k+1],..,A[n] để được dãy A[1],..A[n] cũng được sắp

Thuật toán giả mã

Algorithm ***Mergesort(array A,int i, int j)***
Input: Dãy các phần tử A[i],..,A[j]
Output:Dãy A[i],..,A[j] được sắp.
if i<j **then**
    k←(i+j)/2;
    Mergesort(A,i, k);
    Mergesort(A, k+1,j);
    Merge(A, i, k, j);

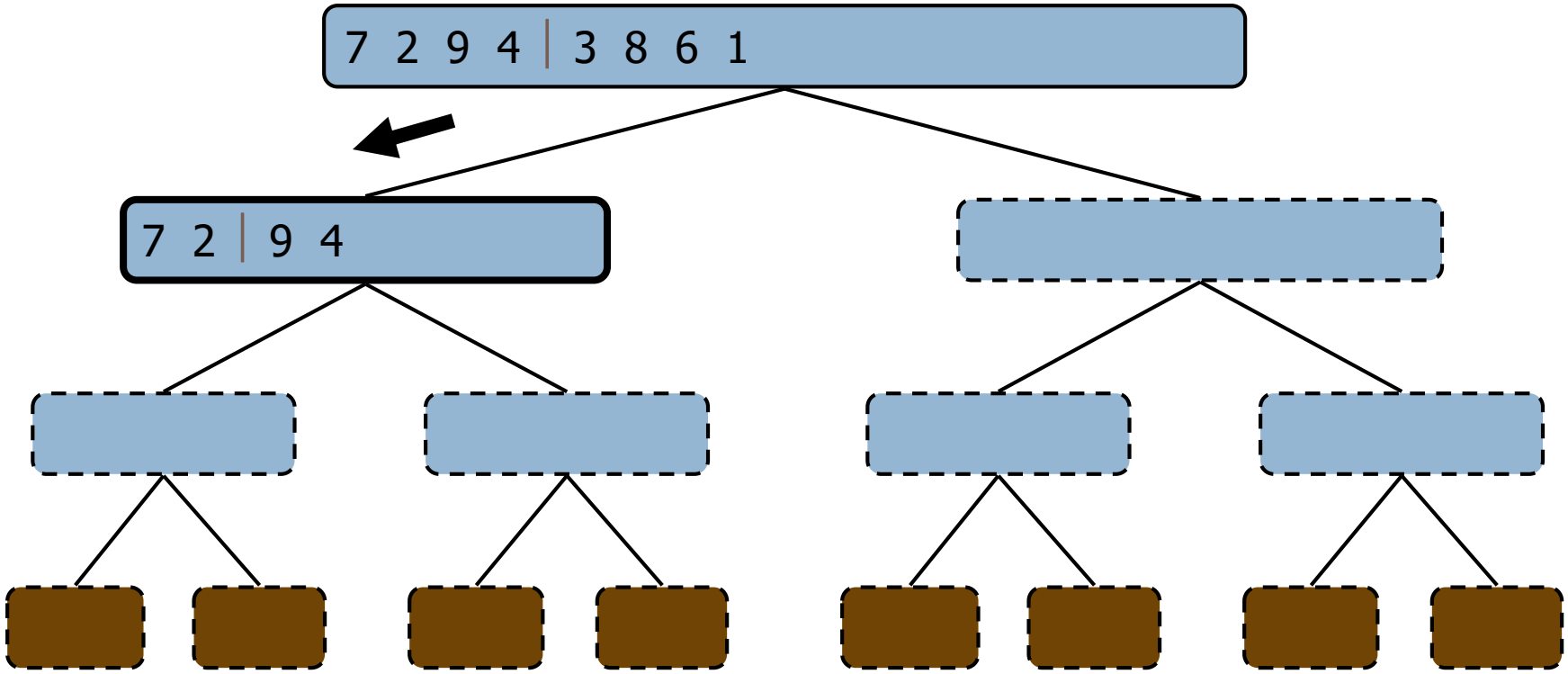
Mô tả quá trình thực hiện sắp xếp

- Ví dụ xắp xếp dãy:A= 7 2 9 4 3 8 6 1
- Gọi thủ tục MergeSort(A, 1, 8), chia đôi dãy



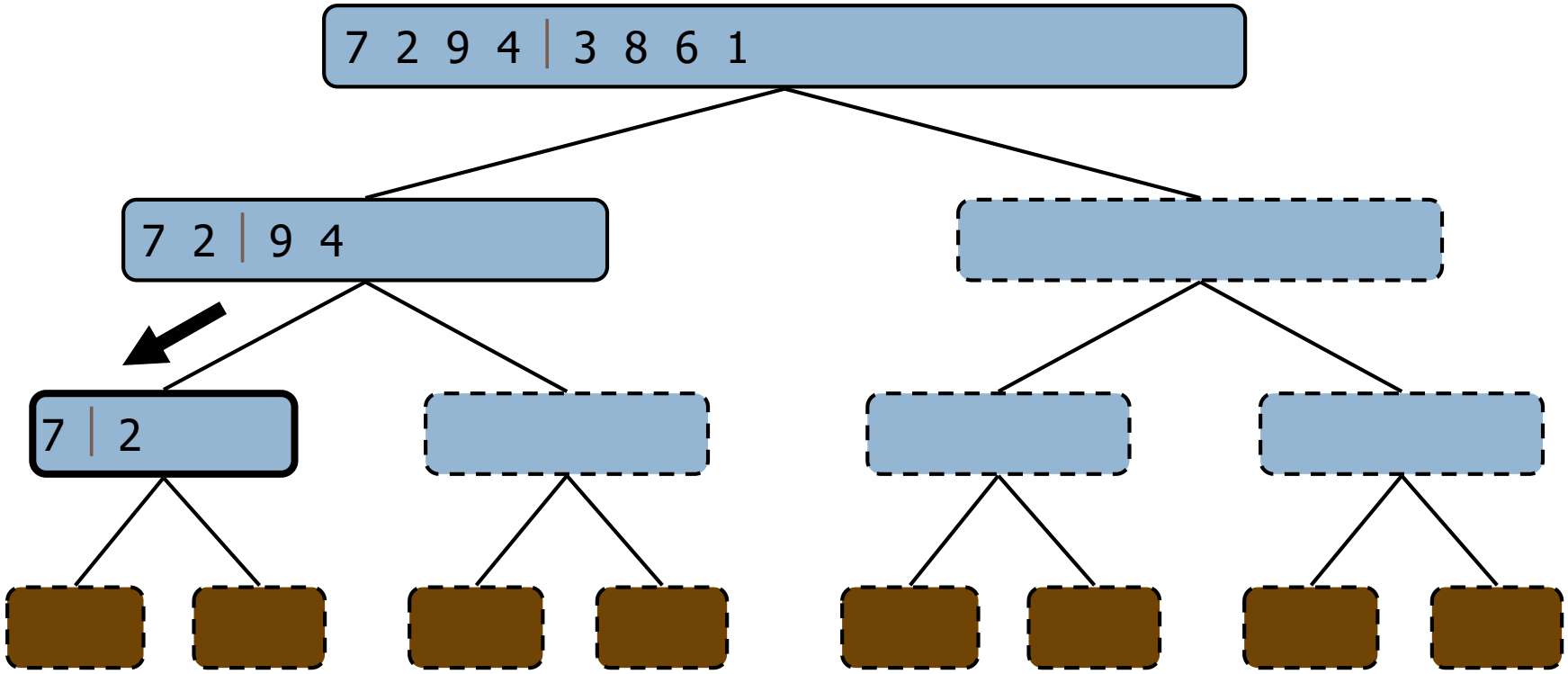
Tiếp

- Gọi đệ qui và phân chia Mergesort(A,1,4)



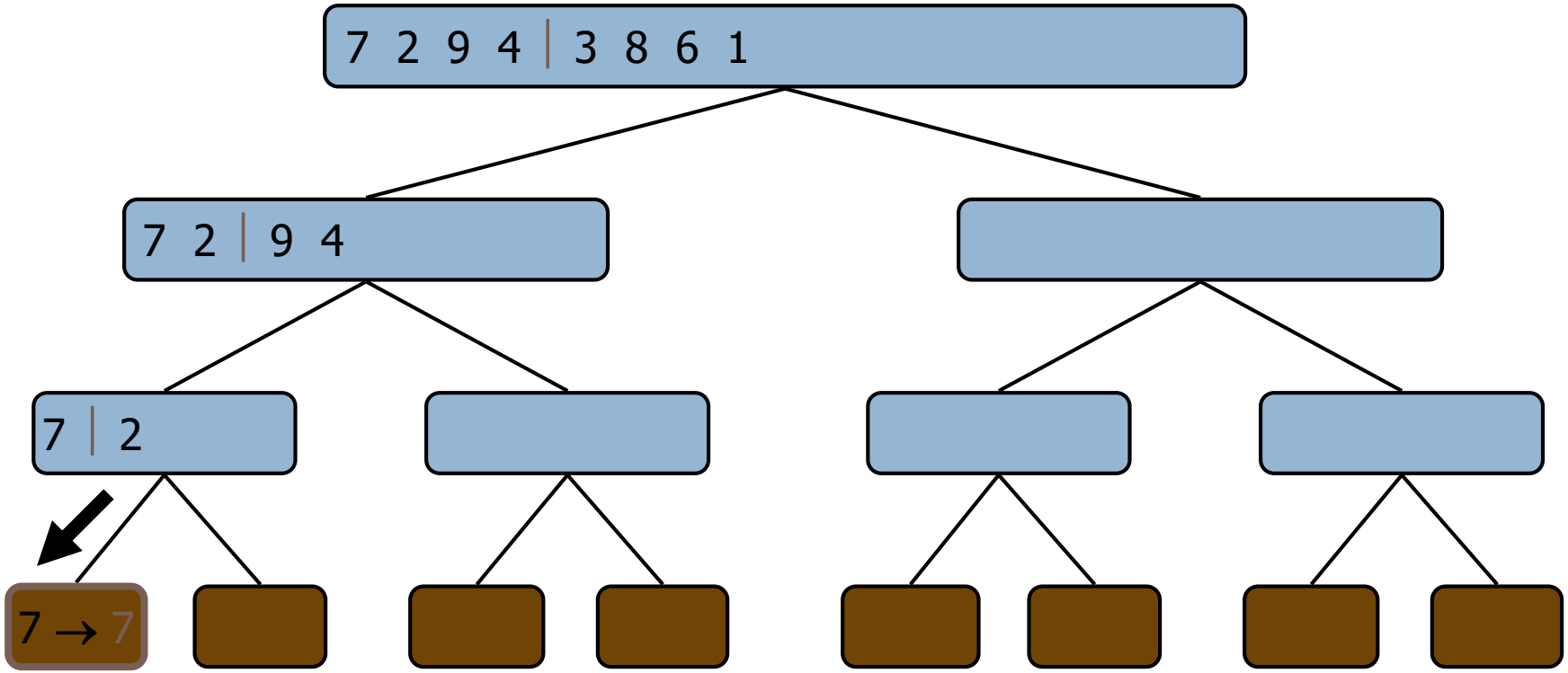
Tiếp

- Gọi đệ qui và phân chia Mergesort(A,1,2)



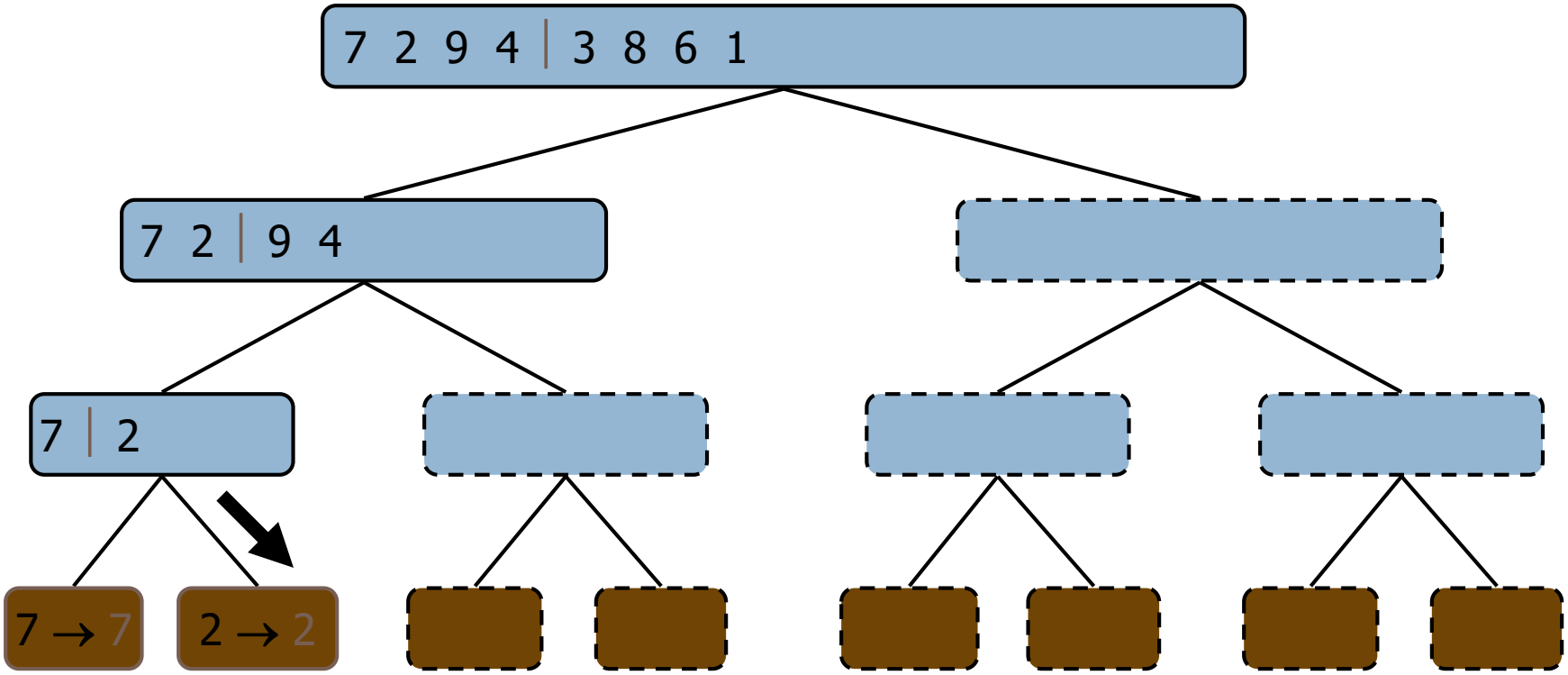
Tiếp

- Gọi đệ qui Mergesort(A,1,1), đây là trường hợp cơ sở



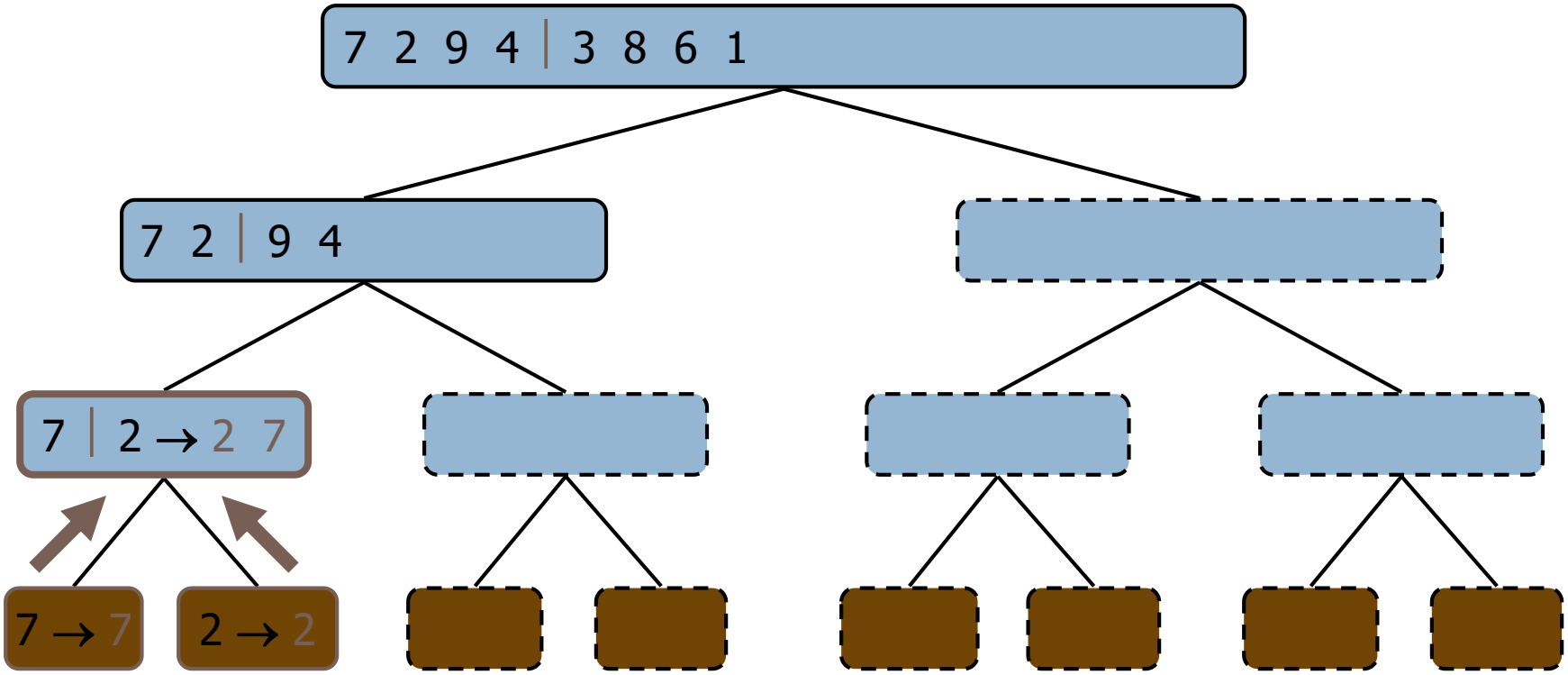
Tiếp

- Gọi đệ qui Mergesort(A,2,2), đây là trường hợp cơ sở



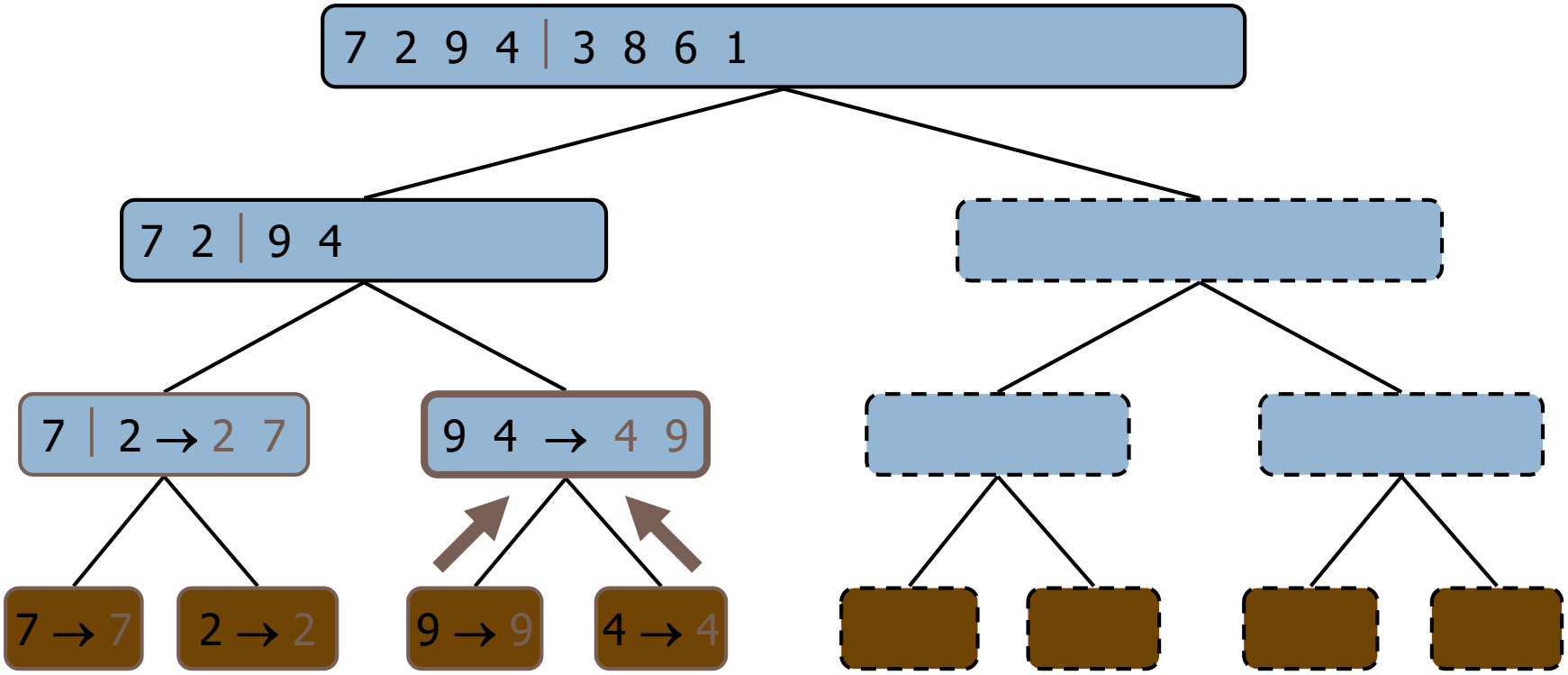
Tiếp

- Trộn merge(A,1,1,2)



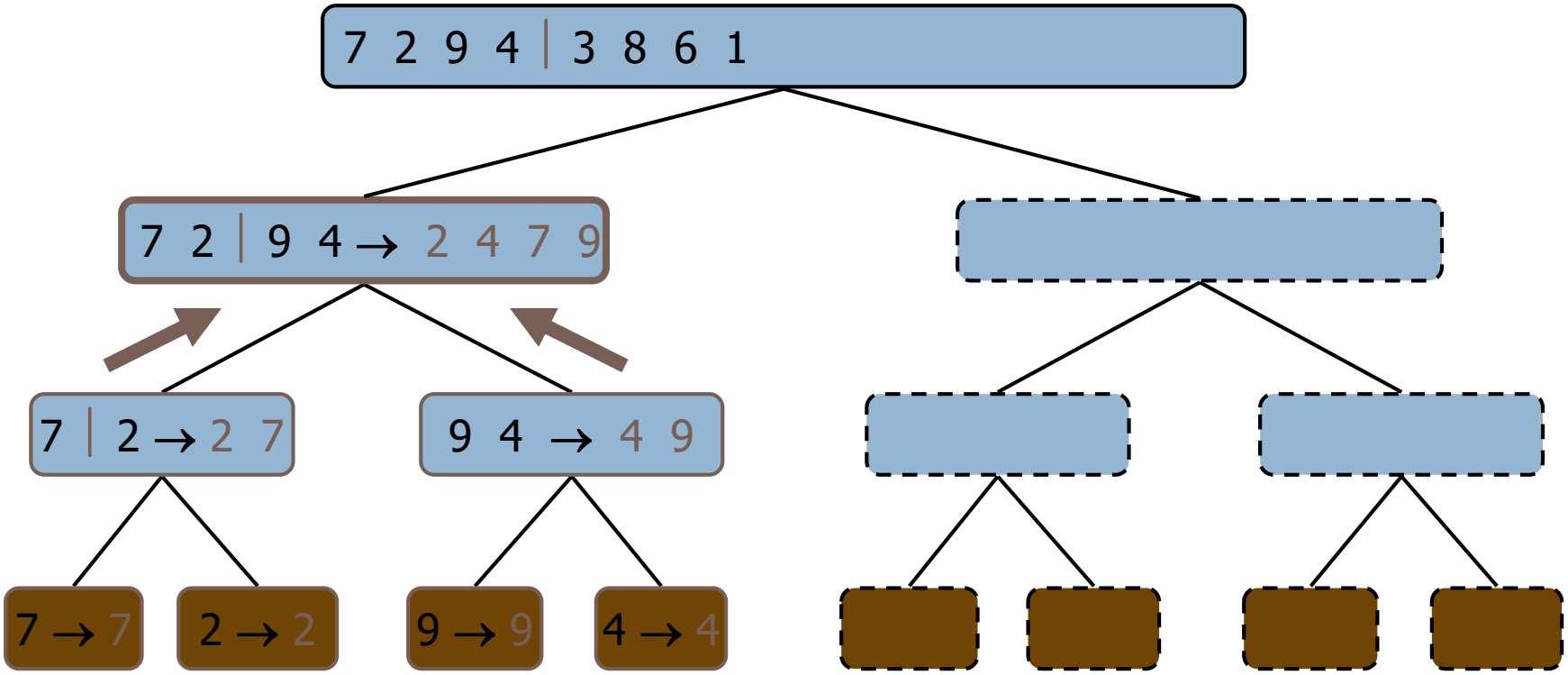
Execution Example (cont.)

- Gọi đệ qui Mergesort(A,3,3), Mergesort(A,4,4) và trộn merge(A,3,3,4)



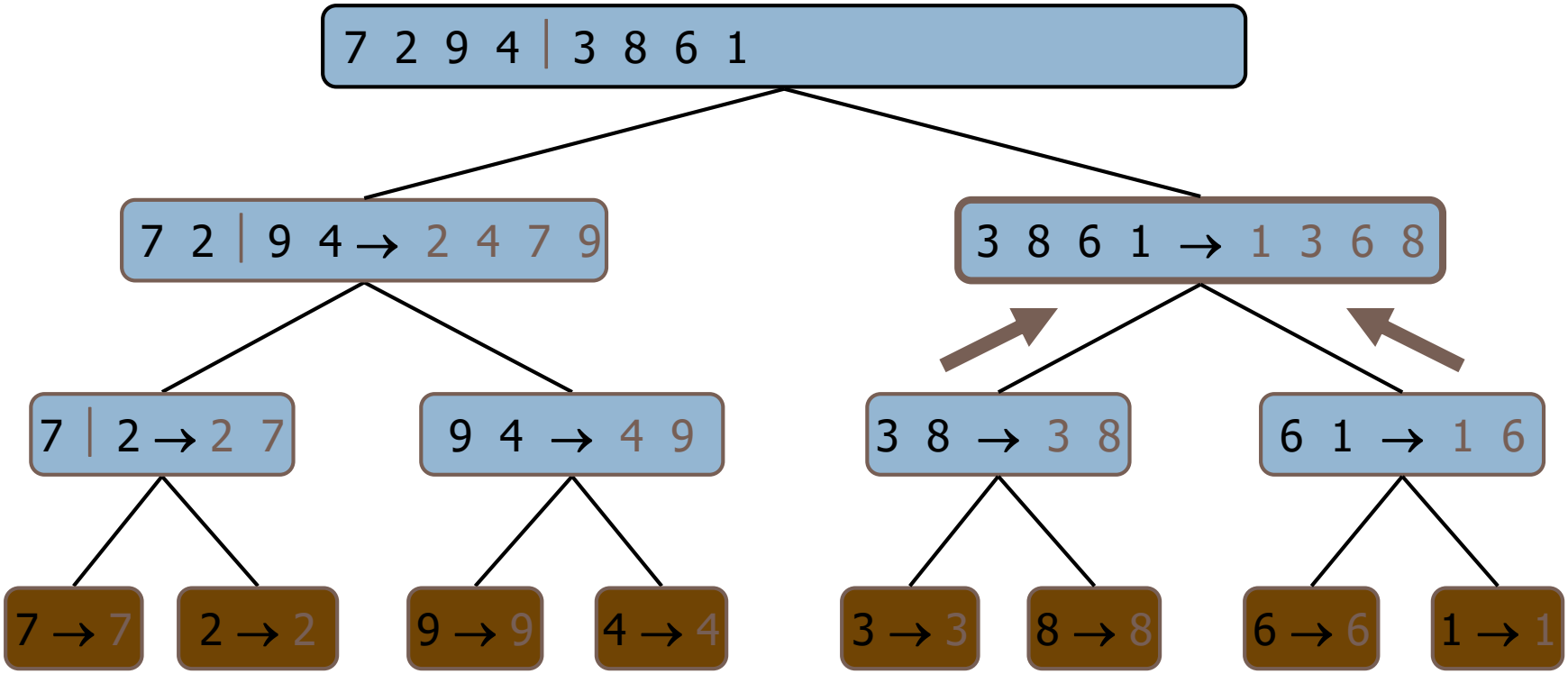
Tiếp

- Trộn merge(A,1,2,4)



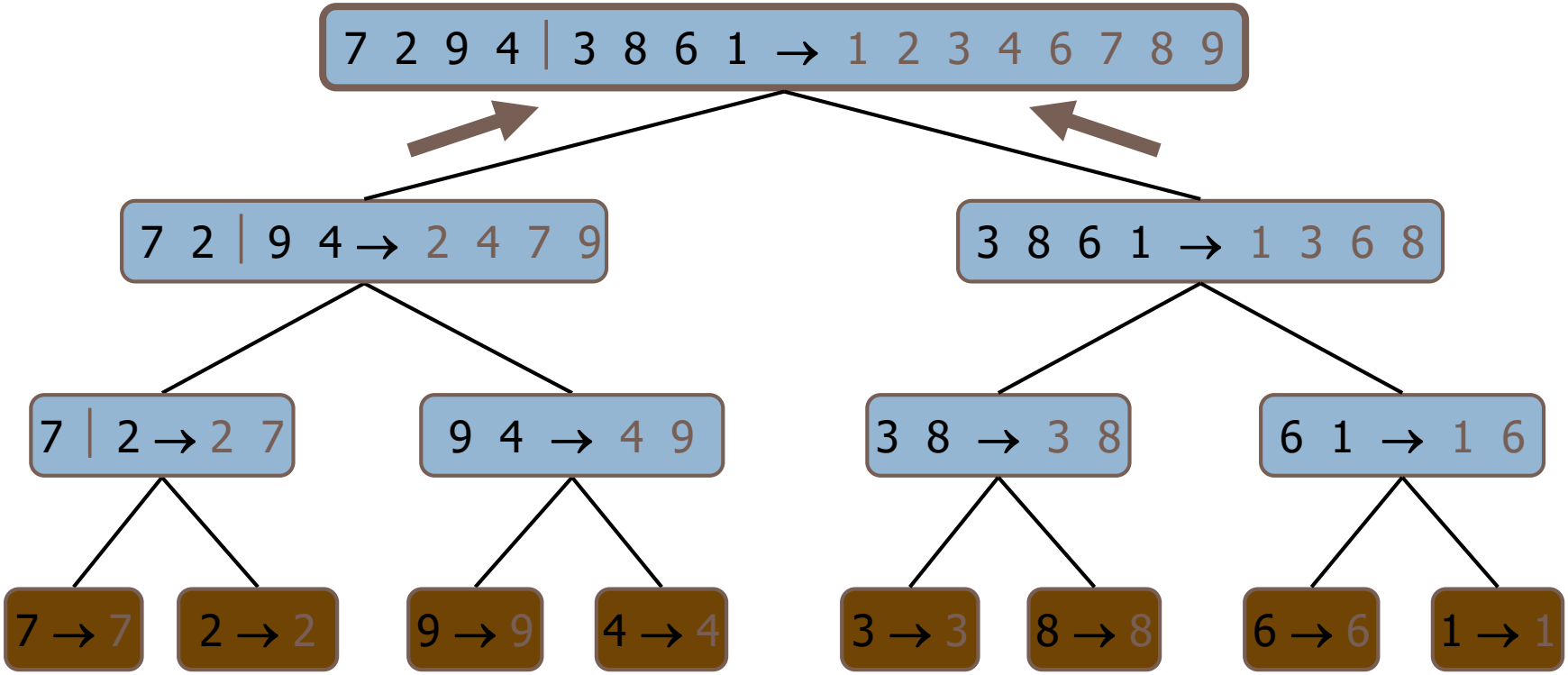
Tiếp

- Tương tự như trên với nửa bên phải của dãy



Tiếp

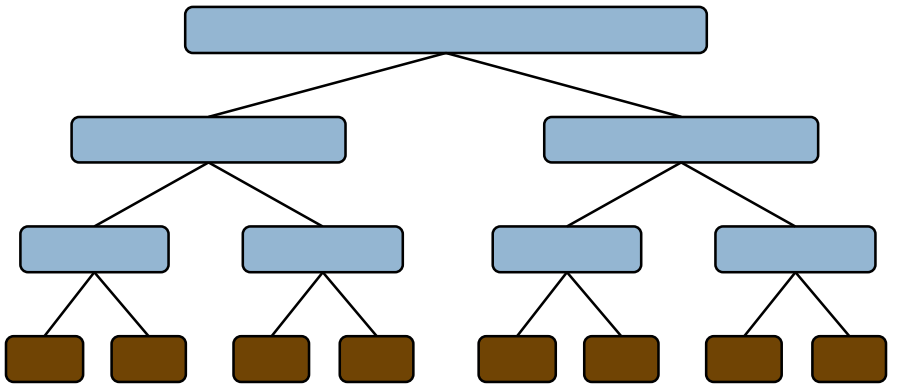
- Trộn hai nửa dãy thành dãy được sắp merge(A, 1, 4, 8)



Thời gian chạy của thuật toán

- Chiều cao ***h*** của cây merge-sort là ***O***(log ***n***)
  - Tại mỗi bước gọi đệ qui ta chia dãy cần sắp thành hai phần,
- Thời tổng thời gian làm việc trên các nút ở mức ***i*** nhiều nhất là ***O***(***n***)
  - Chúng ta chia và trộn 2***i*** chuỗi có kích thước là ***n***/2***i***
  - Chúng ta gọi 2***i***+1 lần đệ qui
- Vì vậy, tổng thời gian chạy của thuật toán mergesort là ***O***(***n*** log ***n***)

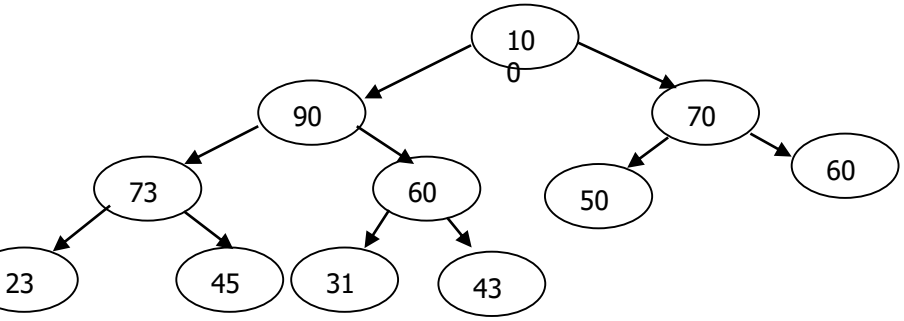
ĐSâu	#dãy	size
0	1	***n***
1	2	***n***/2
i	2^i	***n***/2^i
…	…	…



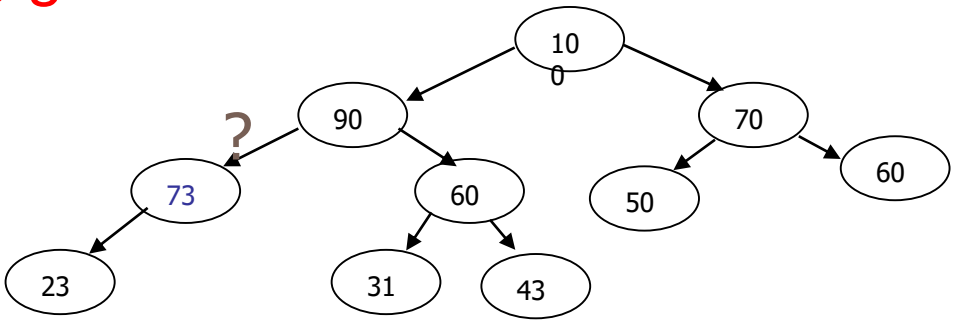
Cây Heap và Thuật toán sắp xếp vun đống Heapsort

- Cây heap (đống) là một cây nhị phân được sắp xếp theo khóa của các nút với các tính chất sau:
  - Giá trị khóa của nút gốc ≥ giá trị khóa của hai con
  - Tất cả các mức đều đầy trừ mức thấp nhất có thể thiếu một số nút
  - Các nút lá phải xuất hiện liên tiếp từ trái qua phải
- Như vậy nút gốc có giá trị khóa lớn nhất
- Ví dụ:

Bổ sung phần insert heap, get max



Cây Heap



Không phải cây Heap

Mảng biểu diễn cây heap

- Mảng A[1],..,A[n] là mảng biểu diễn cây heap nếu:
  - A[i]≥A[2i] và A[i]≥A[2i+1] với i=1..n/2
- Như vậy phần tử đầu của mảng có giá trị lớn nhất
- Ví dụ:

A	100	90	70	73	60	50	60	23	45	31	43

A[1] ≥ A[2], A[1] ≥A[3]

A[2] ≥ A[4], A[2] ≥A[5]

A[3] ≥ A[6], A[3] ≥A[7]

A[4] ≥ A[8], A[4] ≥A[9]

A[5] ≥ A[10], A[5] ≥A[11]

Thuật toán sắp xếp vun đống

Ý tưởng:

- Tạo mảng A[1],..,A[n] biểu diễn cây Heap.
- Tráo đổi phần tử A[1] với phần tử A[n].
- Tạo mảng A[1],..,A[n-1] biểu diễn cây heap
- Tráo đổi phần tử A[1] với phần tử A[n-1].
- Lặp lại quá trình trên đến khi mảng chỉ còn 1 phần tử

Tạo đống

?

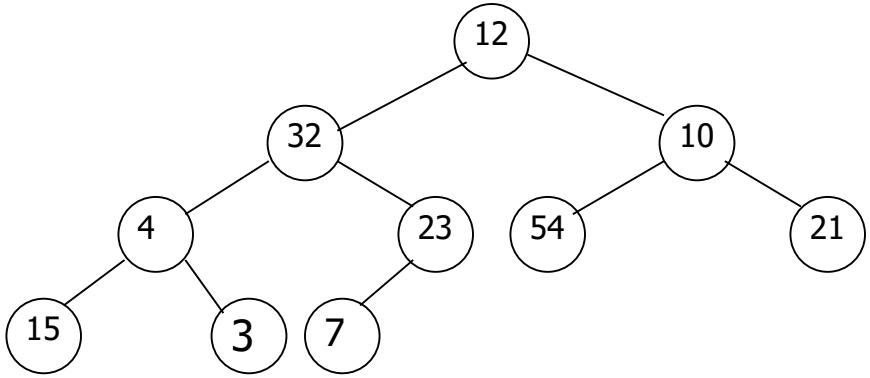
Tạo mảng biểu diễn cây heap

- Theo tính chất của mảng biểu diễn cây Heap thì các phần tử từ n/2+1 đến n không cần điều kiện ràng buộc. Vì vậy ta thực coi các phần tử này đã thỏa mãn điều kiện cây heap.
- Ta thực hiện:
  - Bổ sung phần tử n/2 vào A[n/2+1],..,A[n] để được mảng gồm A[n/2],..,A[n] thỏa mãn kiện
  - Bổ sung phần tử n/2-1 vào A[n/2],..,A[n] để được mảng gồm A[n/2-1] ,..,A[n] thỏa mãn kiện
  - Và cứ tiếp tục làm như vậy cho đến khi bổ sung phần tử A[1] vào A[2],..,A[n] để được mảng gồm A[1],..,A[n] thỏa mãn điều kiện

Ví dụ

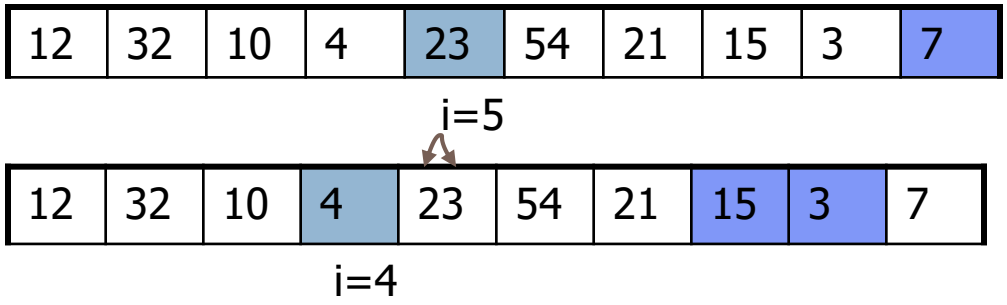
Cho mảng như dưới đây, hãy biến đổi mảng để được mảng thỏa mãn tính chất mảng biểu diễn cây heap

12	32	10	4	23	54	21	15	3	7



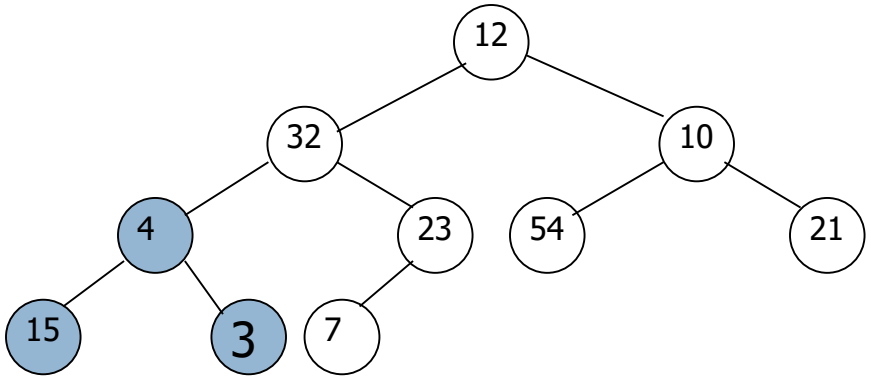
Cây tương ứng với mảng

Mô tả trên mảng: *N=10*

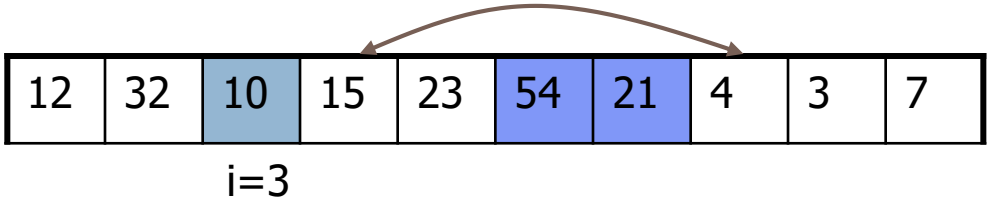


- Đổi chỗ A[5] và A[10] nếu A[5]<A[10]

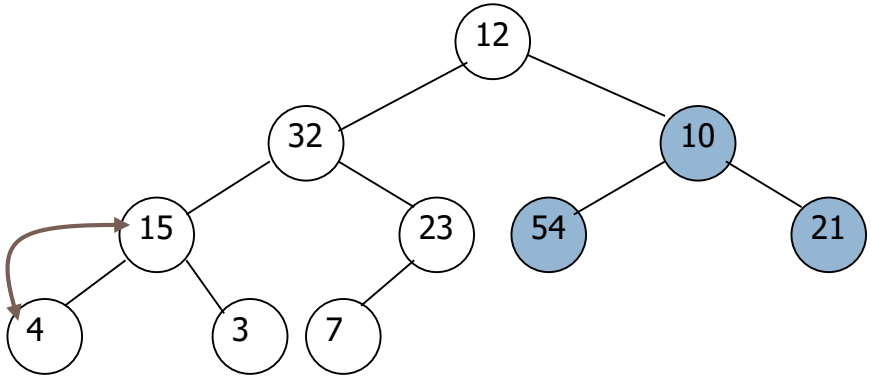
- Tính max(A[8], A[9]). Nếu A[4]<max thì đổi chỗ A[4] với phần tử đạt max



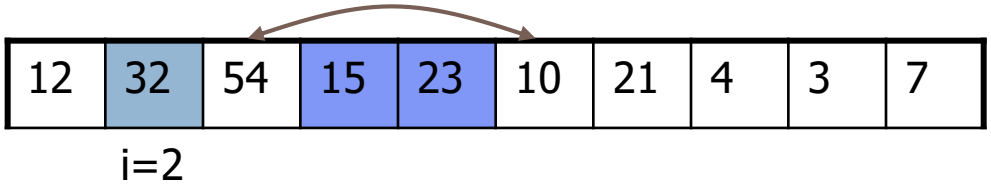
Cây tương ứng với mảng



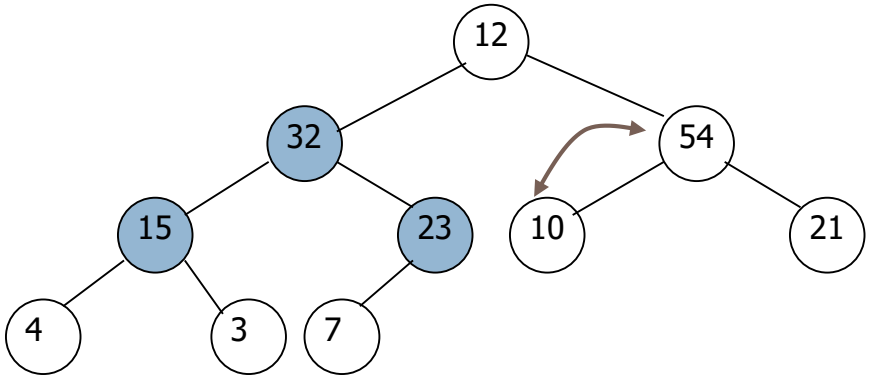
- Tính max(A[6], A[7]). Nếu A[3]<max thì đổi chỗ A[3] với phần tử đạt max



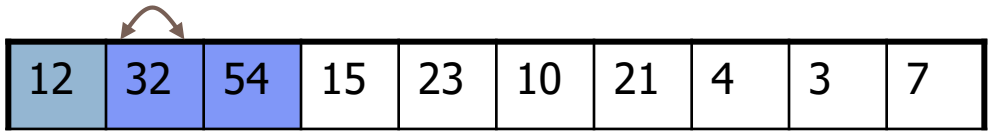
Cây tương ứng với mảng



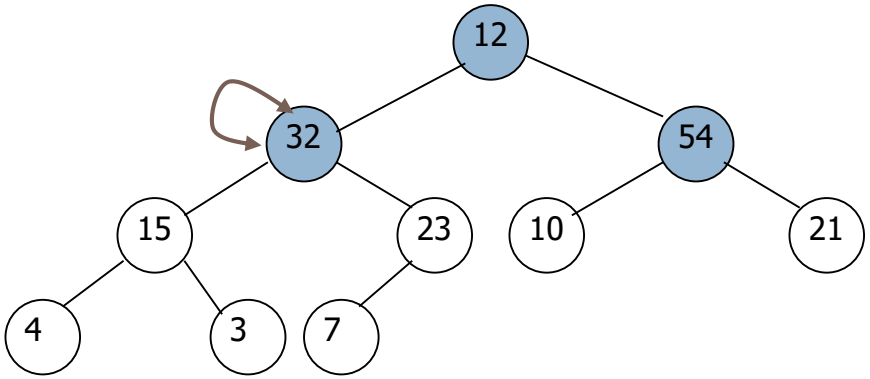
- Tính max(A[4], A[5]). Nếu A[2]<max thì đổi chỗ A[2] với phần tử đạt max



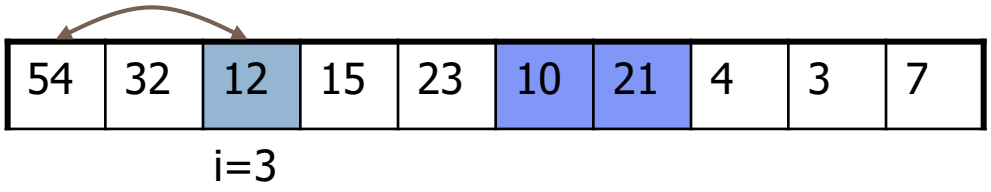
Cây tương ứng với mảng



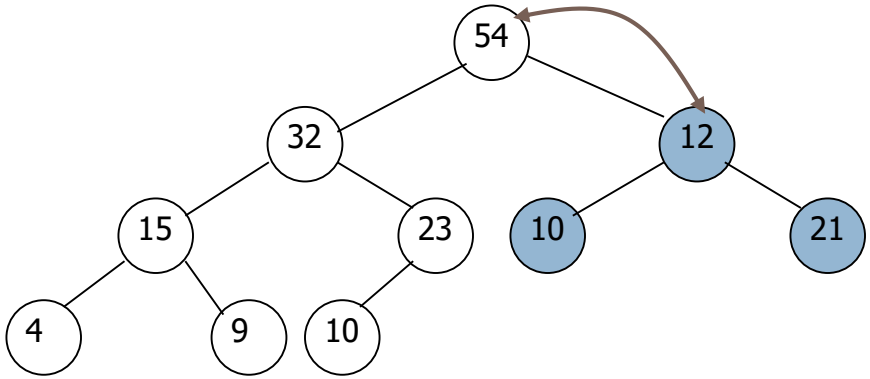
- Tính max(A[2], A[3]). Nếu A[1]<max thì đổi chỗ A[1] với phần tử đạt max



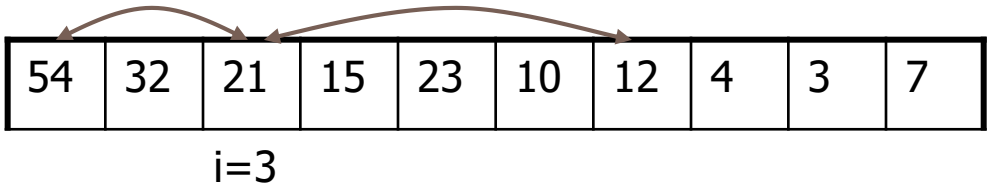
Cây tương ứng với mảng



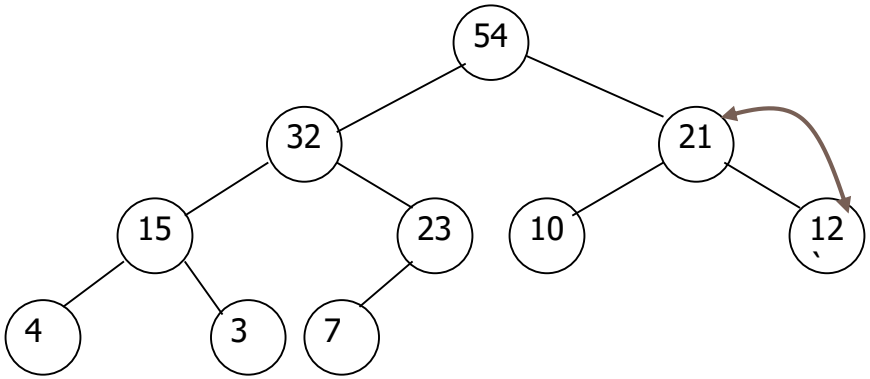
- Tính max(A[6], A[7]). Nếu A[3]<max thì đổi chỗ A[3] với phần tử đạt max



Cây tương ứng với mảng



- Tính max(A[6], A[7]). Nếu A[3]<max thì đổi chỗ A[3] với phần tử đạt max



Cây heap

Thuật toán bổ sung một phần tử để tạo mảng biểu diễn cây heap

Algorithm ***Pushdown (Array A, i, n);***

Input: số nguyên i, n, mảng A[i],..,A[n], trong đó A[i+1],..,A[n] thỏa mãn tính chất cây heap

Output: Mảng A[i],..,A[n] thỏa mãn tính chất cây heap

```
j ← i; kt ← 0;
while (j≤ n/2) and (kt=0) do
    if 2*j = n then
        max ← 2*j;
    else
        if A[2*j].key ≤ A[2*j+1].key then
            max ← 2*j+1
        else
            max ← 2*j;
    if A[j].key < A[max].key then
        swap (A[j], A[max]);
        j ← max;
    else
        kt ←1;
```

Thuật toán sắp xếp vun đống

Algorithm ***Heapsort(Array A, n);***

Input: Mảng A có n phần tử và số nguyên n

Output: Mảng A được sắp theo thứ tự tăng dần của thuộc tính khóa

```
for i← n/2 downto 1 do
    Pushdown(A, i, n);
for i← ndownto 2 do
    swap(A[1],A[i]);
    Pushdown(A,1,i-1);
```

Ví dụ:

Mô tả quá trình sắp xếp của dãy số

12 43 11 34 23 43 12 435

Thời gian chạy

- Thời gian thực hiện thủ tục Pushdown.
  - Là t/g thực hiện của vòng lặp while.
  - Gọi k là số lần lặp, ta có $i*2^k \leq n$ hay $k \leq \log_2(n/i)$.
  - T/g thực hiện hàm Pushdown (A,i, n) là 0(log(n/i)
- Xét thủ tục HeapSort
  - Vòng lặp for đầu có số lần lặp là n/2
  - Mỗi lần gọi hàm Pushdown 1 lần. Do đó t/g thực hiện là $0(\log_2 n)$.
  - Tương tự, vòng lặp for thứ 2 có số lần lặp là n-1. $0(n\log_2 n)$.
  - Vì vậy t/g thực hiện HeapSort là $O(n\log_2 n)$.