CHƯƠNG 1_2: SẮP XẾP (SORTING)

Nội dung

- Tổng quan
- Các phương pháp sắp xếp thông dụng

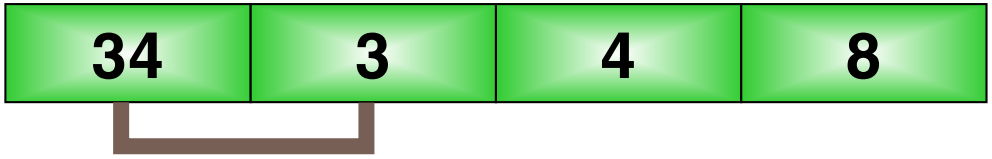
Tổng quan

- 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 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
  - Ví dụ:
    - Sắp xếp danh sách lớp học tăng theo điểm trung bình.
    - Sắp xếp danh sách sinh viên tăng theo tên.

Khái niệm nghịch thế

- 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

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



a[0], a[1] là cặp nghịch thế

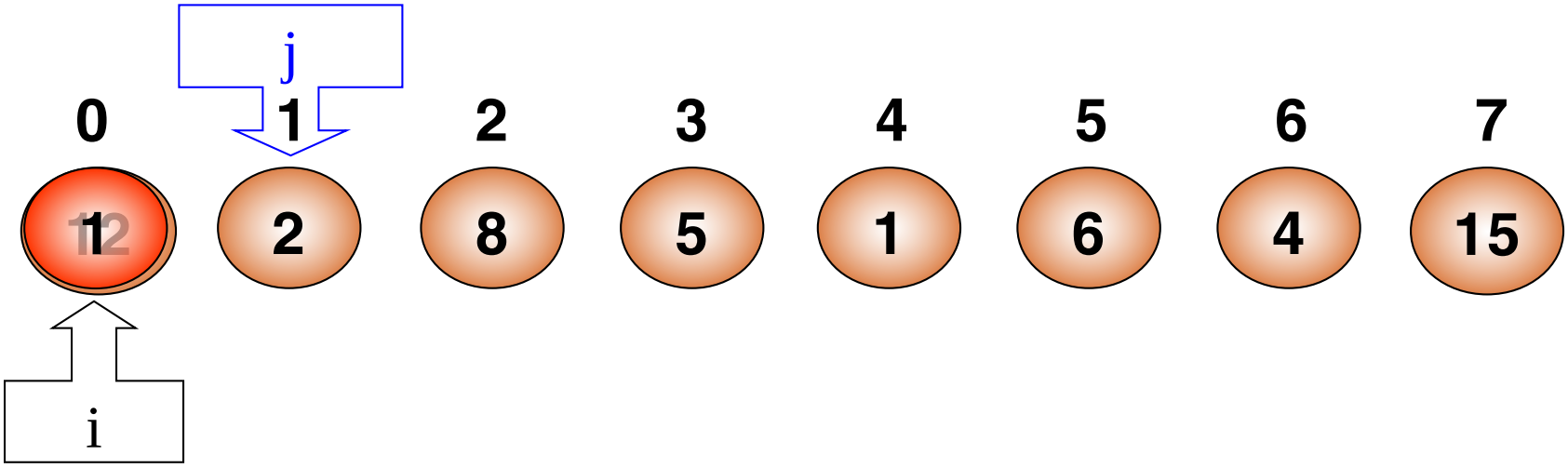
Các phương pháp sắp xếp thông dụng

1. Phương pháp Đổi chỗ trực tiếp (Interchange sort)
2. Phương pháp Nổi bọt (Bubble sort)
3. Phương pháp Chèn trực tiếp (Insertion sort)
4. Phương pháp Chọn trực tiếp (Selection sort)
5. Phương pháp dựa trên phân hoạch (Quick sort)
6. Phương pháp sắp xếp vun đống Heapsort
7. Phương pháp sắp xếp MergeSort
8. Phương pháp ShellSort

Interchange Sort – Ý tưởng

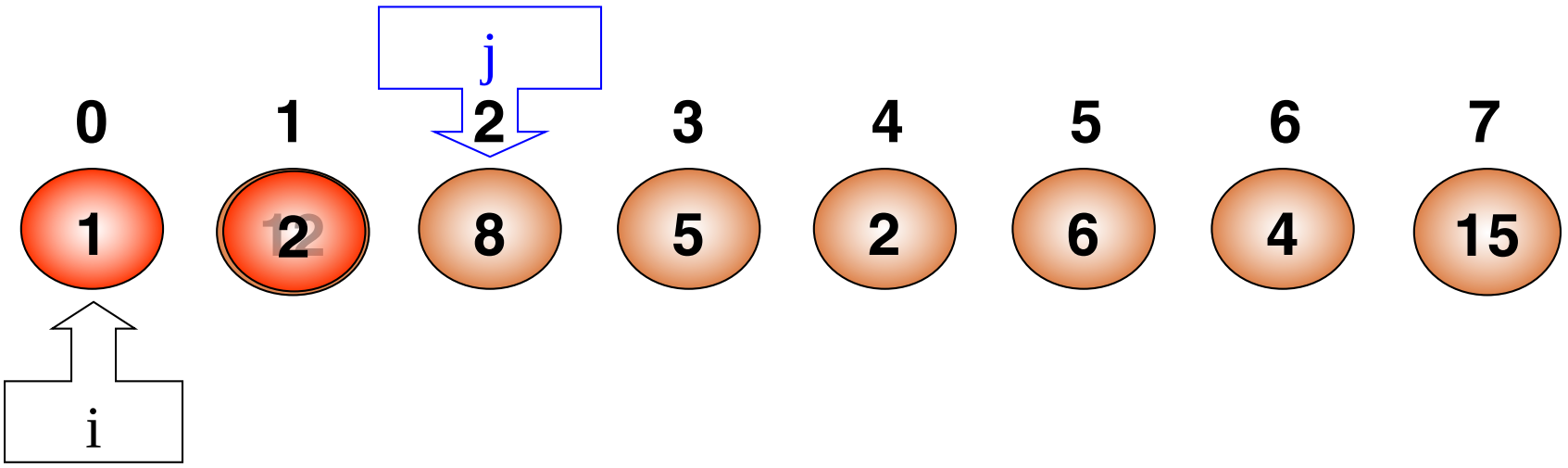
- **Nhận xét:**
  - Để sắp xếp một dãy số, ta có thể xét các nghịch thế có trong dãy và làm triệt tiêu dần chúng đi.
- **Ý tưởng:**
  - Xuất phát từ đầu dãy, tìm tất cả nghịch thế chứa phần tử này, triệt tiêu chúng bằng cách đổi chỗ phần tử này với phần tử tương ứng trong cặp nghịch thế
  - Lặp lại xử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

Interchange Sort – Ví dụ



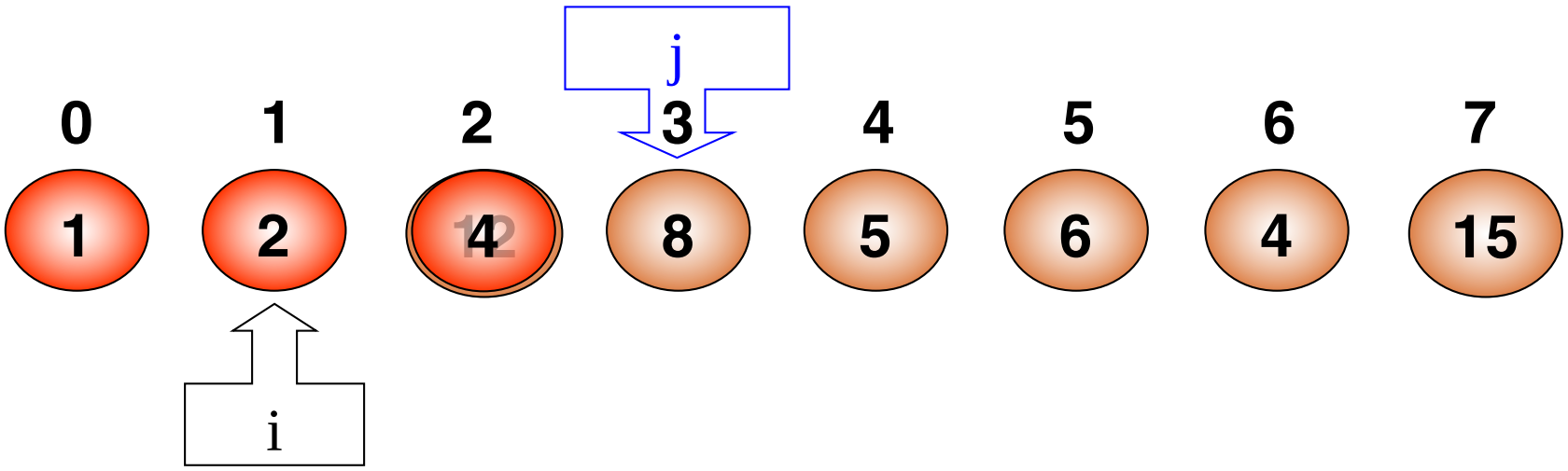
Nếu a[i] > a[j] thì đổi chỗ a[i], a[j]

Interchange Sort – Ví dụ



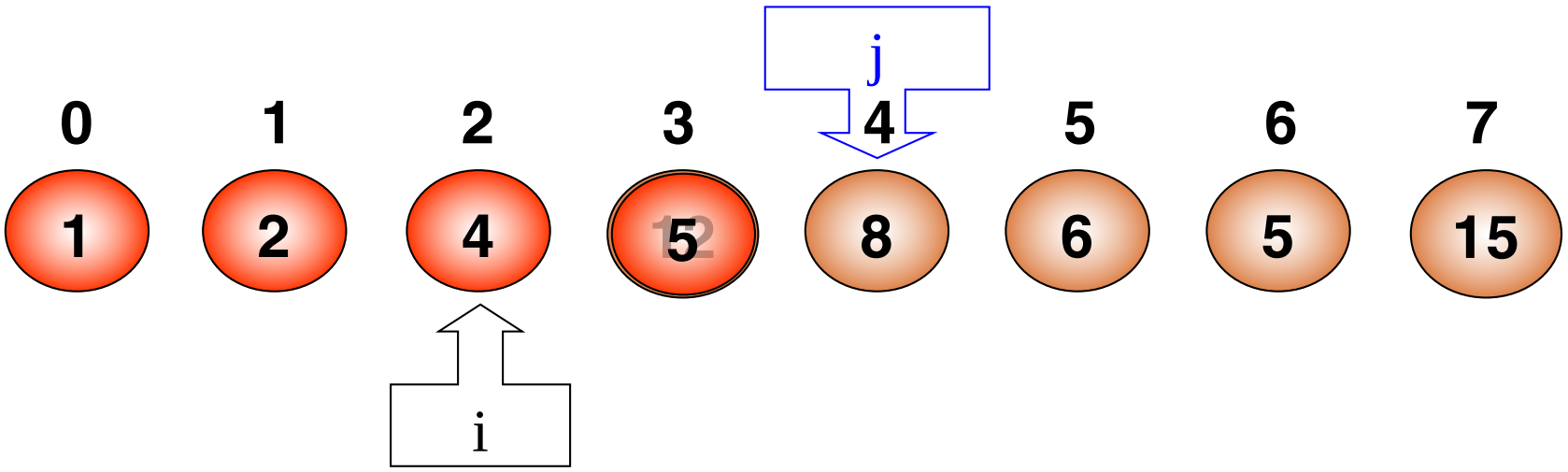
Nếu a[i] > a[j] thì đổi chỗ a[i], a[j]

Interchange Sort – Ví dụ



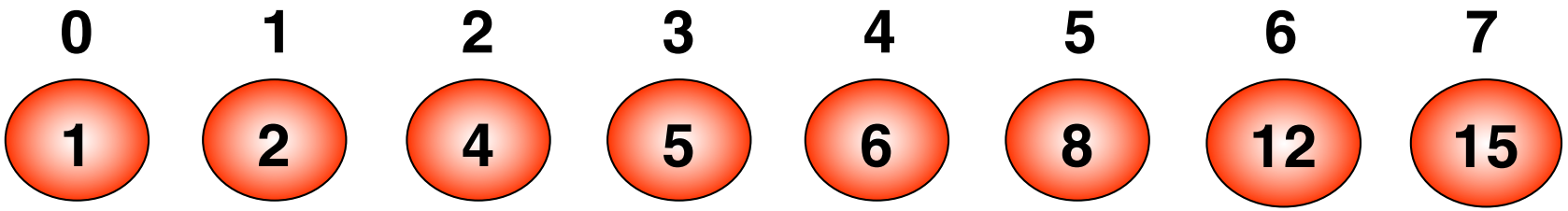
Nếu a[i] > a[j] thì đổi chỗ a[i], a[j]

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 – 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 - 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]) //nếu có nghịch thế thì đổi chỗ
           Swap(a[i], a[j]);
}
void Swap(int &a, int &b)
{
      int  temp = a;
             a = b;
             b = temp;
}
```

Interchange Sort - Cài đặt

Chú ý


```
1  print("1. Interchange Sort")
2  a=[6,5,3,1,8,7,2,4]
3  print("Mảng chưa sắp xếp:")
4  print(a)
5  def InterchangeSort(a):
6      n=len(a)
7      for i in range(n):
8          for j in range(i+1,n,1):
9              if a[i]>a[j]:
10                 a[i],a[j]=a[j],a[i]
11     return a
12 print("Mảng đã sắp xếp")
13 print(InterchangeSort(a))
14
```

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 phép so sánh**: Ở mỗi lượt thứ i, luôn có (n-i) lần so sánh **a[i]>a[j]**. Số lần so sánh là:

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

- Số lượng phép hoán vị thực hiện tùy thuộc vào kết quả so sánh. Độ phức tạp của thuật toán là $O(n^2)$

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

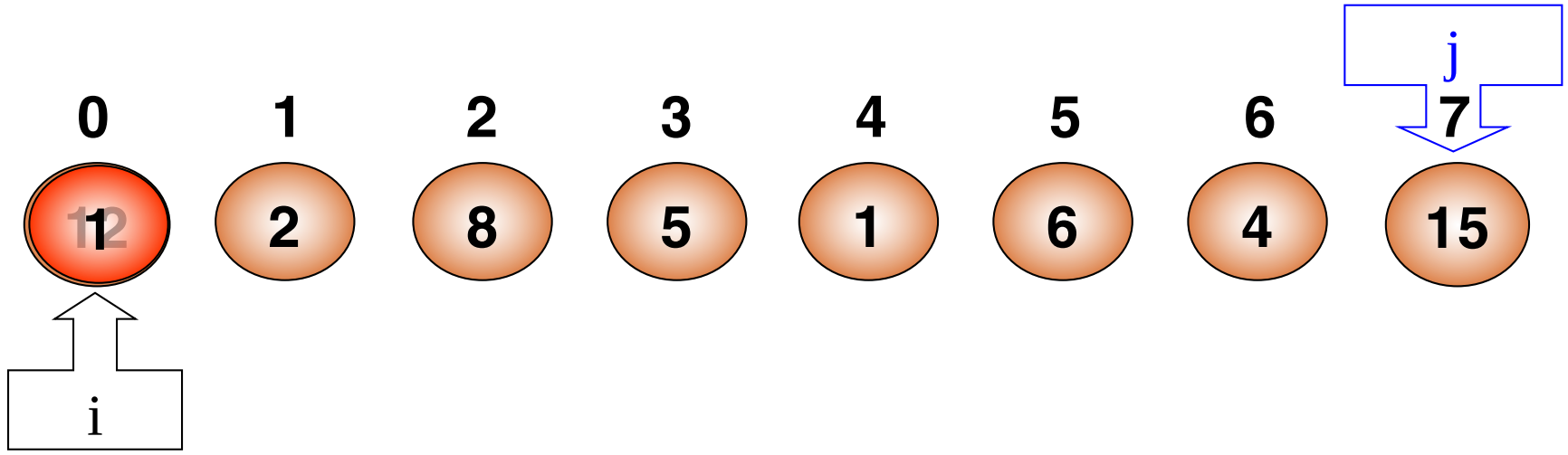
Các phương pháp sắp xếp thông dụng

- Phương pháp Đổi chỗ trực tiếp (Interchange sort)
- Phương pháp Nổi bọt (Bubble sort)
- Phương pháp Chèn trực tiếp (Insertion sort)
- Phương pháp Chọn trực tiếp (Selection sort)
- Phương pháp dựa trên phân hoạch (Quick sort)

Bubble Sort – Ý tưởng

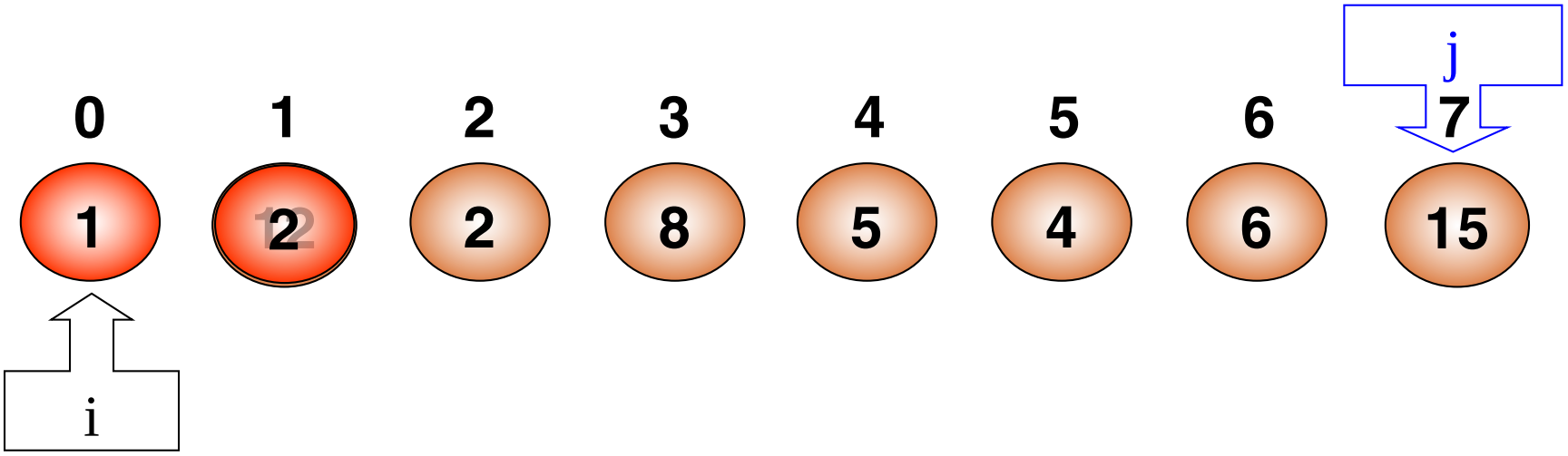
- Xuất phát từ cuối dãy, đổi chỗ các cặp phần tử kế cận để đưa phần tử nhỏ hơn trong cặp phần tử đó về vị trí đầu dãy hiện hành, sau đó sẽ không xét đến nó ở bước tiếp theo
- Ở lần xử lý thứ i có vị trí đầu dãy là i
- Lặp lại xử lý trên cho đến khi không còn cặp phần tử nào để xét

Bubble Sort – Ví dụ



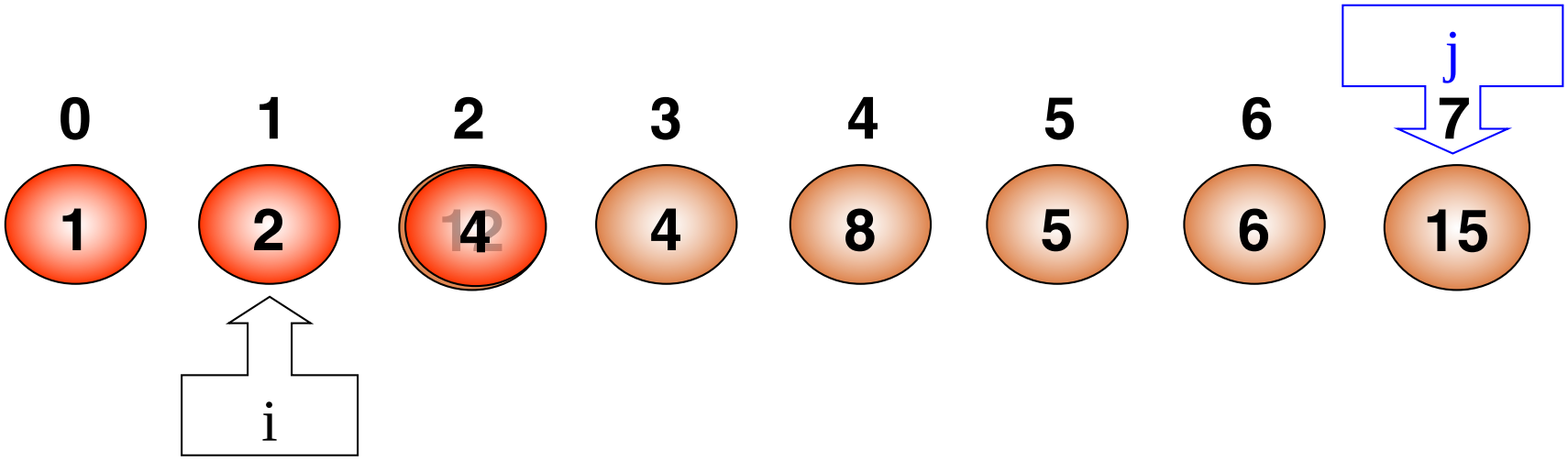
Nếu a[j]<a[j-1] thì đổi chỗ a[j], a[j-1]

Bubble Sort – Ví dụ



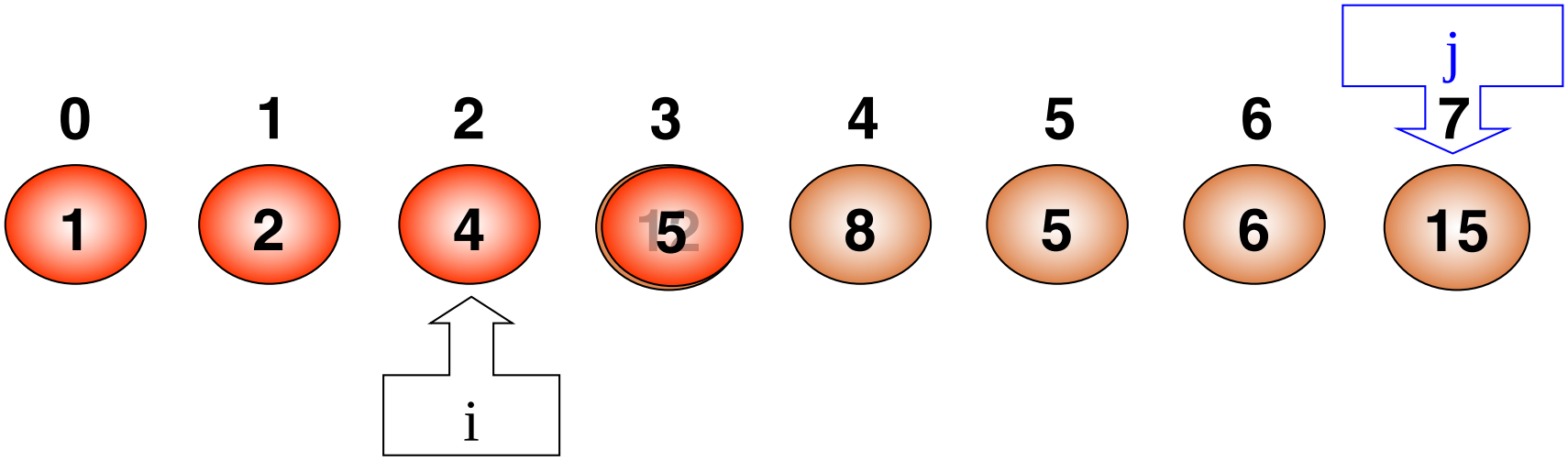
Nếu a[j]<a[j-1] thì đổi chỗ a[j], a[j-1]

Bubble Sort – Ví dụ



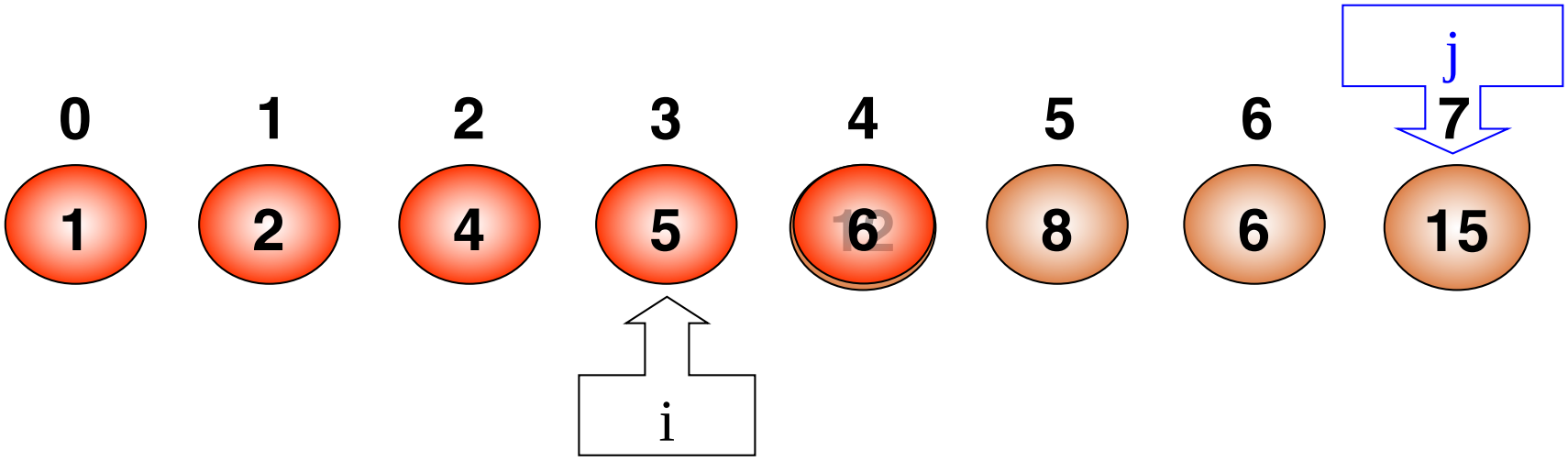
Nếu a[j]<a[j-1] thì đổi chỗ a[j], a[j-1]

Bubble Sort – Ví dụ



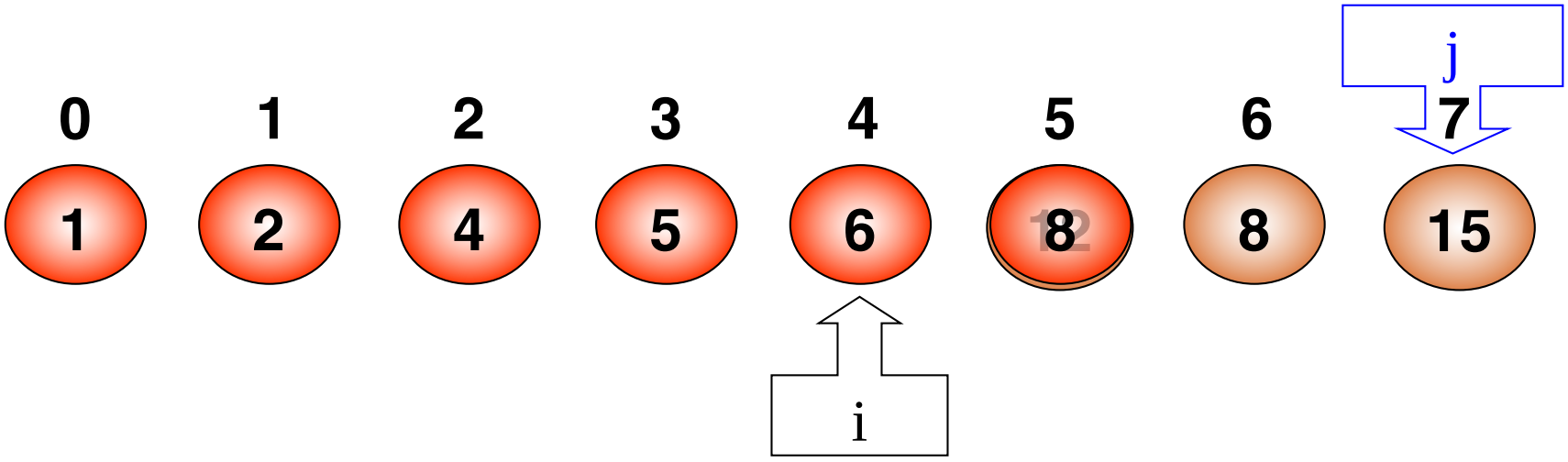
Nếu a[j]<a[j-1] thì đổi chỗ a[j], a[j-1]

Bubble Sort – Ví dụ



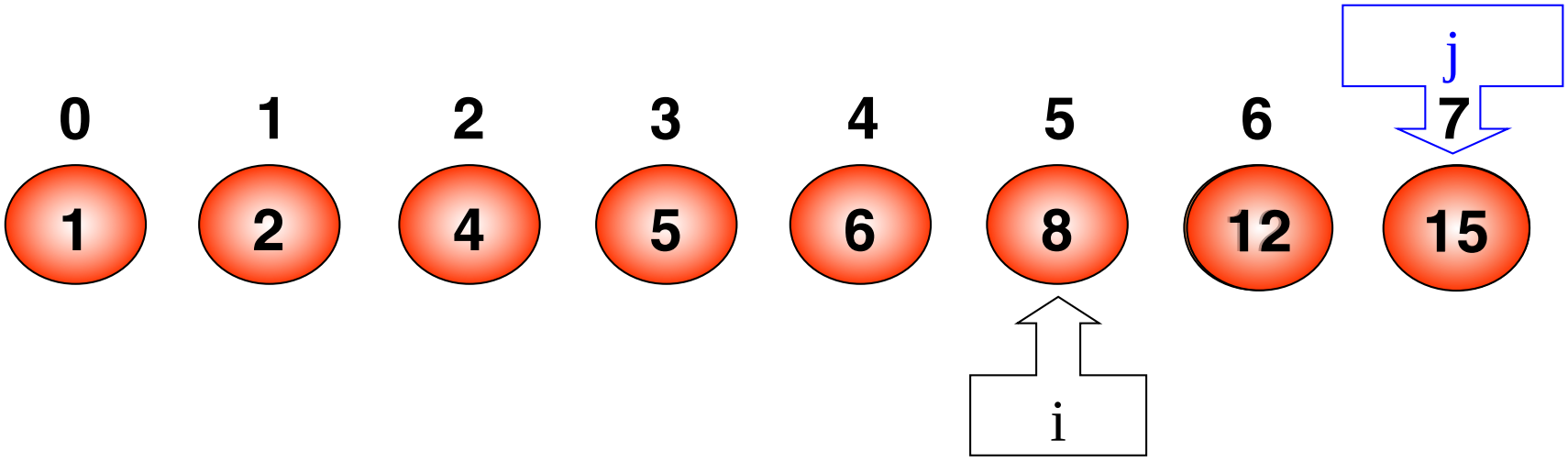
Nếu a[j]<a[j-1] thì đổi chỗ a[j], a[j-1]

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 – 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-1: Dừng *// Hết dãy*
  - Ngược lại: Lặp lại Bước 2

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 - Cài đặt


```
a=[6,5,3,1,8,7,2,4]
print(a)
def BubbleSort(a):
  n=len(a)
  for i in range(n):
     for j in range(0,n-i-1):
        if(a[j] > a[j+1])
          a[j],a[j+1]=a[j+1],a[j]
  return a
```

Cách thực hiện:

So sánh hai phần tử liền kề(i,i+1) trong mảng, nếu a[i]>a[i+1] thì đổi chỗ (swap)

Bubble Sort - Cài đặt

Chú ý


```
15  print("2. BubbleSort")
16  a=[6,5,3,1,8,7,2,4]
17  print("Mảng chưa sắp xếp:")
18  print(a)
19  def BubbleSort(a):
20      n=len(a)
21      for i in range(n):
22          for j in range(0,n-i-1,1):
23              if a[j]>a[j+1]:
24                  a[j],a[j+1]=a[j+1],a[j]
25      return a
26  print("Mảng đã sắp xếp")
27  print(BubbleSort(a))
```

Bubble Sort - Đánh giá giải thuật

Phân tích thuật toán:

- Trong mọi trường hợp:
  - Số phép gán: G = 0
  - Số phép so sánh: S = (N-1) + (N-2) + … + 1 = ½N(N-1)
- Trong trường hợp tốt nhất: khi mảng ban đầu đã có thứ tự tăng
  - Số phép hoán vị: Hmin = 0
- Trong trường hợp xấu nhất: khi mảng ban đầu đã có thứ tự giảm
  - Số phép hoán vị: Hmin = (N-1) + (N-2) + … + 1 = ½N(N-1)
- Số phép hoán vị trung bình: Havg = ¼N(N-1)

$$T_n = \sum_{i=2}^{n}\sum_{j=i}^{n} 1 = \sum_{i=2}^{n}(n-i) = \sum_{i=2}^{n} n - \sum_{i=2}^{n} i = n(n-1) - \frac{n(n-1)}{2} + 1 = \frac{n(n-1)}{2} + 1 = O(n^2)$$

Bubble Sort - Đánh giá giải thuật

- Độ phức tạp thời gian của Bubble Sort là O (n^2).

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
  - 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

Các phương pháp sắp xếp thông dụng

1. Phương pháp Đổi chỗ trực tiếp (Interchange sort)
2. Phương pháp Nổi bọt (Bubble sort)
3. Phương pháp Chèn trực tiếp (Insertion sort)
4. Phương pháp Chọn trực tiếp (Selection sort)
5. Phương pháp dựa trên phân hoạch (Quick sort)
6. Phương pháp sắp xếp vun đống HeapSort
7. Phương pháp sắp xếp MergeSort
8. Phương pháp sắp xếp ShellSort

Phương pháp Chèn trực tiếp 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 (i ≥ 2)
- **Ý tưởng chính:**
  - Tìm cách chèn phần tử **a[i]** 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 vị trí giữa 2 phần tử a[pos-1] và a[pos] thỏa :

a[pos-1] ≤ a[i]< a[pos] (1≤pos≤i)

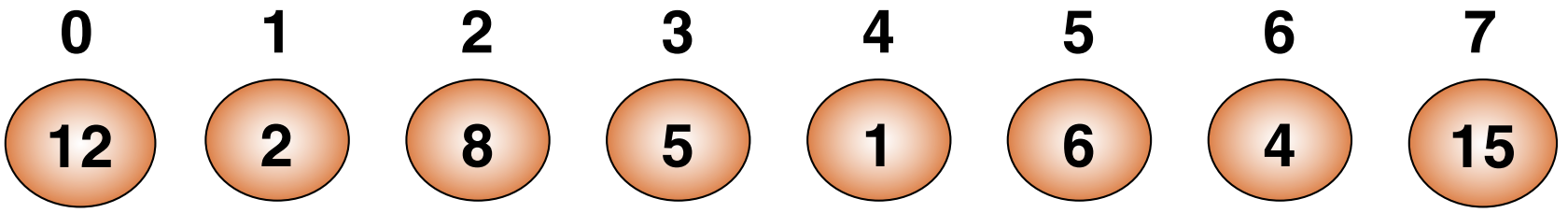
- Xem dãy con có 1 phần tử tại địa chỉ 0 là dãy đã được sắp xếp
- Lấy 1 phần tử kề sau dãy con và tìm cách chèn vào dãy con trước nó thích hợp
- Lặp lại việc chèn này tử phần tử thứ 1 đến phần tử cuối cùng.

Insertion Sort – Ý tưởng

Chi tiết hơn:

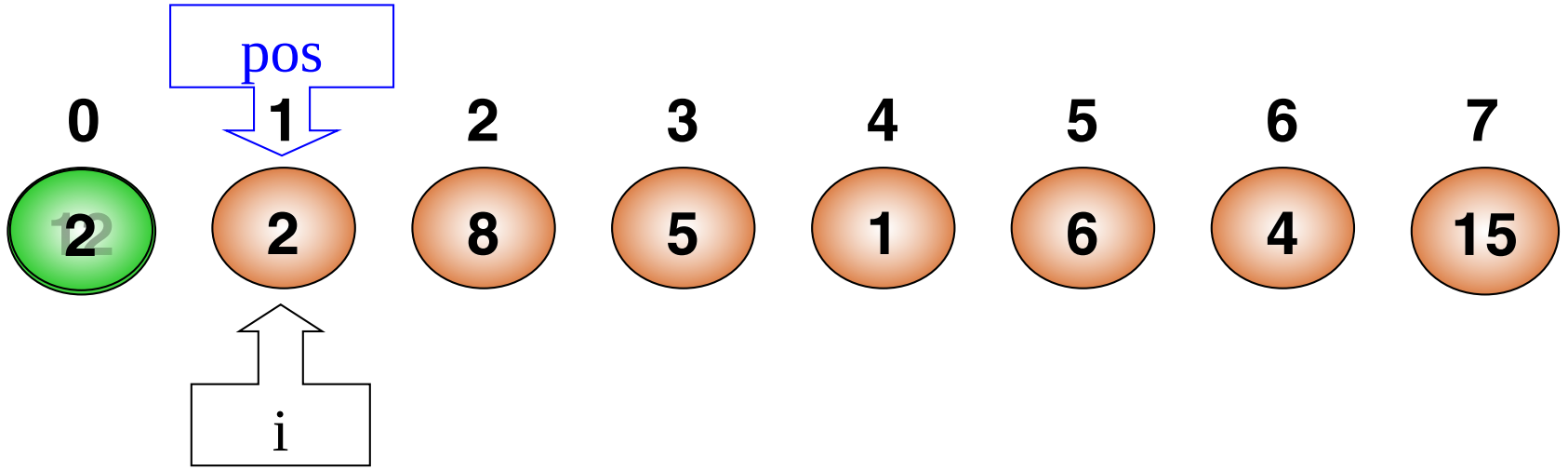
- *Dãy ban đầu* ***a[0] , a[1] ,..., a[n-1]****, xem như đã có đoạn gồm một phần tử* ***a[0]*** *đã được sắp*
- *Thêm* ***a[1]*** *vào đoạn* ***a[0]*** *sẽ có đoạn* ***a[0] a[1]*** *được sắp*
- *Thêm* ***a[2]*** *vào đoạn* ***a[0] a[1]*** *để có đoạn* ***a[0] a[1] a[2]*** *được sắp*
- *Tiếp tục cho đến 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 – Ví dụ



Insertion Sort – Ví dụ

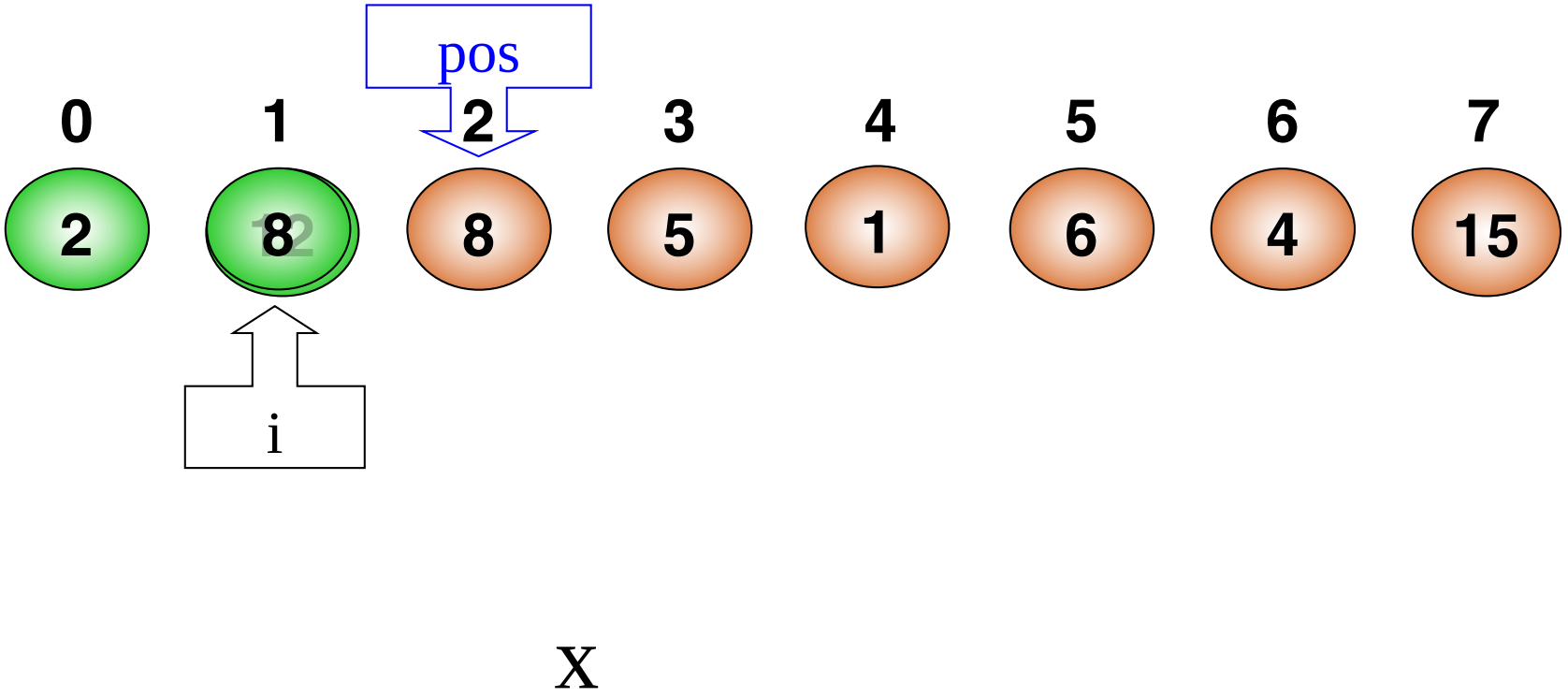
Chèn a[1] vào (a[0], a[1])



X

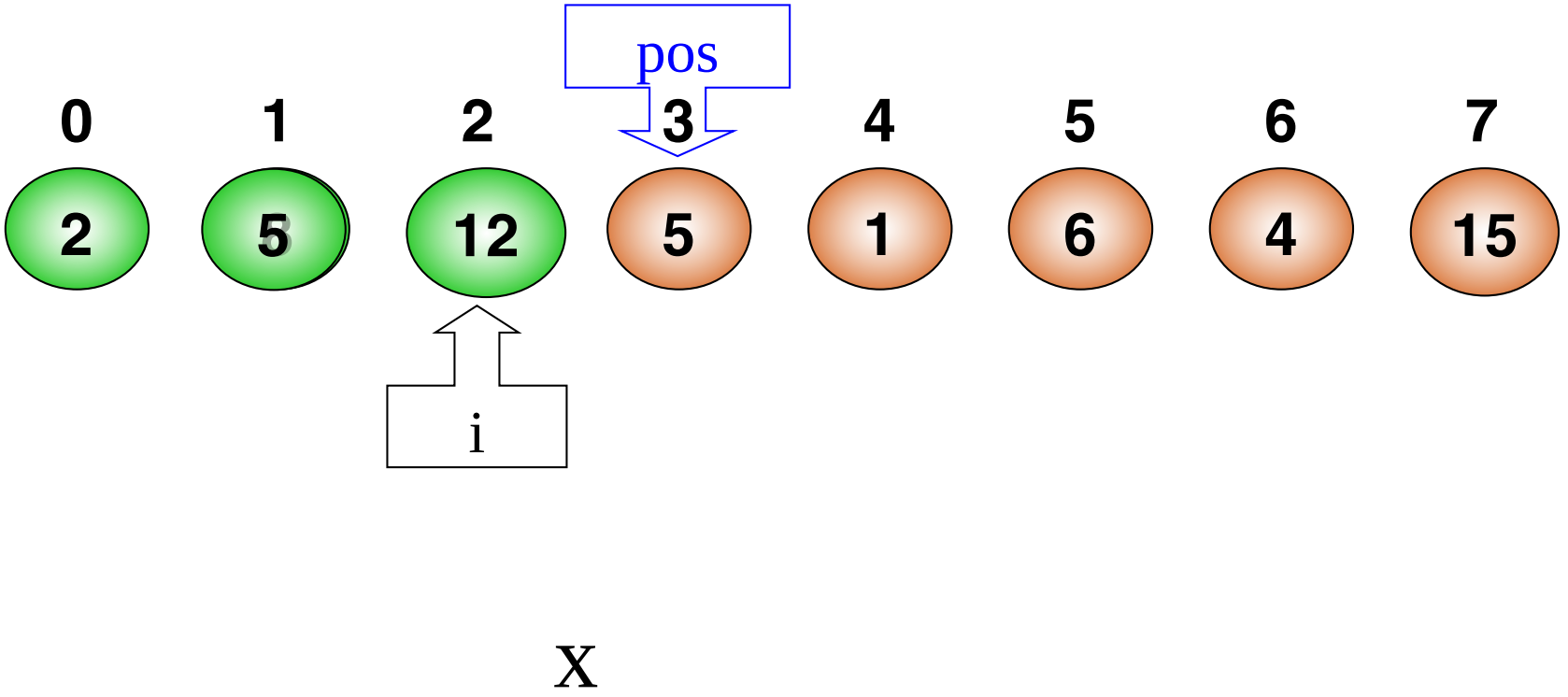
Insertion Sort – Ví dụ

Chèn a[2] vào (a[0] … a[2])



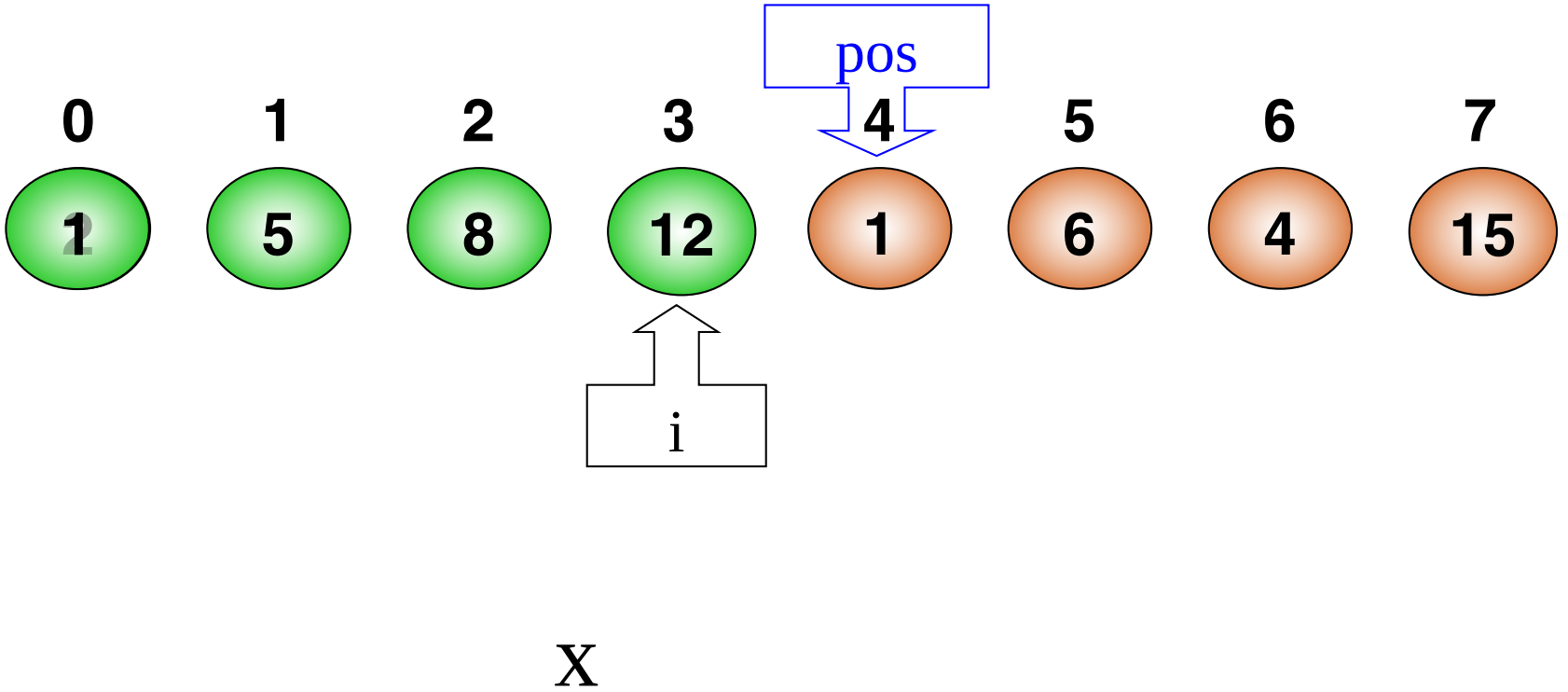
Insertion Sort – Ví dụ

Chèn a[3] vào (a[0] … a[3])



Insertion Sort – Ví dụ

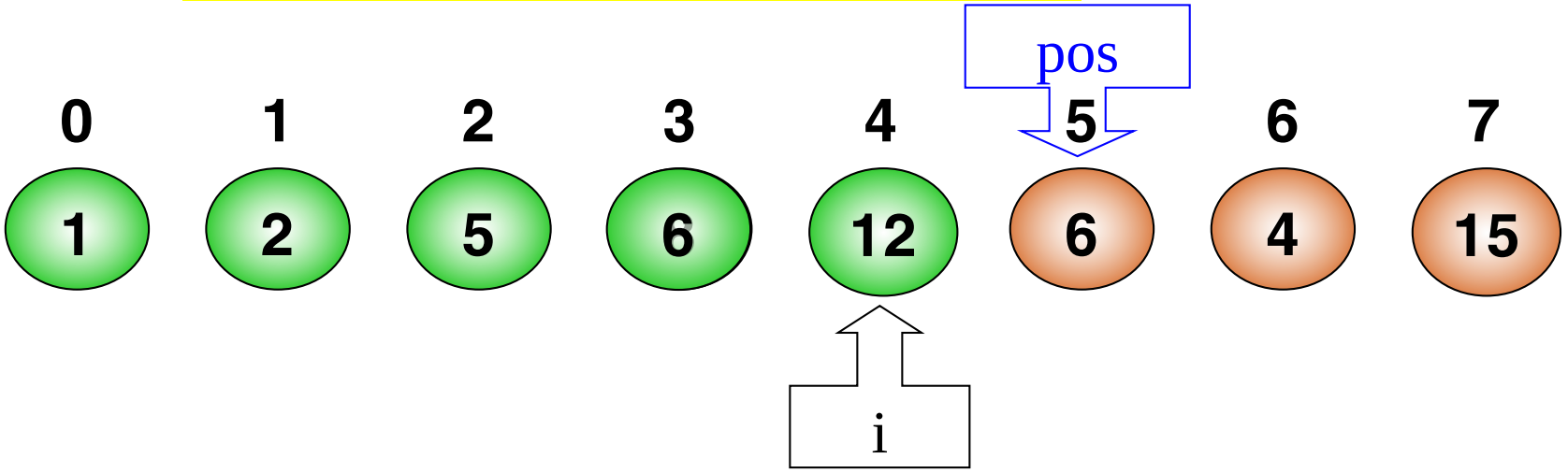
Chèn a[4] vào (a[0] … a[4])



X

Insertion Sort – Ví dụ

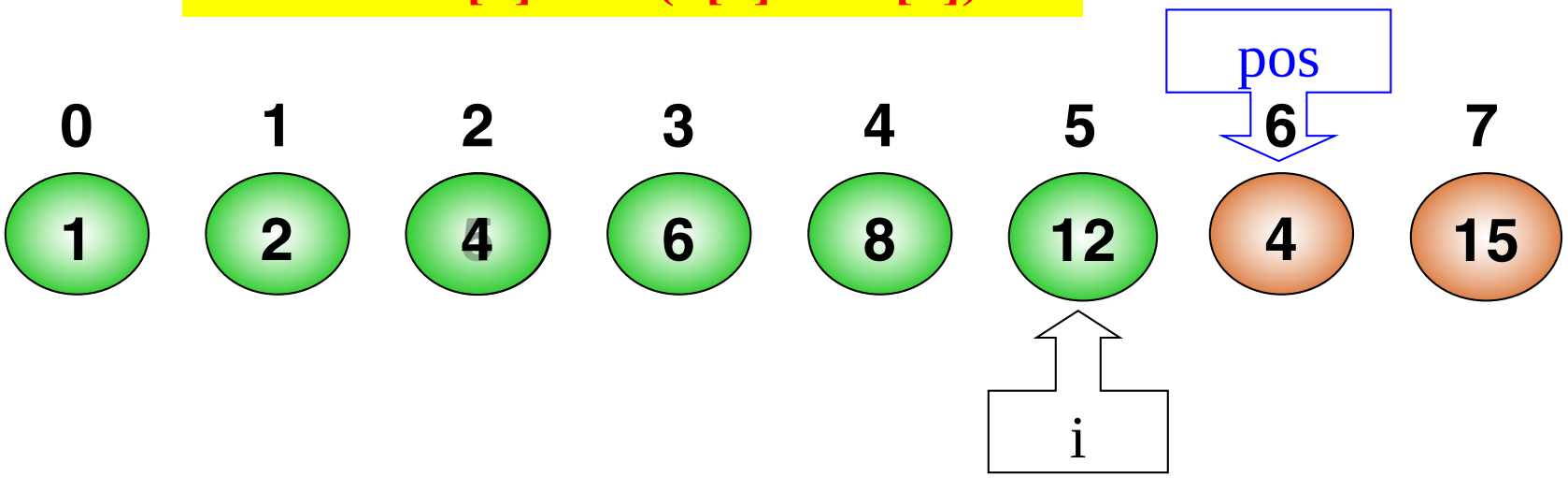
Chèn a[5] vào (a[0]… a[5])



X

Insertion Sort – Ví dụ

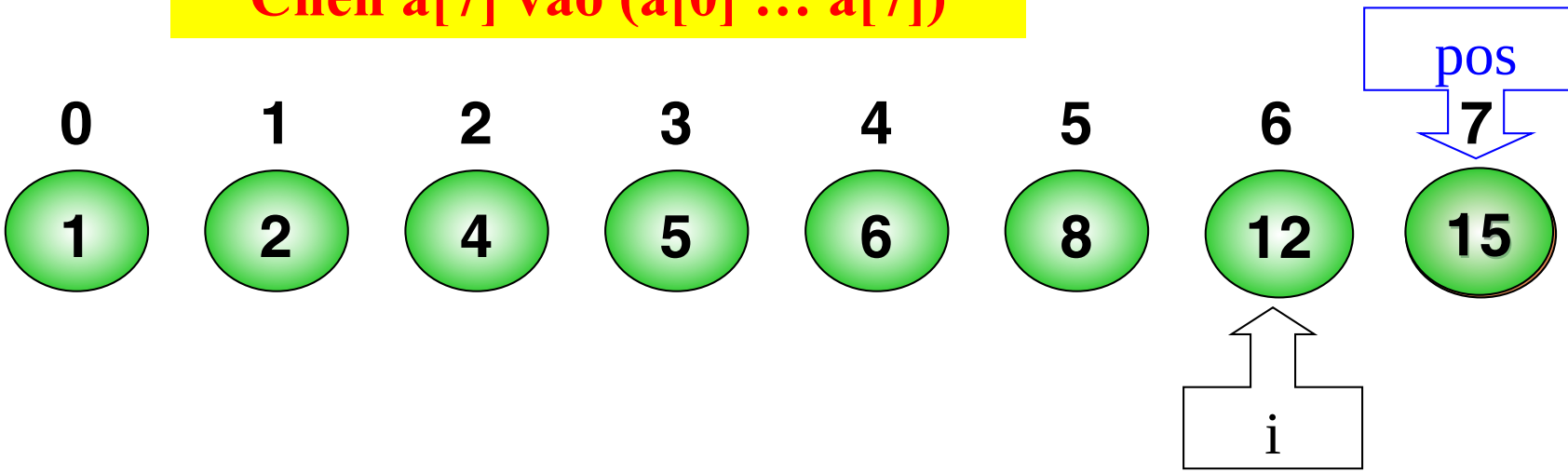
Chèn a[6] vào (a[0] … a[6])



X

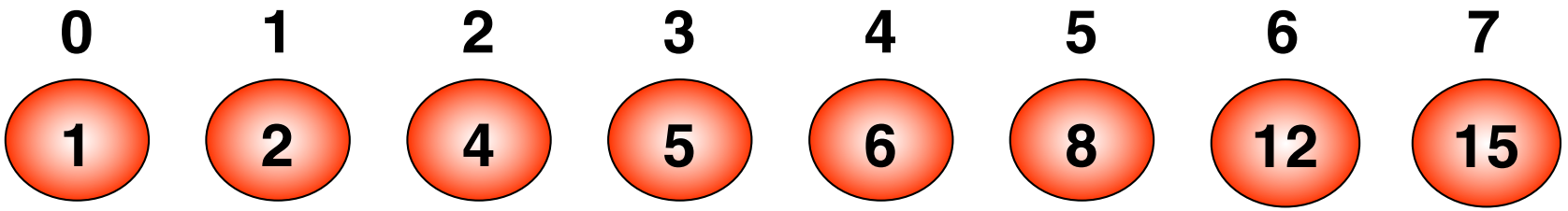
Insertion Sort – Ví dụ

Chèn a[7] vào (a[0] … a[7])



X

Insertion Sort – Ví dụ



Insertion Sort – Thuật toán

// input: dãy (a, n)

// output: dãy (a, n) đã được sắp xếp

- Bước 1: i = 1; *// 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, pos=i
- 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 – Cài đặt


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

Insertion Sort – Cài đặt

Chú ý


```
29  print("3. InsertionSort")
30  a=[6,5,3,1,8,7,2,4]
31  print("Mảng chưa sắp xếp:")
32  print(a)
33  def InsertionSort(a):
34      n=len(a)
35      for i in range(n):#từ 1 tới n-1
36          x=a[i] #phần tử hiện tại được thêm vào
37          pos =i #tìm vị trí đúng cho phần tử hiện tại
38          while pos>0 and x<a[pos-1]:
39              a[pos]=a[pos-1] #dời chỗ
40              pos =pos -1
41          a[pos] = x
42      return a
43  print("Mảng đã sắp xếp")
44  print(InsertionSort(a))
```

Insertion Sort – Đánh giá giải thuật

Phân tích thuật toán:

- Trường hợp tốt nhất, khi mảng M ban đầu đã có thứ tự tăng:
  - Số phép gán: Gmin = 2×(N-1)
  - Số phép so sánh: Smin = 1+2+…+(N-1) = N×(N-1)/2
  - Số phép hoán vị: Hmin = 0
- Trường hợp xấu nhất, khi mảng M ban đầu luôn có phần tử nhỏ nhất trong N-K phần tử còn lại đứng ở vị trí sau cùng sau mỗi lần hoán vị:
  - Số phép gán: Gmax = [2×(N-1)]+[1+2+…+(N-1)] = [2×(N-1)] + [N×(N-1)/2]
  - Số phép so sánh: Smax = (N-1)
  - Số phép hoán vị: Hmax = 0
- Trung bình:
  - Số phép gán: Gavg = 2×(N-1) + [N×(N-1)/4]
  - Số phép so sánh: Savg = [N×(N-1)/2 + (N-1)]/2 = (N+2)×(N-1)/4
  - Số phép hoán vị: Havg = 0

Insertion Sort – Đánh giá giải thuật

- Trong trường hợp xấu nhất, vòng lặp for bên trong sẽ hoán đổi một lần, sau đó hoán đổi hai lần và cứ thế. Số lượng giao dịch hoán đổi sau đó sẽ là 1 + 2 + ... + (n - 3) + (n - 2) + (n - 1)→Độ phức tạp thời gian của O (n ^ 2).

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

Các phương pháp sắp xếp thông dụng

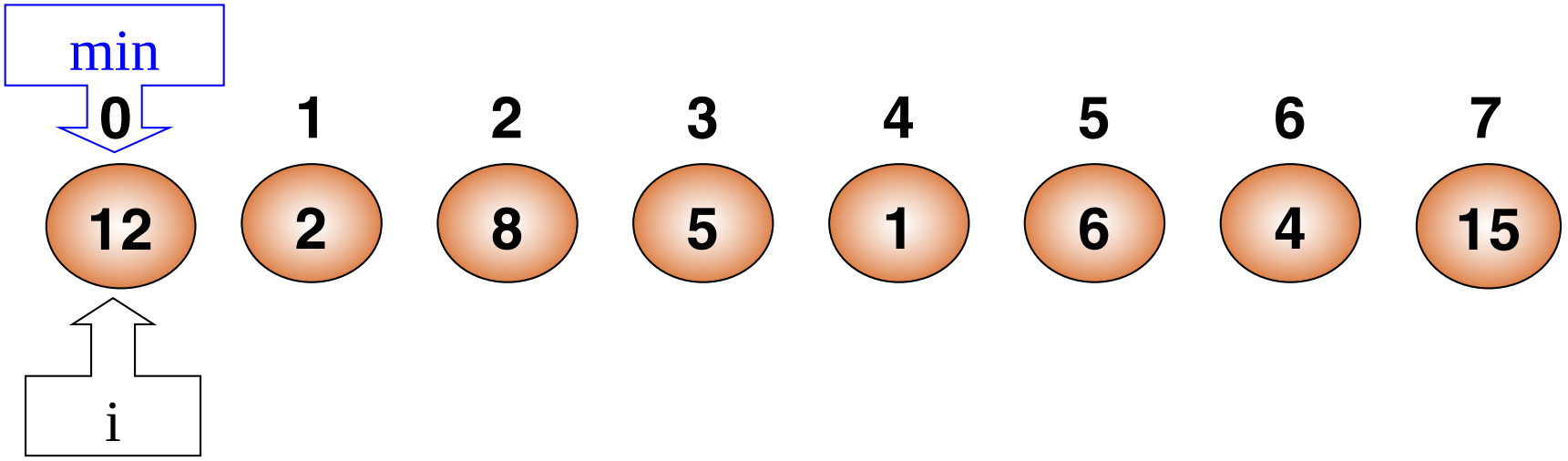
- Phương pháp Đổi chỗ trực tiếp (Interchange sort)
- Phương pháp Nổi bọt (Bubble sort)
- Phương pháp Chèn trực tiếp (Insertion sort)
- Phương pháp Chọn trực tiếp (Selection sort)
- Phương pháp dựa trên phân hoạch (Quick sort)

Selection Sort – Ý tưởng

- **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

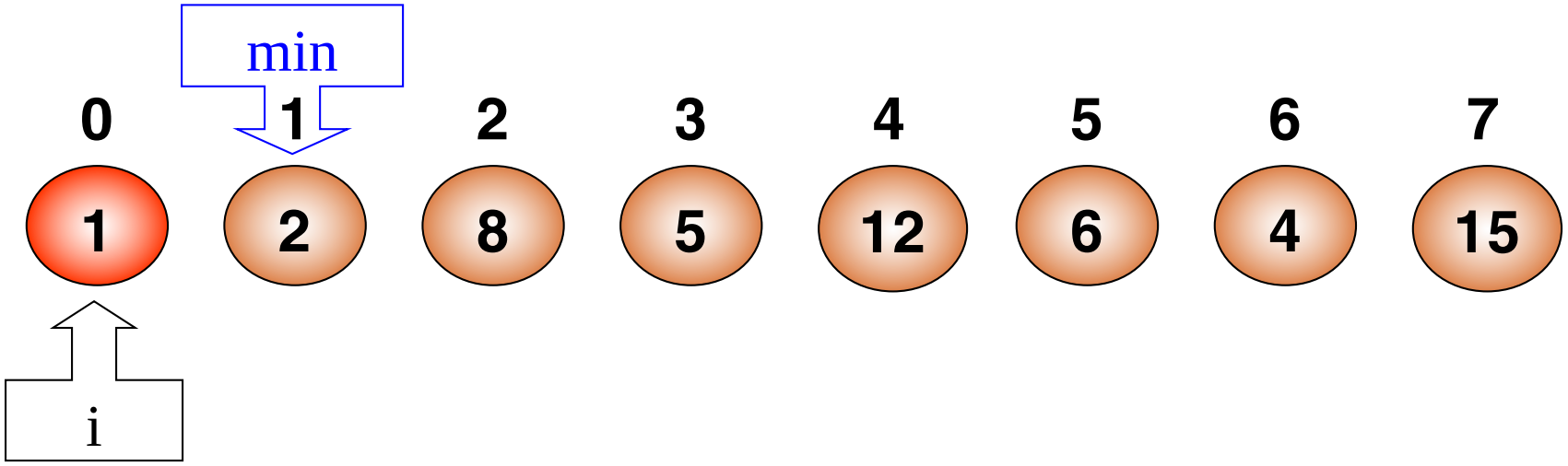
Selection Sort – Ví dụ

Find MinPos(0, 7) **Swap(a[i], a[min])**



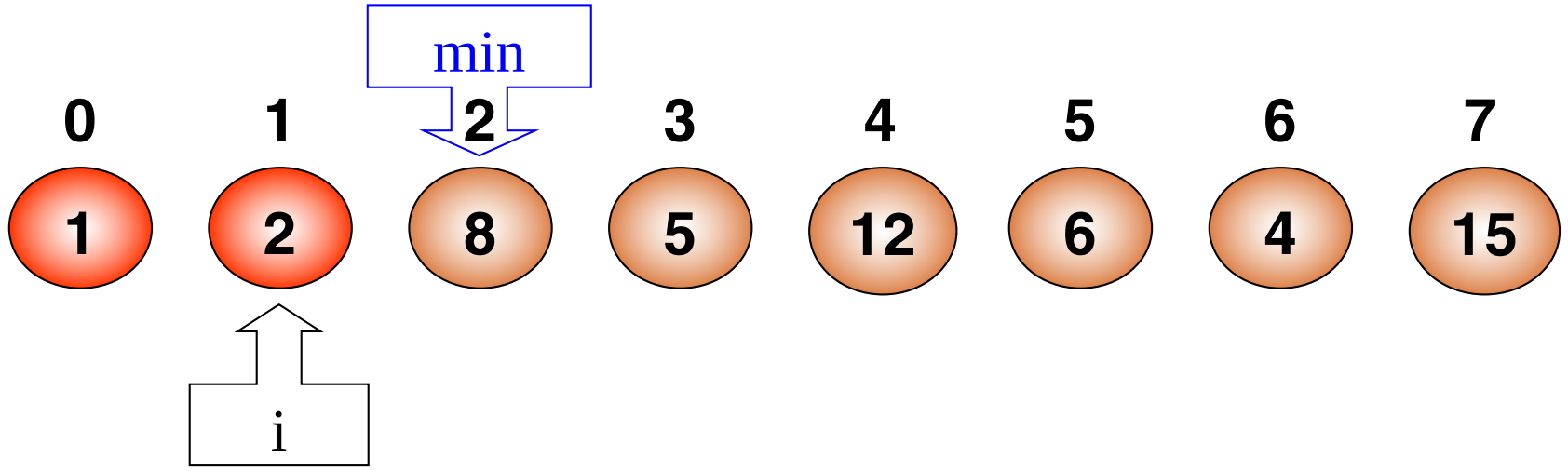
Selection Sort – Ví dụ

Find MinPos(1, 7) **Swap(a[i], a[min])**



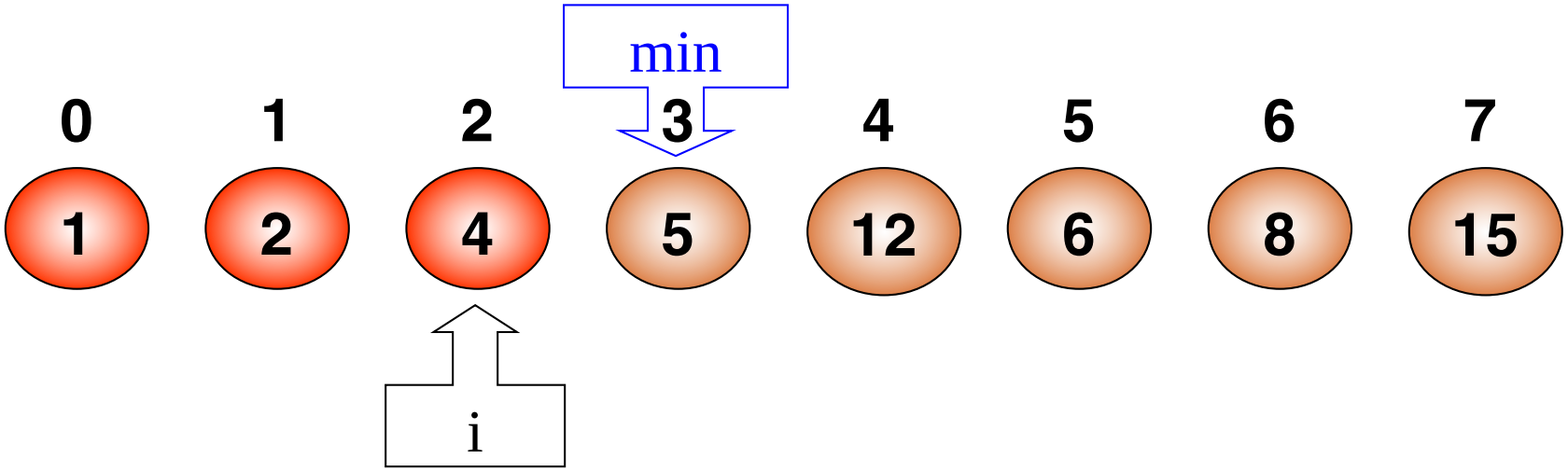
Selection Sort – Ví dụ

Find MinPos(2, 7) **Swap(a[i], a[min])**



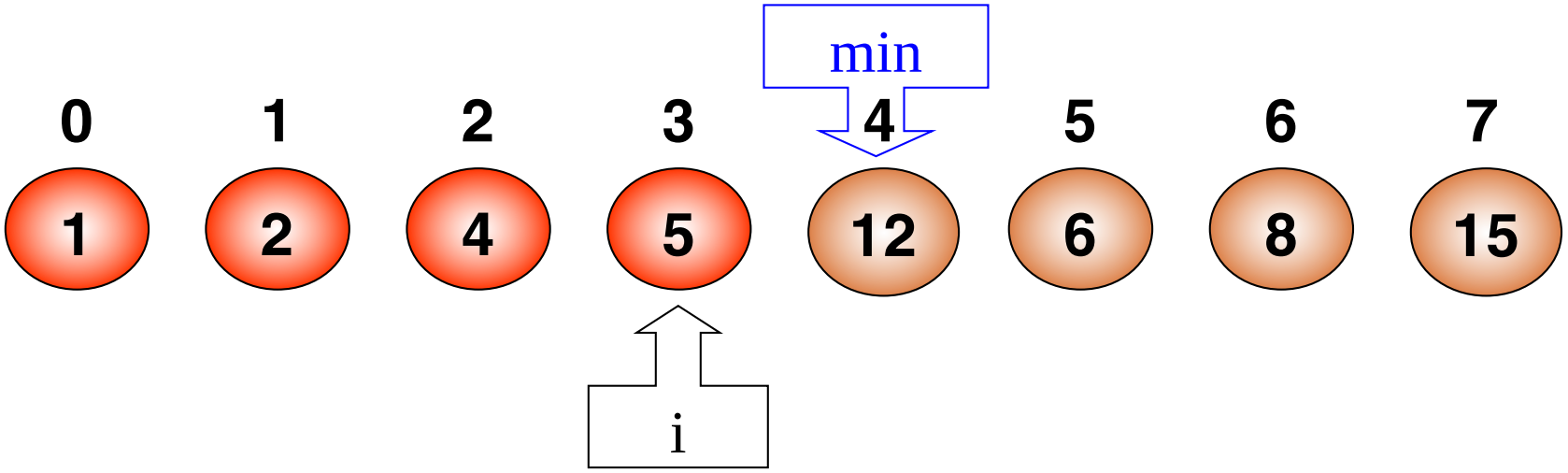
Selection Sort – Ví dụ

Find MinPos(3, 7) **Swap(a[i], a[min])**



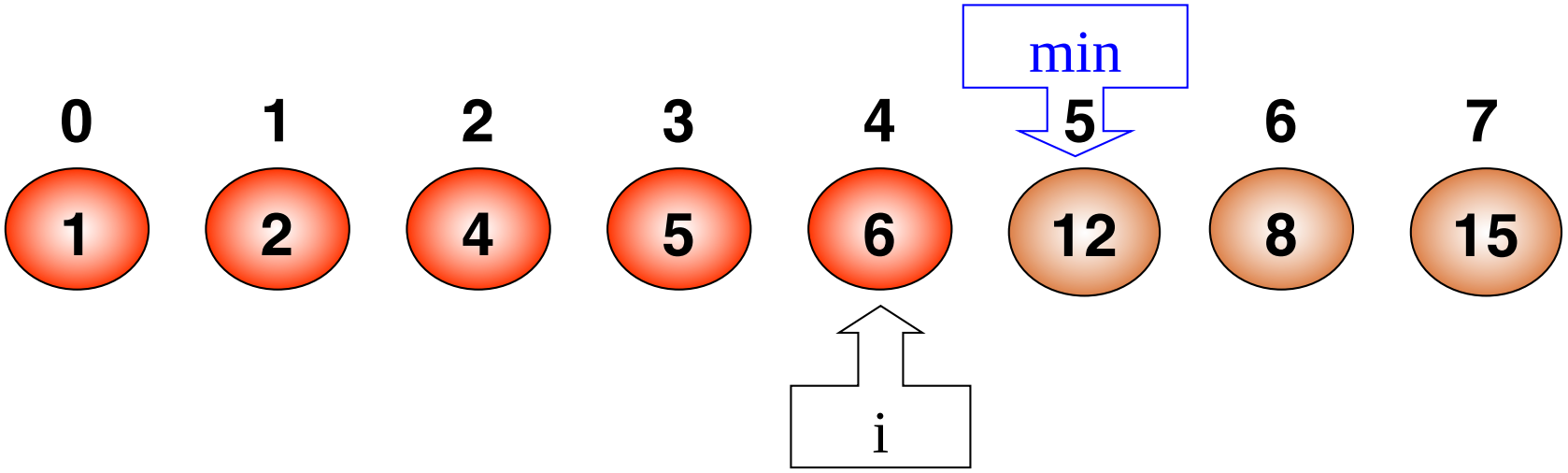
Selection Sort – Ví dụ

Find MinPos(4, 7) **Swap(a[i], a[min])**



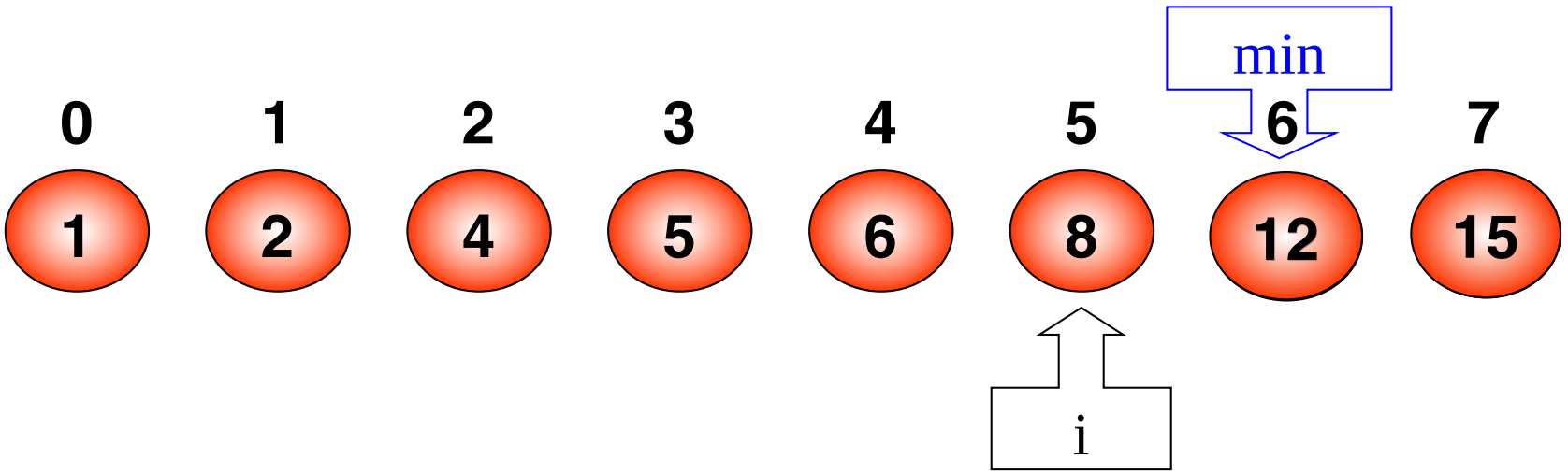
Selection Sort – Ví dụ

Find MinPos(5, 7) **Swap(a[i], a[min])**



Selection Sort – Ví dụ

Find MinPos(6, 7) **Swap(a[i], a[min])**



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 – 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 – Cài đặt~~

Chú ý


```
46  print("4. SelectionSort")
47  a=[6,5,3,1,8,7,2,4]
48  print("Mảng chưa sắp xếp:")
49  print(a)
50  def SelectionSort(a):
51      n=len(a)
52      for i in range(0,n-1,1):#từ 1 tới n-1
53          min=i #chỉ số phần tử nhỏ nhất trong dãy hiện hành
54          for j in range(i+1,n,1):
55              if a[j] < a[min]:
56                  min = j
57          if min !=i:
58              a[i],a[min]=a[min],a[i]
59      return a
60  print("Mảng đã sắp xếp")
61  print(SelectionSort(a))
```

Selection Sort – Đánh giá giải thuật

Phân tích thuật toán:

- Trong mọi trường hợp:
  - Số phép so sánh: S = (N-1)+(N-2)+…+1 = N×(N-1)/2
  - Số phép hoán vị: H = N-1
- Trường hợp tốt nhất, khi mảng M ban đầu đã có thứ tự tăng:
  - Số phép gán: Gmin = 2×(N-1)
- Trường hợp xấu nhất, khi mảng M ban đầu đã có thứ tự giảm dần:
  - Số phép gán: Gmax = 2×[N+(N-1)+ … +1] = N×(N+1)
- Trung bình:
  - Số phép gán: Gavg = [2×(N-1)+N×(N+1)]/2 = (N-1) + N×(N+1)/2
- Số lần hoán vị (một hoán vị bằng 3 phép gán) phụ thuộc vào tình trạng ban đầu của dãy số

Selection Sort – Đánh giá giải thuật

- Đối với một danh sách có n phần tử, vòng lặp ngoài lặp lại n lần. Vòng lặp bên trong lặp lại n-1 khi i bằng 1, và sau đó n-2 vì i bằng 2 và Ở lượt thứ i, cần (n-i) lần so sánh để xác định phần tử nhỏ nhất hiện hành.
- Số lượng so sánh là (n - 1) + (n - 2) + ... + 1, 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: $\sum_{i=1}^{n-1}(n-i) = \frac{n(n-1)}{2}$
- Độ phức tạp thời gian của O (n^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

Các phương pháp sắp xếp thông dụng

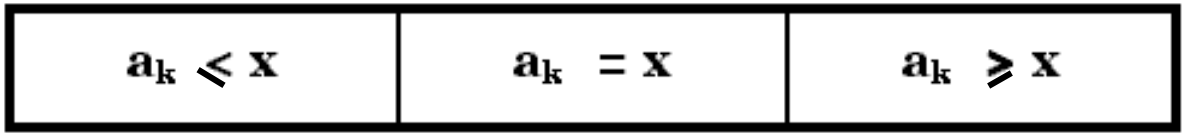
- Phương pháp Đổi chỗ trực tiếp (Interchange sort)
- Phương pháp Nổi bọt (Bubble sort)
- Phương pháp Chèn trực tiếp (Insertion sort)
- Phương pháp Chọn trực tiếp (Selection sort)
- Phương pháp dựa trên phân hoạch (Quick sort)

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 (nhỏ hơn) **x**
  - Phần 2: Gồm các phần tử có giá trị bằng **x**
  - Phần 3: Gồm các phần tử có giá trị không nhỏ hơn (lớn 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

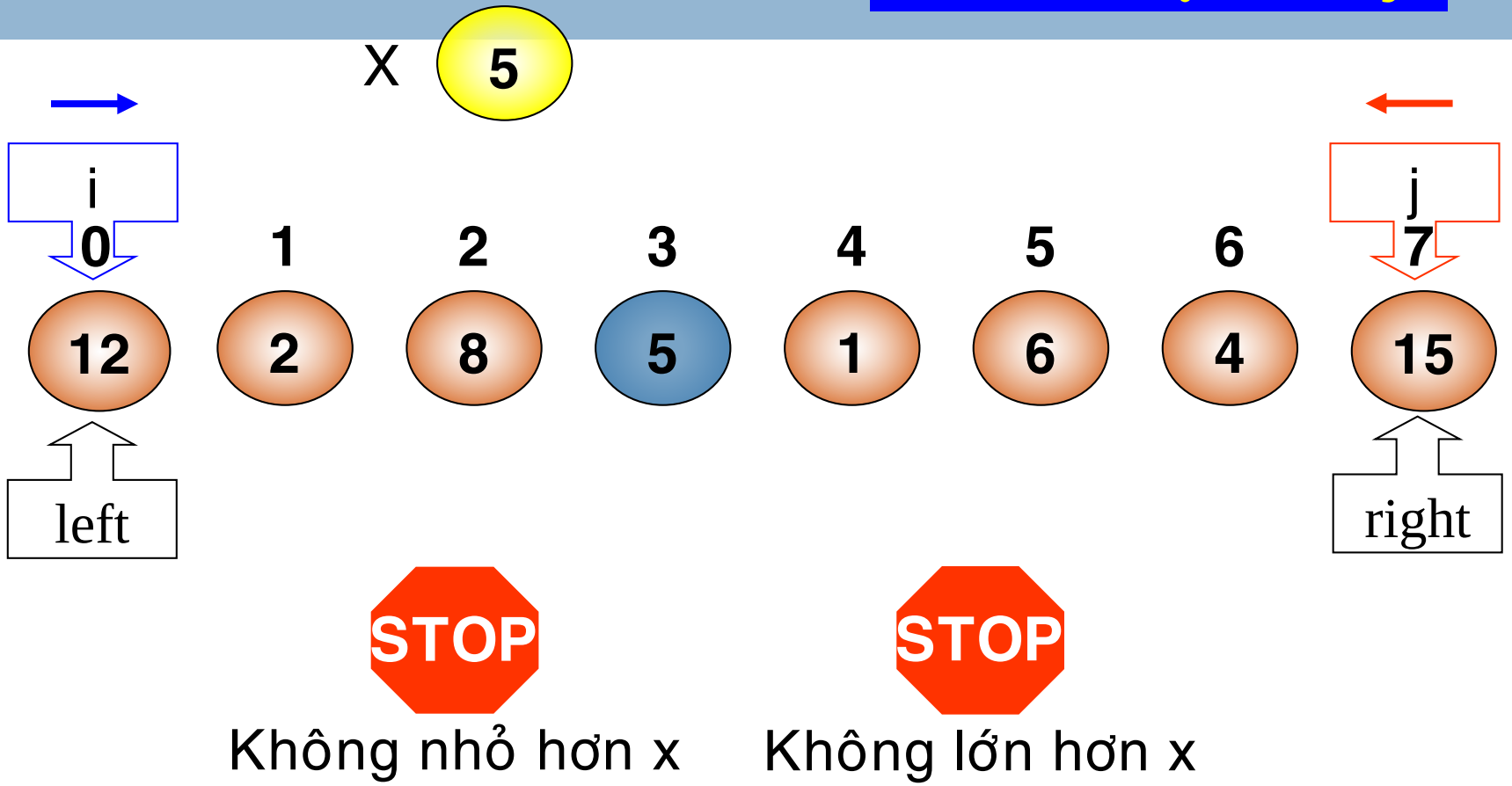
Quick Sort – Ý tưởng



- 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 như ban đầu …
- Chọn x như thế nào?

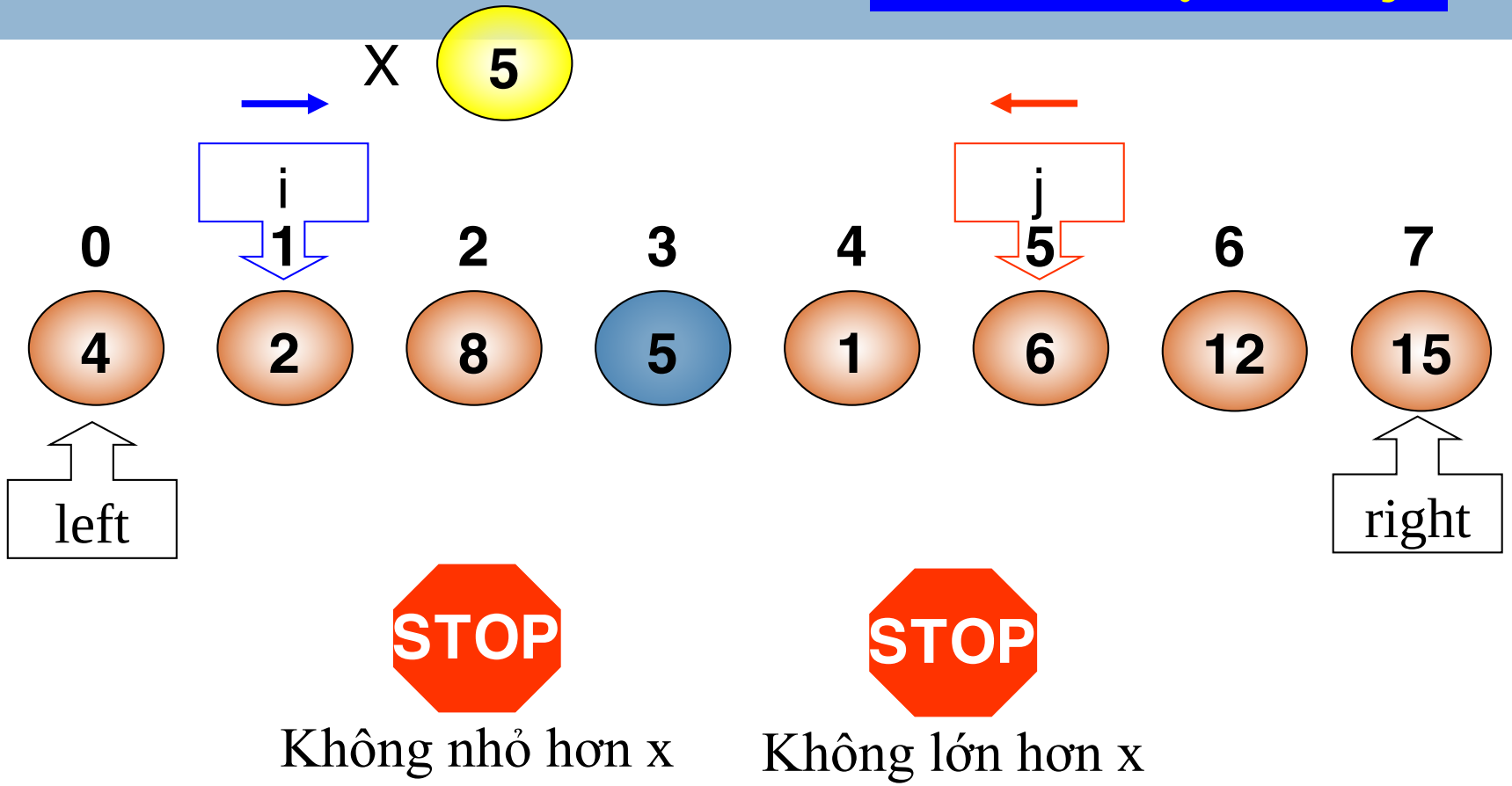
Quick Sort – Ví dụ

Phân hoạch dãy

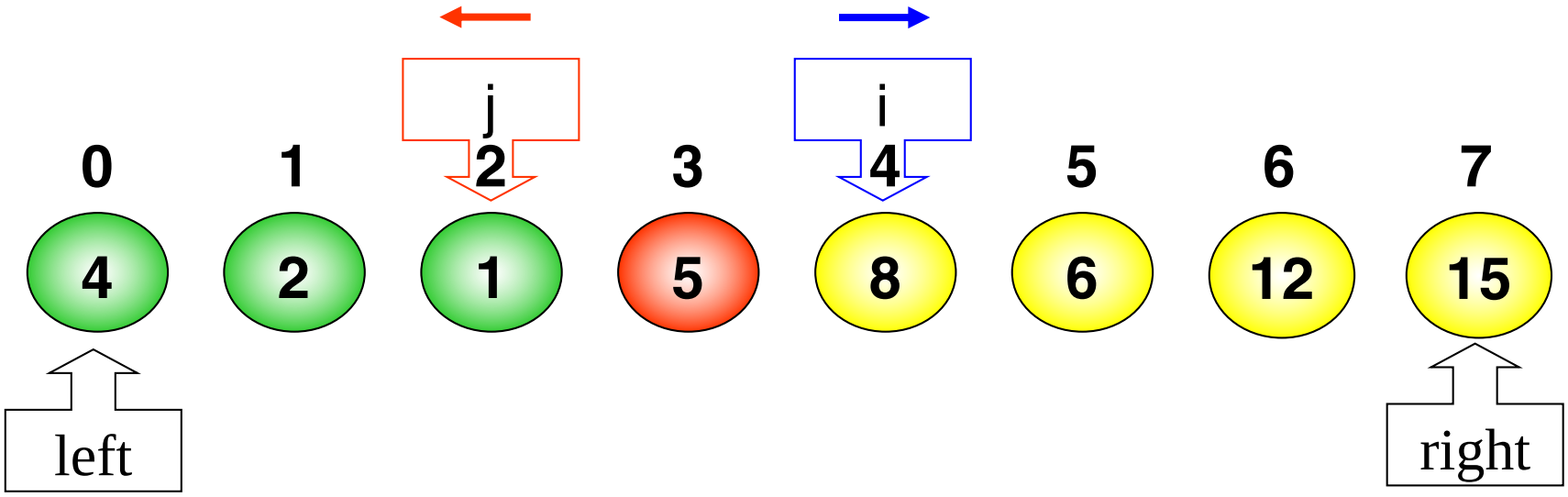


Quick Sort – Ví dụ

Phân hoạch dãy

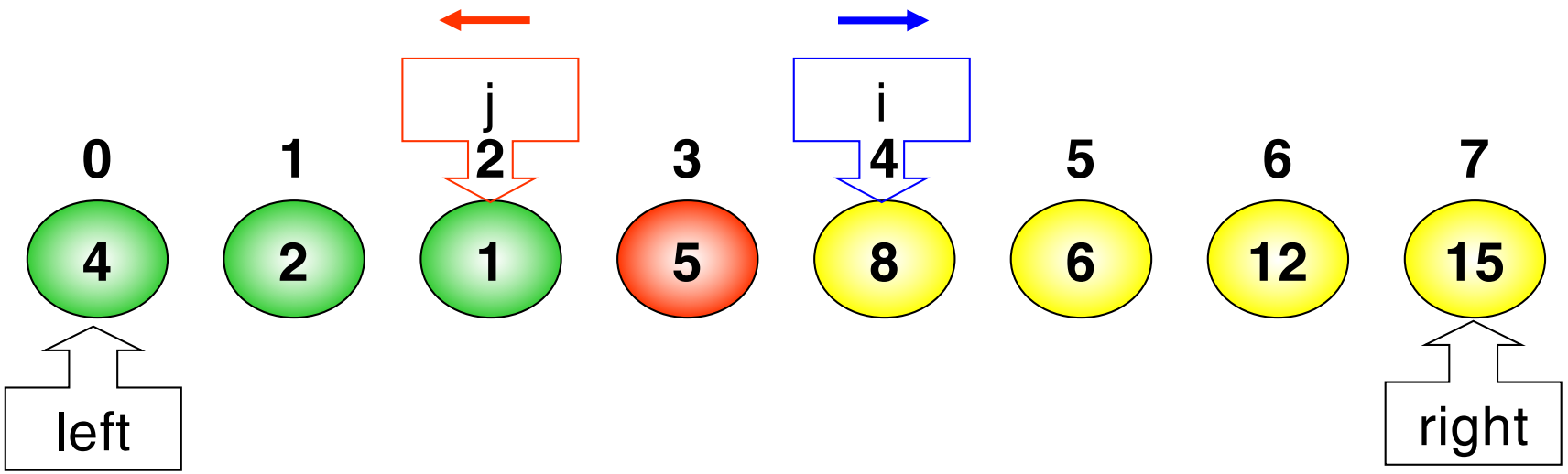


Quick Sort – Ví dụ



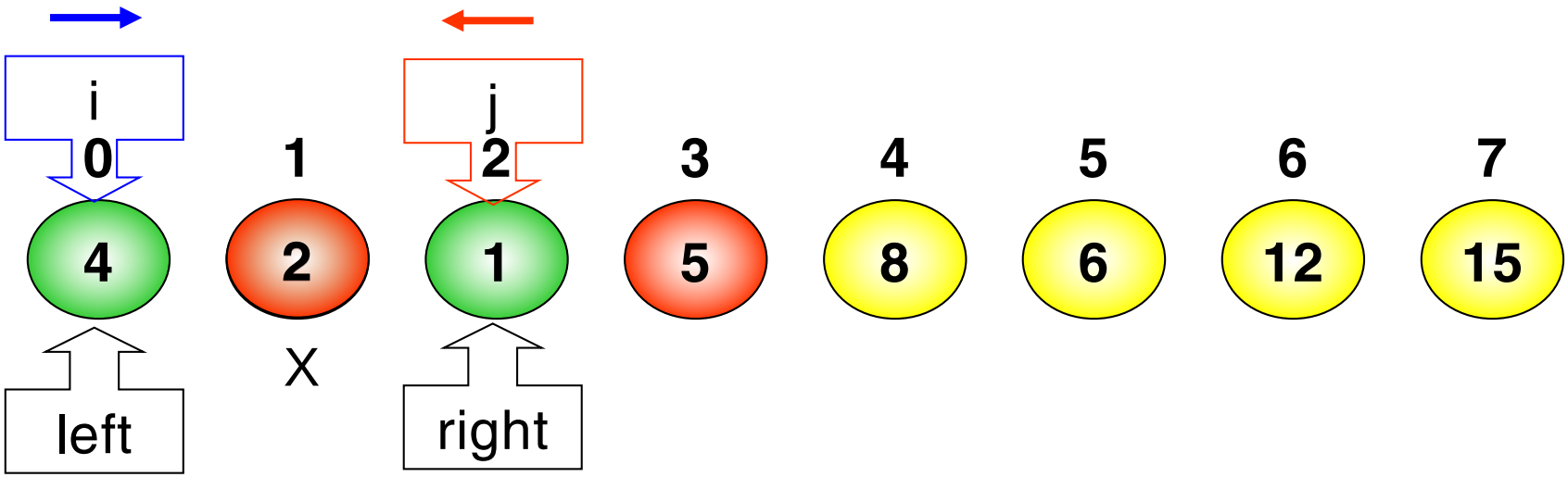
Quick Sort – Ví dụ

- Phân hoạch đoạn [0,2]

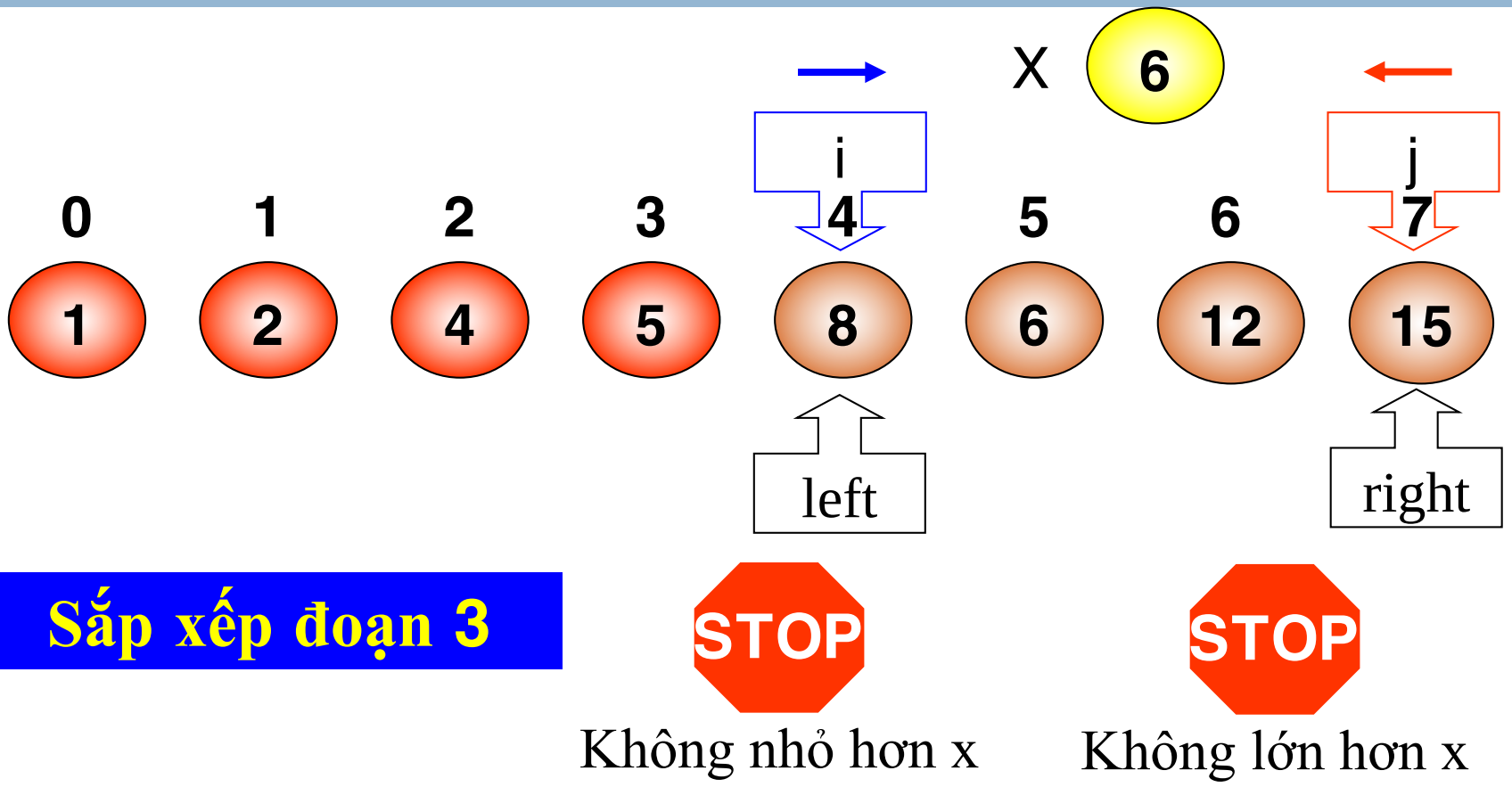


Quick Sort – Ví dụ

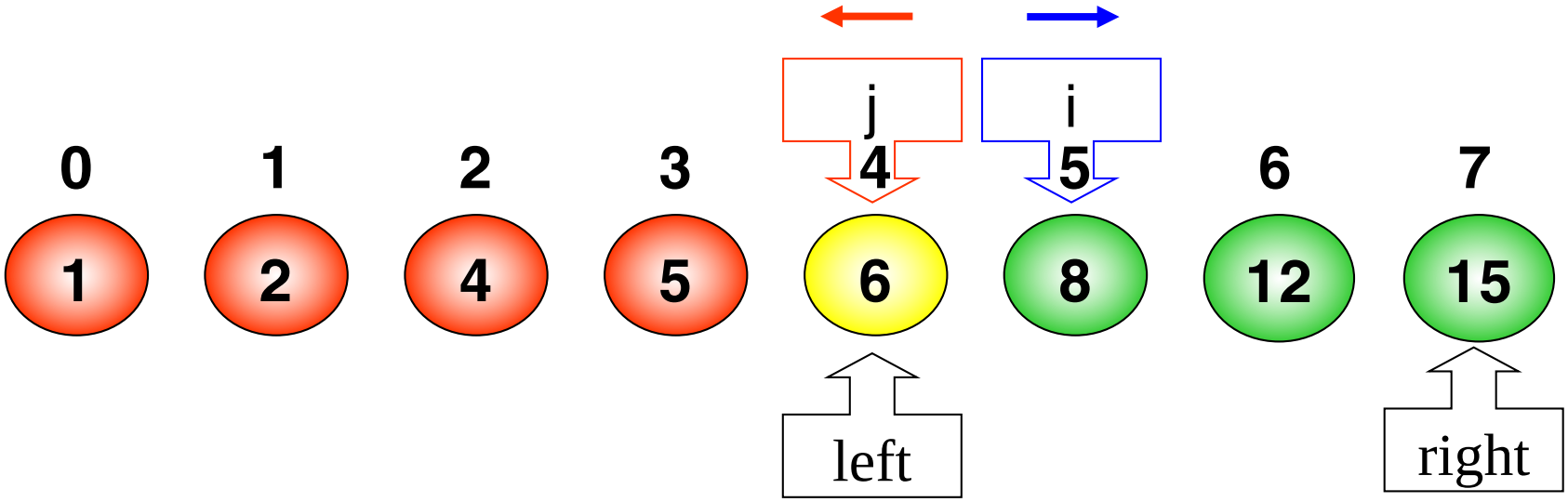
➢ Phân hoạch đoạn [0,2]



Quick Sort – Ví dụ

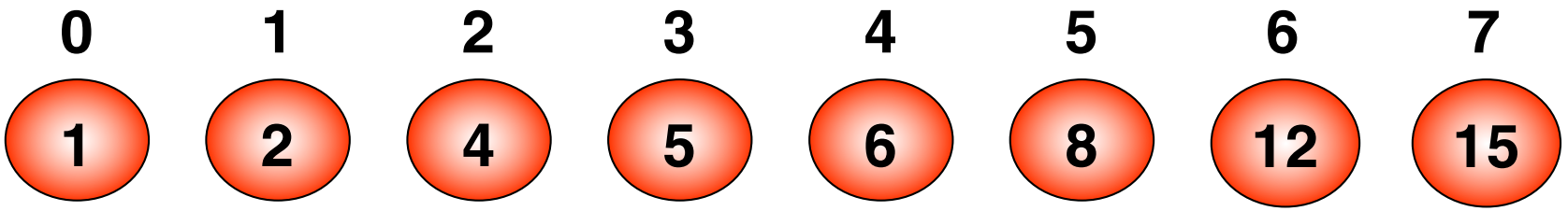


Quick Sort – Ví dụ



Sắp xếp đoạn 3

Quick Sort – Ví dụ



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 3 đoạn:
  a[left].. a[j], a[j+1].. a[i-1], a[i].. a[right]

  // Đoạn 1: a[left].. a[j] $\leq x$

  // Đoạn 2: a[j+1].. a[i-1] $= x$

  // Đoạn 3: a[i].. a[right] $\geq$ 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 – 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 – Cài đặt

Chú ý


```
63  def partition(a,left,right):
64      #Chọn phần tử bên phải cuối cùng làm phần tử pivot
65      pivot =a[right]
66      #con trỏ cho phần tử lớn hơn
67      i=left-1
68      #Duyệt qua tất cả các phần tử và so sánh với phần tử pivot
69      for j in range(left,right):
70          if a[j]<=pivot:
71              #Nếu phần tử nhỏ hơn pivot, thực hiện hoán đổi với phần tử lớn hơn được trỏ bởi i
72              i =i+1
73              #Hoán đổi phần tử tại i với phần tử tại j
74              a[i],a[j]=a[j],a[i]
75      #Hoán đổi phần tử pivot với phần tử lớn hơn chỉ định bởi i
76      a[i+1],a[right] =a[right],a[i+1]
77      return i+1
```

Quick Sort – Cài đặt

Chú ý


```
78  #Hàm thực hiện sắp xếp QuickSort
79  def QuickSort(a,left,right):
80      if left <right:
81          #Tìm phần tử pivot sao cho phần tử pivot nằm bên trái và lớn hơn nằm bên phải
82          pi = partition(a,left,right)
83          #Gọi đệ quy hàm với phần bên trái của pivor
84          QuickSort(a,left,pi-1)
85          #Gọi đệ quy hàm với phần bên phải của pivot
86          QuickSort(a,pi+1,right)
87  print("5. QuickSort")
88  a=[6,5,3,1,8,7,2,4]
89  print("Mảng chưa sắp xếp:")
90  print(a)
91  print("Mảng đã sắp xếp")
92  n=len(a)
93  QuickSort(a,0,n-1)
94  print(a)
```

Quick sort – Đánh giá giải thuật

Nhận xét:

- 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

Phân tích thuật toán:

- Trường hợp tốt nhất, khi mảng M ban đầu đã có thứ tự tăng:
  - Số phép gán: Gmin = 1 + 2 + 4 + … + 2^[Log2(N) – 1] = N-1
  - Số phép so sánh: Smin = N×Log2(N)/2
  - Số phép hoán vị: Hmin = 0
- Trường hợp xấu nhất, khi phần tử X được chọn ở giữa dãy con là giá trị lớn nhất của dãy con. Trường hợp này thuật toán QuickSort trở nên chậm chạp nhất:
  - Số phép gán: Gmax = 1 + 2 + … + (N-1) = N×(N-1)/2
  - Số phép so sánh: Smax = (N-1)×(N-1)
  - Số phép hoán vị: Hmax = (N-1) + (N-2) + … + 1 = N×(N-1)/2
- Trung bình:
  - Số phép gán: Gavg = [(N-1)+N(N-1)/2]/2 = (N-1)×(N+2)/4
  - Số phép so sánh: Savg = [N×Log2(N)/2 + N×(N-1)]/2 = N×[Log2(N)+2N–2]/4
  - Số phép hoán vị: Havg = N×(N-1)/4

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	$O(N^{2)}$

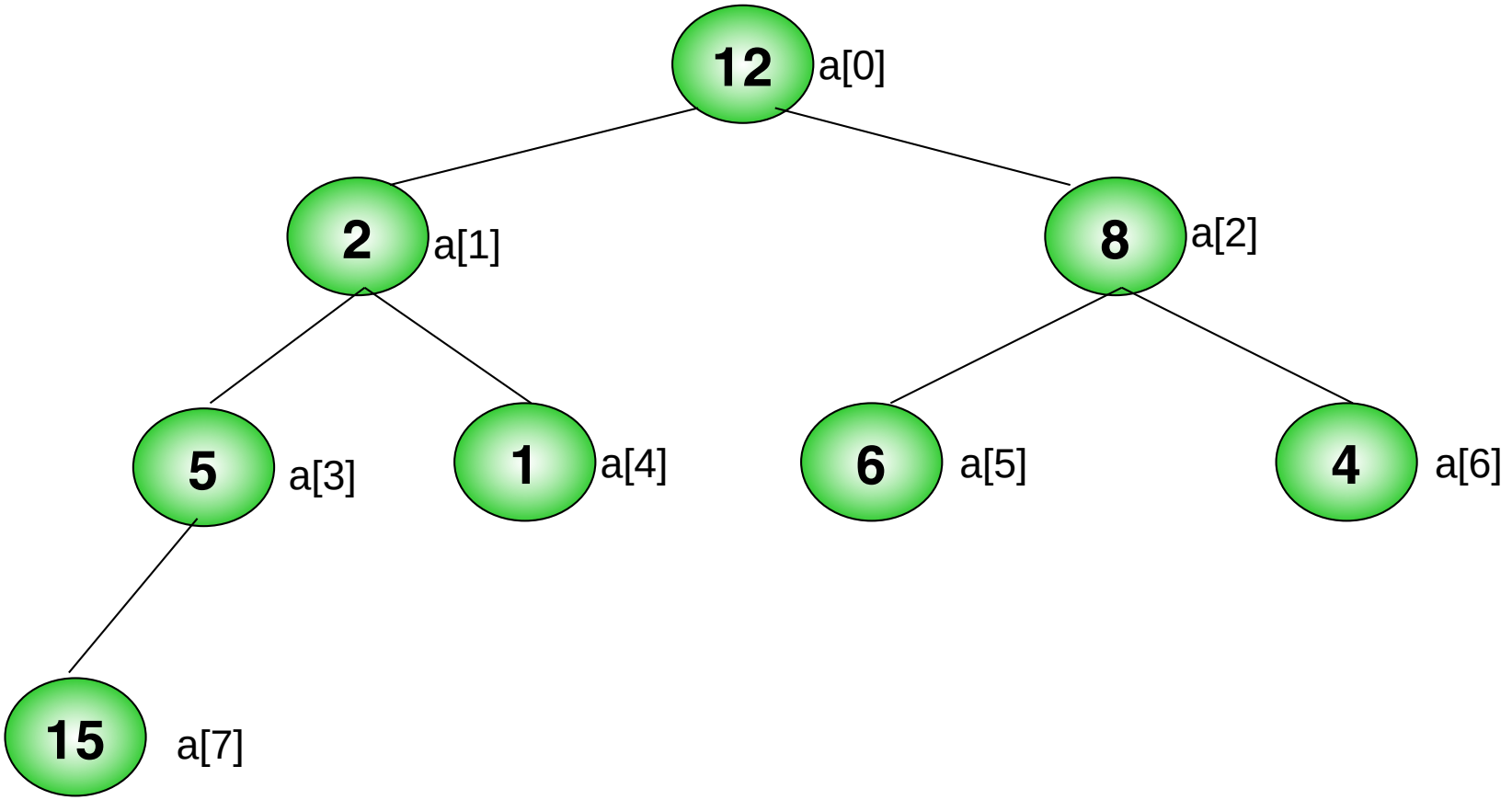
Thuật Toán Sắp Xếp Heap Sort

- Heap Sort tận dụng được các phép so sánh ở bước i-1 mà thuật toán sắp xếp chọn trực tiếp không tận dụng được. Để làm được điều này Heap sort thao tác dựa trên cây.
- Chúng ta bắt đầu bằng cách chuyển đổi danh sách thành Max Heap - Cây nhị phân trong đó phần tử lớn nhất là nút gốc. Sau đó chúng ta đặt mục đó vào cuối danh sách. Sau đó, chúng ta xây dựng lại Heap Max của chúng ta hiện có một giá trị nhỏ hơn, đặt giá trị lớn nhất mới trước mục cuối cùng của danh sách, lặp lại quá trình xây dựng heap này cho đến khi tất cả các nút được loại bỏ.

Thuật Toán Sắp Xếp Heap Sort

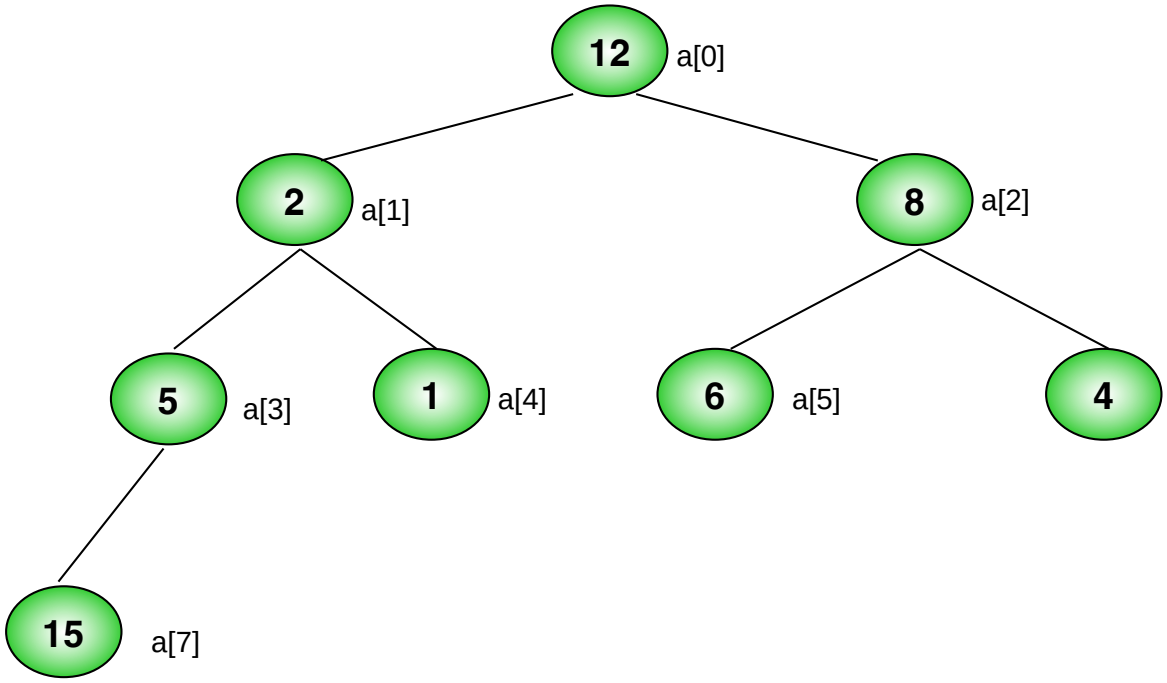
- Cho dãy số :

12	2	8	5	1	6	4	15
0	1	2	3	4	5	6	7

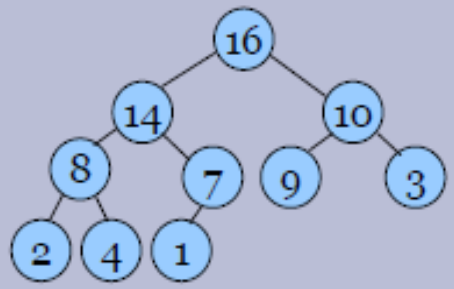


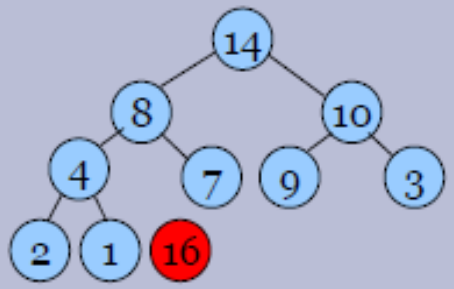
Thuật toán sắp xếp Heap Sort

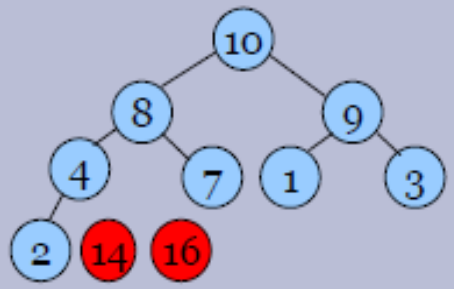
- Ở cây trên, phần tử ở mức i chính là phần tử lớn trong cặp phần tử ở mức i +1, do đó phần tử ở nút gốc là phần tử lớn nhất.
- Nếu loại bỏ gốc ra khỏi cây, thì việc cập nhật cây chỉ xảy ra trên những nhánh liên quan đến phần tử mới loại bỏ, còn các nhánh khác thì bảo toàn.
- Bước kế tiếp có thể sử dụng lại kết quả so sánh của bước hiện tại.
- Vì thế độ phức tạp của thuật toán $O(n\log_2 n)$

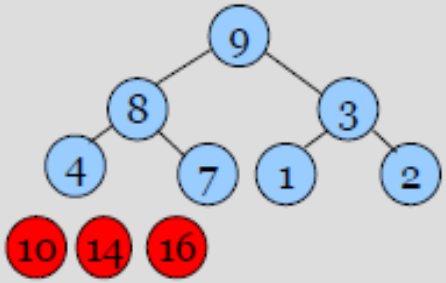


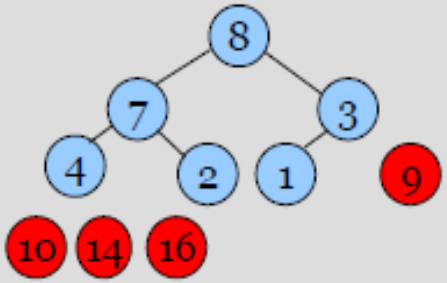
Heap Sort

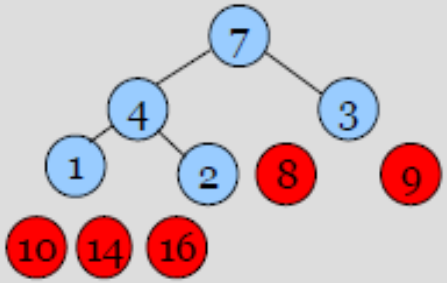


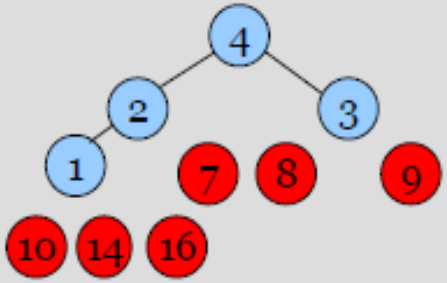


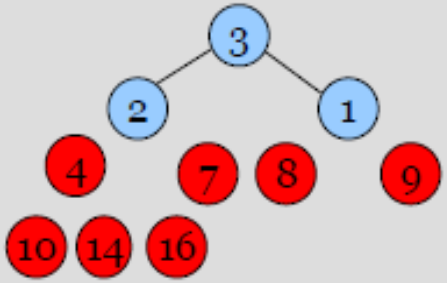


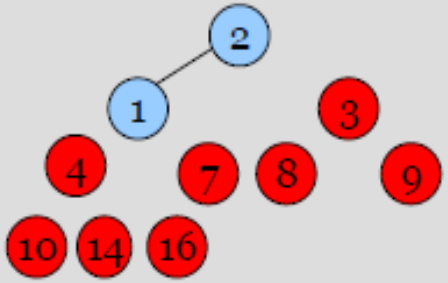


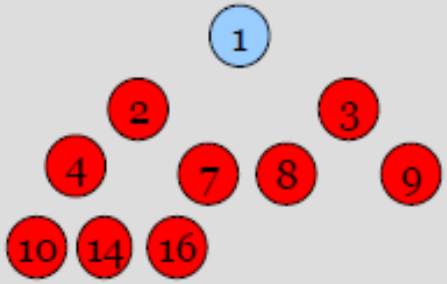












1 2 3 4 7 8 9 10 14 16

Các Bước Thuật Toán

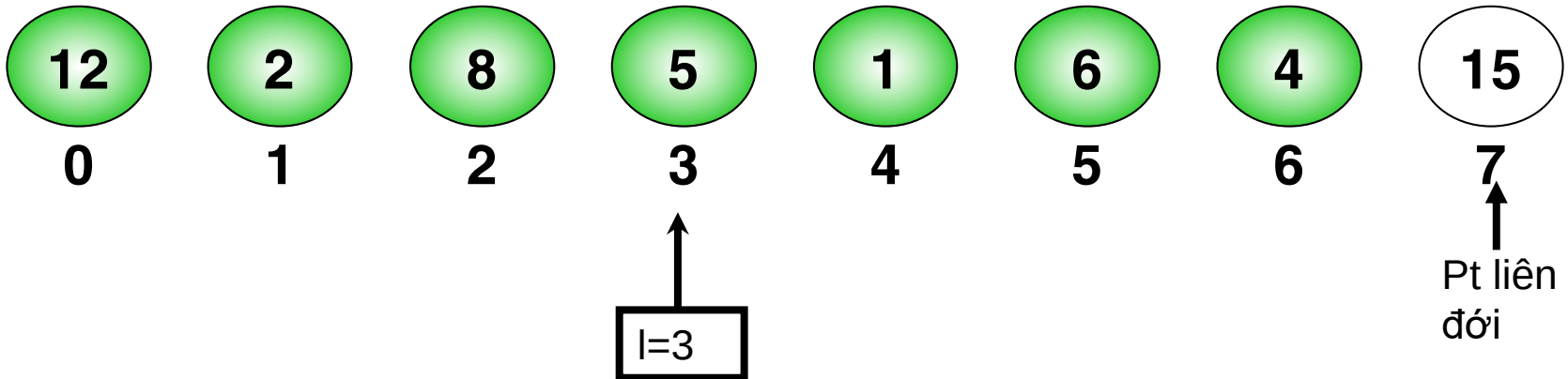
- Giai đoạn 1 : Hiệu chỉnh dãy số ban đầu thành heap
- Giai đoạn 2: Sắp xếp dãy số dựa trên heap:
  - Bước 1:Đưa phần tử lớn nhất về vị trí đúng ở cuối dãy:
    r = n-1; Swap (a_1 , a_r);
  - Bước 2: Loại bỏ phần tử lớn nhất ra khỏi heap: r = r-1;
    Hiệu chỉnh phần còn lại của dãy từ a1 , a2 ... ar thành một heap.
  - Bước 3:

    Nếu r>1 (heap còn phần tử): Lặp lại Bước 2
    Ngược lại : Dừng

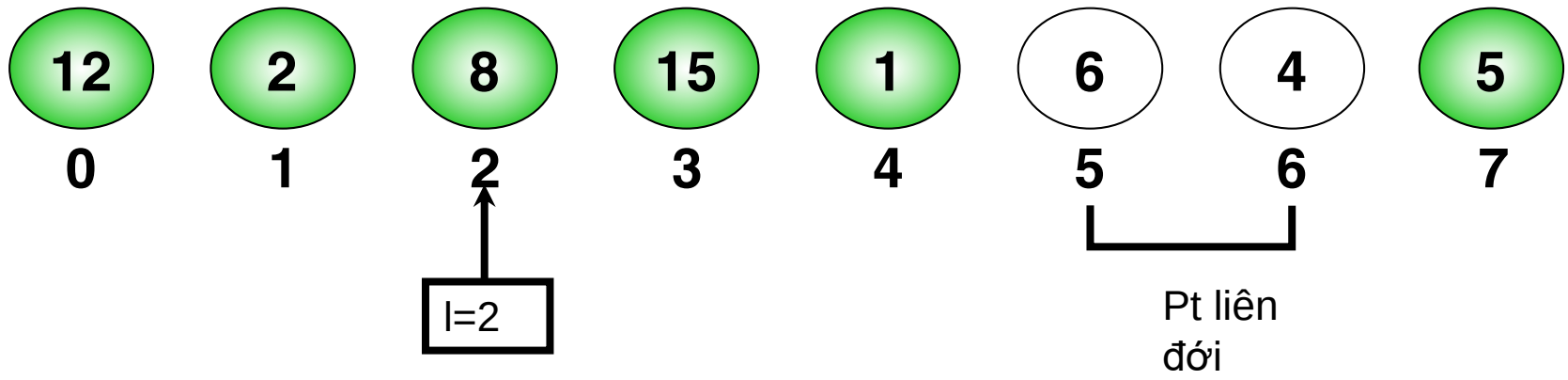
Minh Họa Thuật Toán

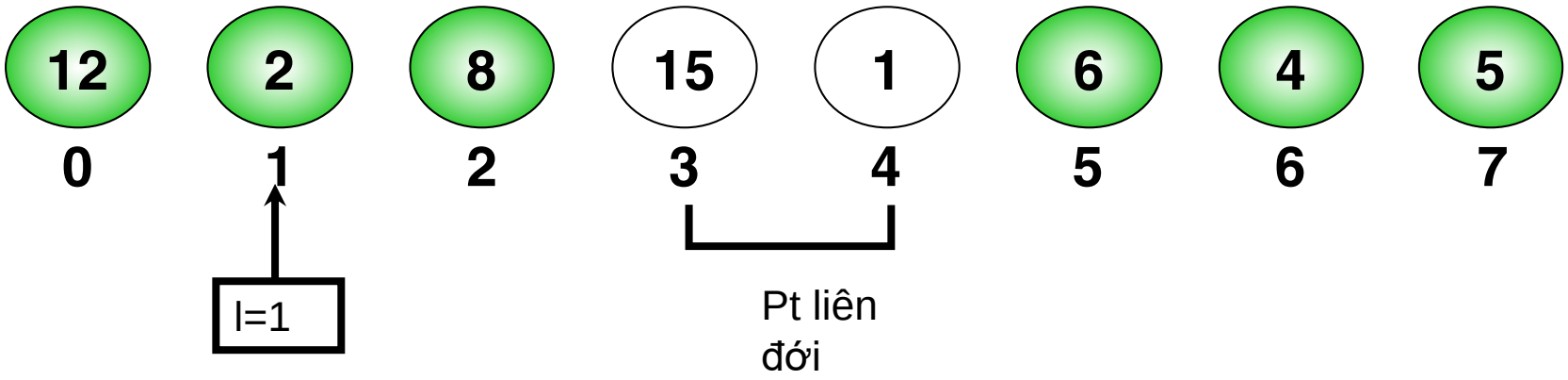
- **Heap:** Là một dãy các phần tử $a_l, a_{l+1}, \ldots, a_r$ thoả các quan hệ với mọi $i \in [l, r]$:
  - $a_i \geq a_{2i+1}$
  - $a_i \geq a_{2i+2}$ // (a_i, a_{2i+1}), (a_i, a_{2i+2}) là các cặp phần tử liên đới
- Cho dãy số : 12 2 8 5 1 6 4 15

➢Giai đoạn 1: Hiệu chỉnh dãy ban đầu thành Heap, chọn l=n/2-1

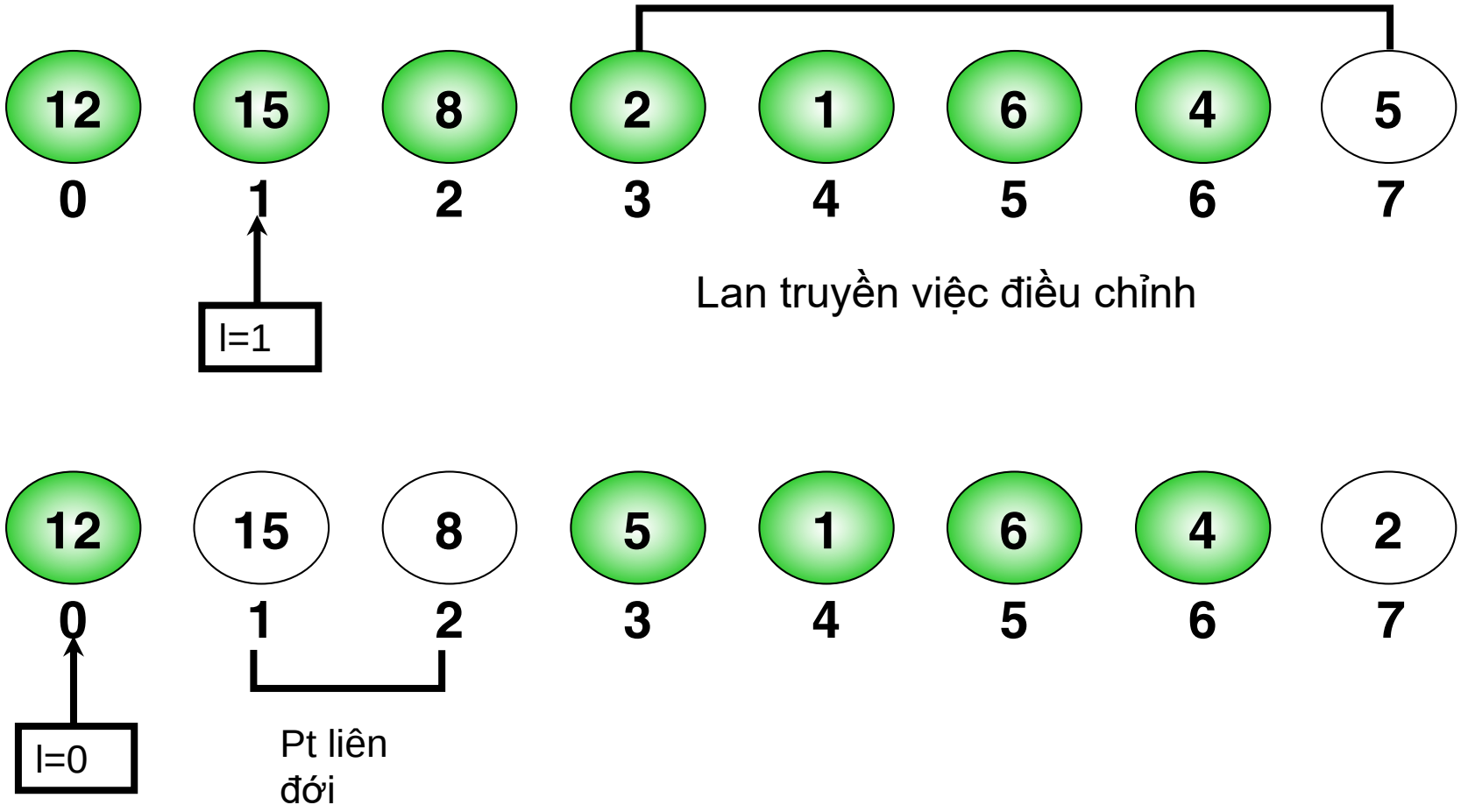


Minh Họa Thuật Toán

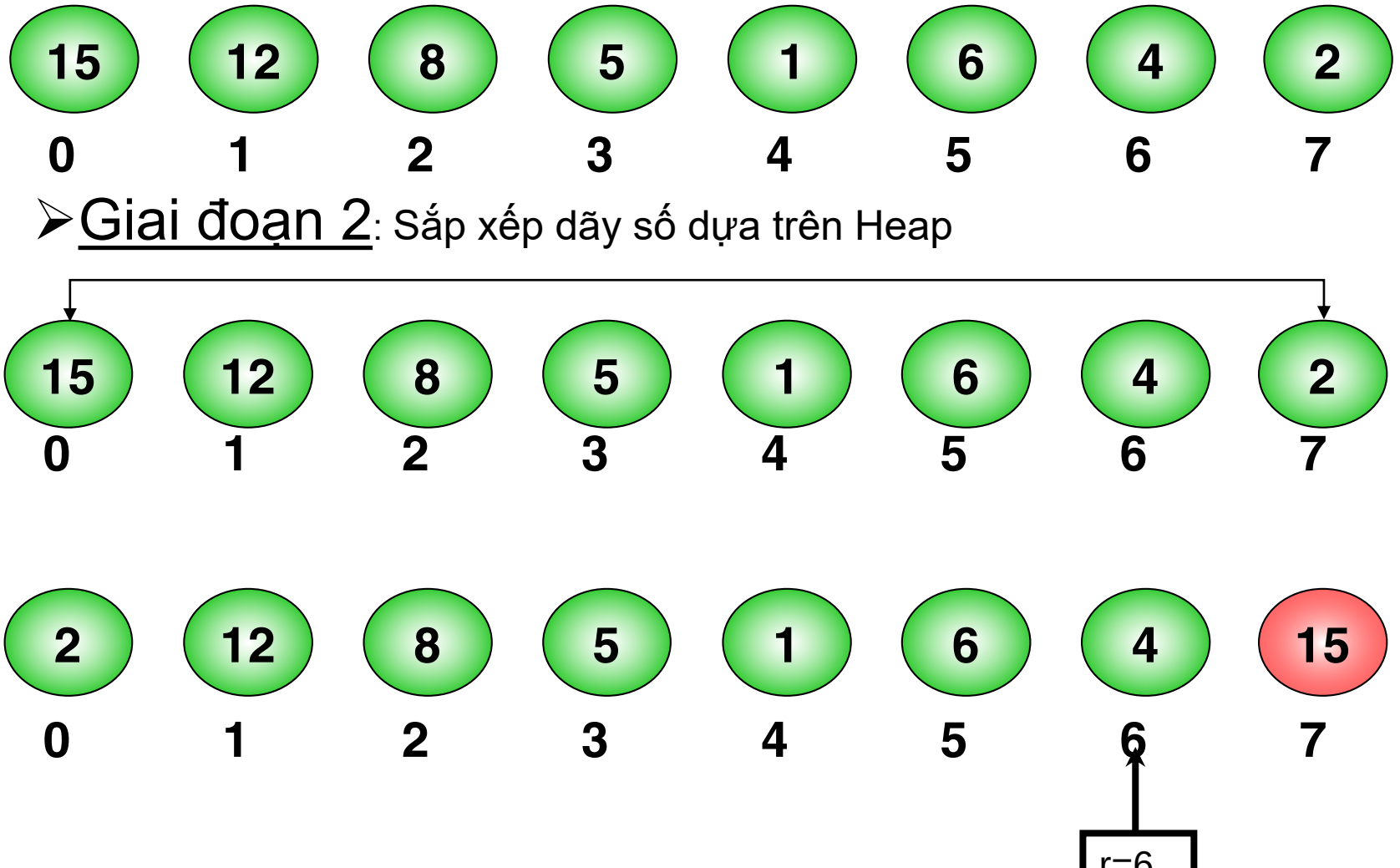




Minh Họa Thuật Toán

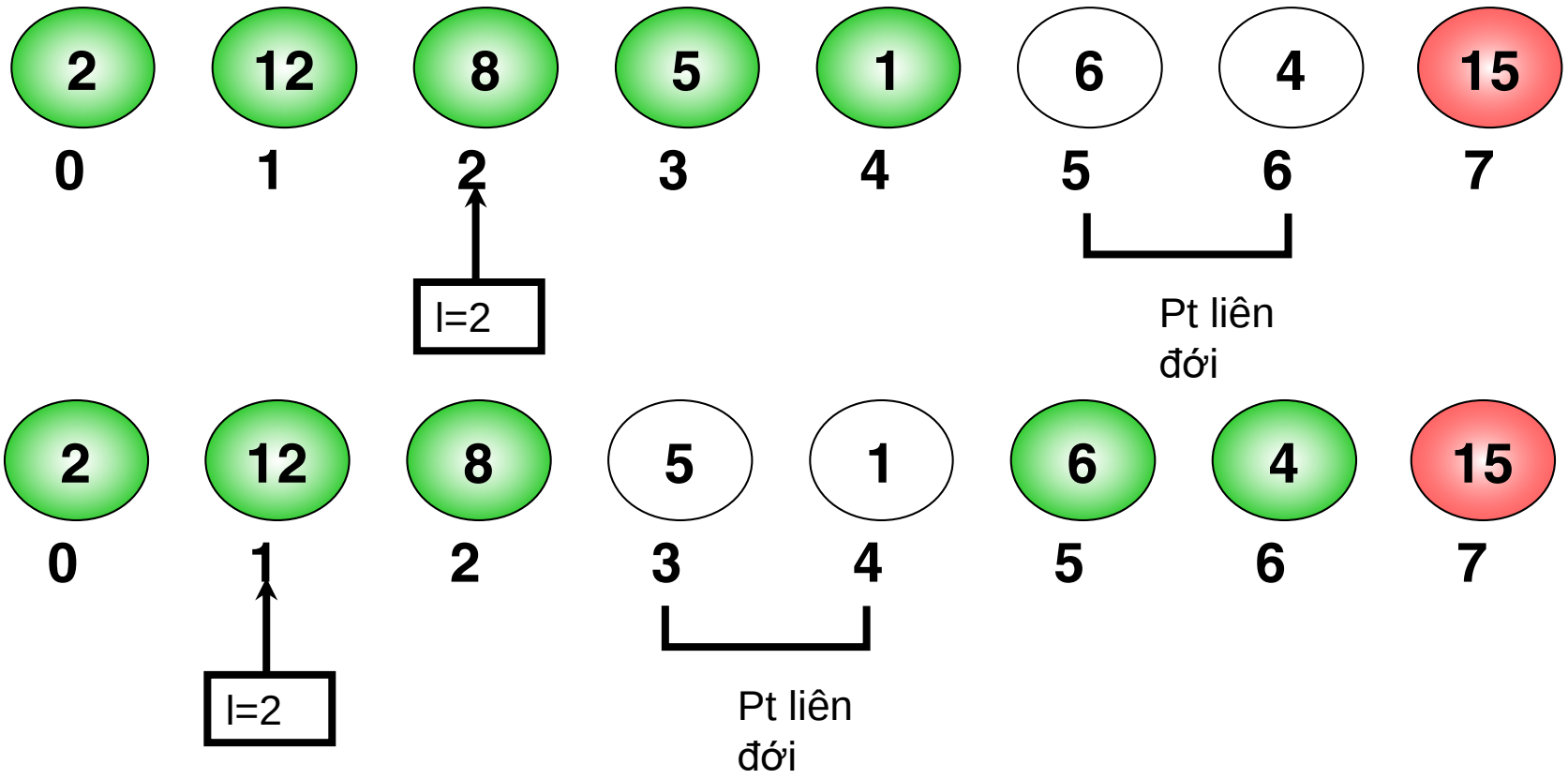


Minh Họa Thuật Toán

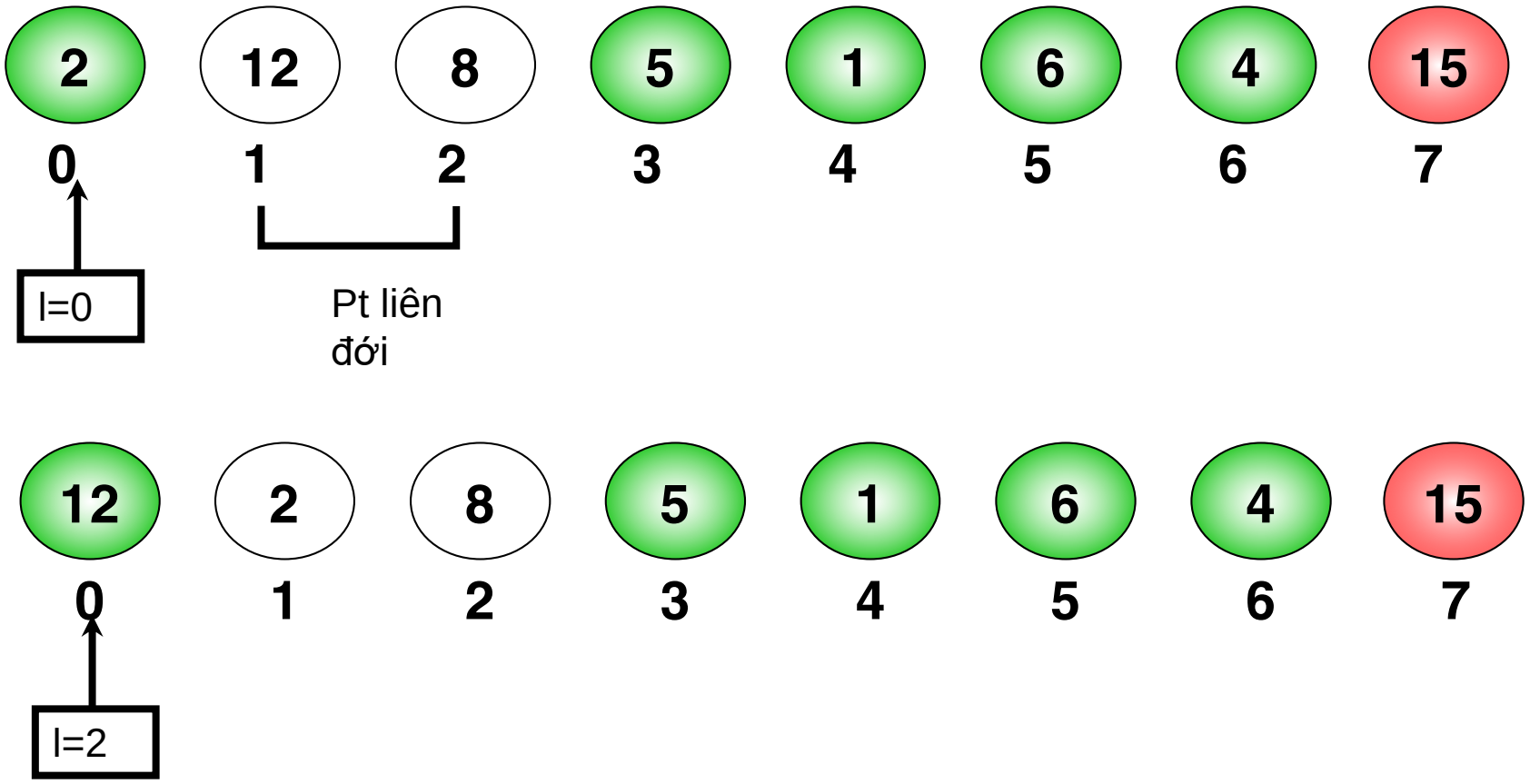


Minh Họa Thuật Toán

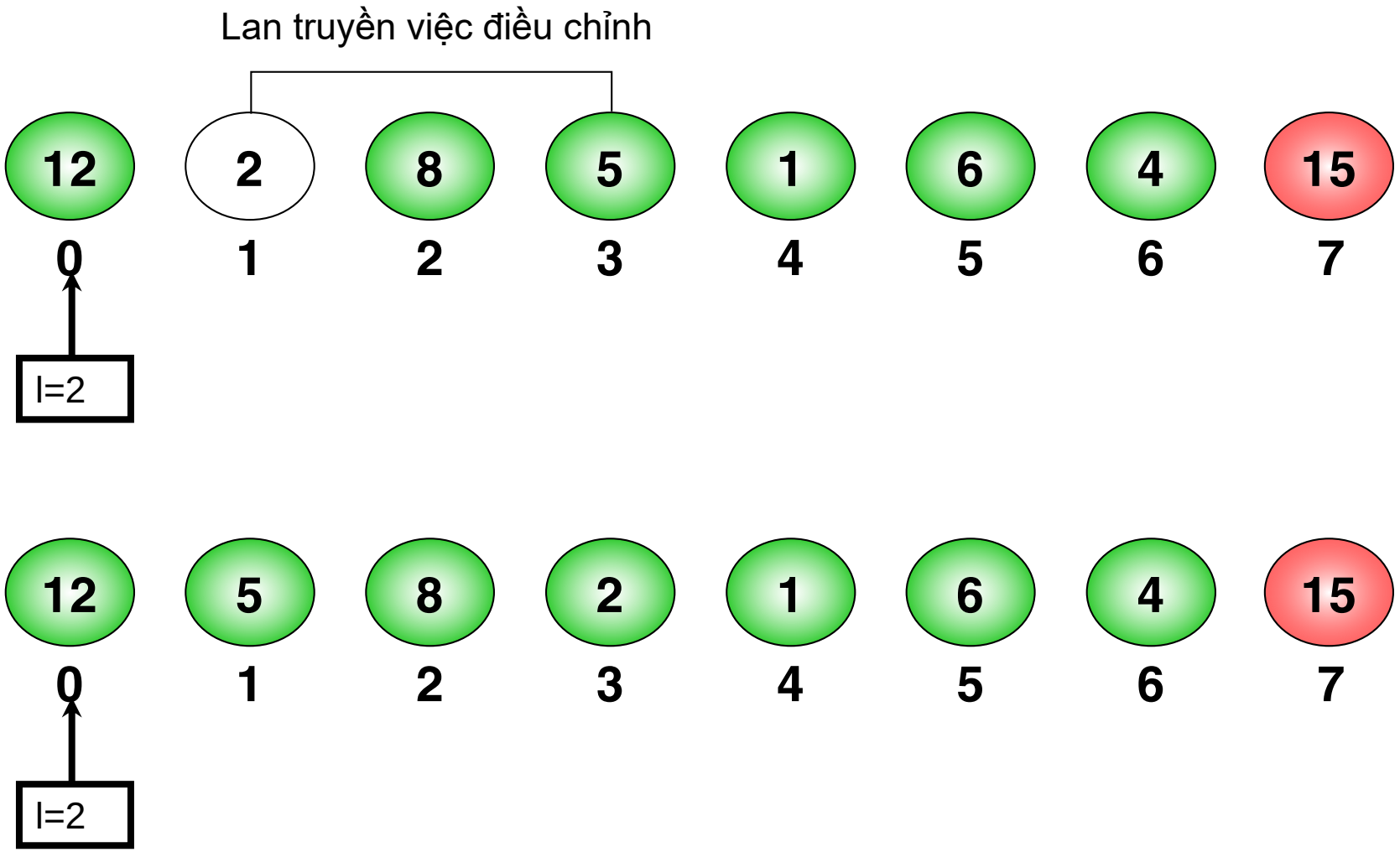
➢Hiệu chỉnh Heap



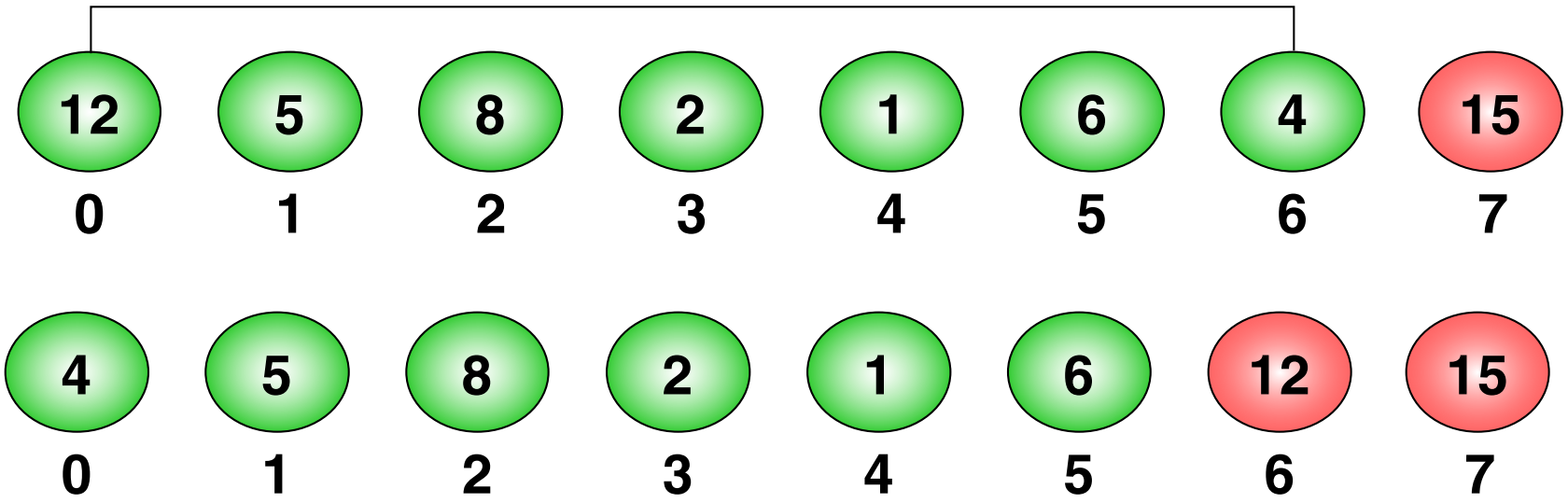
Minh Họa Thuật Toán



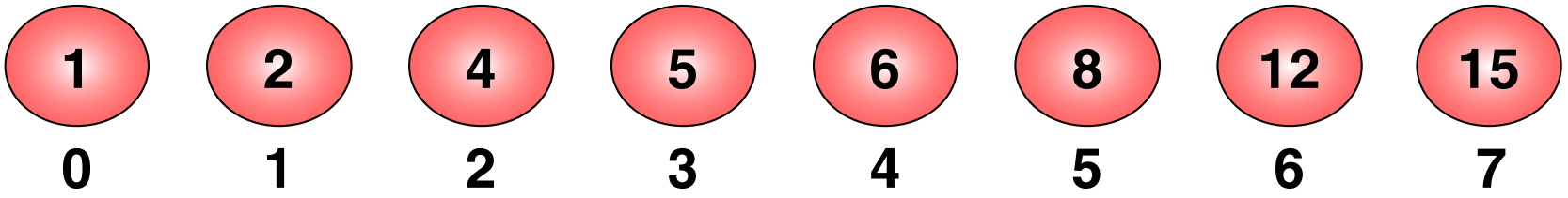
Minh Họa Thuật Toán



Minh Họa Thuật Toán



- Thực hiện với r= 5,4,3,2 ta được



Cài Đặt Thuật Toán

https://dnmtechs.com/thuat-toan-sap-xep-vun-dong-heap-sort/

```
95  print("============HEAPKSORT============")
96  def HEAP_SORT(a,n,i):
97      largest =i
98      left =2*i+1
99      right =2*i+2
100     if left<n and a[largest]<a[left]:
101         largest = left
102     if right<n and a[largest]<a[right]:
103         largest = right
104     if largest !=i:
105         a[i],a[largest] =a[largest],a[i]
106         HEAP_SORT(a,n,largest)
```

Cài Đặt Thuật Toán

```
108  #Hàm thực hiện sắp xếp vun đống HeapSort
109  def HEAPSORT(a):
110      n =len(a)
111      for i in range(n,-1,-1):
112          HEAP_SORT(a,n,i)
113      for i in range(n-1,0,-1):
114          a[i],a[0]=a[0],a[i]
115          HEAP_SORT(a,i,0)
116  print("5. HEAPSORT")
117  a=[6,5,3,1,8,7,2,4]
118  print("Mảng chưa sắp xếp:")
119  print(a)
120  HEAPSORT(a)
121  print("Mảng đã sắp xếp")
122  print(a)
```

Độ Phức Tạp của Thuật Toán

- Trước tiên chúng ta hãy xem xét độ phức tạp thời gian của hàm heapify. Trong trường hợp xấu nhất, phần tử lớn nhất không bao giờ là phần tử gốc, điều này gây ra một lệnh gọi đệ quy để heapify. Trong khi các cuộc gọi đệ quy có vẻ expensive, hãy nhớ rằng chúng ta đang làm việc với cây nhị phân.
- Hình dung một cây nhị phân có 3 phần tử, nó có chiều cao là 2. Bây giờ hãy hình dung một cây nhị phân có 7 phần tử, nó có chiều cao là 3. Cây phát triển logarit theo n. Hàm heapify đi ngang qua cây đó trong thời gian O (log (n)).
- Hàm heap_sort lặp lại qua mảng n lần. Do đó, độ phức tạp thời gian tổng thể của thuật toán Heap Sort là O (nlog (n)).

Merge Sort – Ý Tưởng

- Giải thuật Merge sort sắp xếp dãy $\mathbf{a_1, a_2, ..., a_n}$ dựa trên nhận xét sau:
  - Mỗi dãy $\mathbf{a_1, a_2, ..., a_n}$ bất kỳ là một tập hợp các dãy con liên tiếp mà mỗi dãy con đều đã có thứ tự.
    - Ví dụ: dãy 12, 2, 8, 5, 1, 6, 4, 15 có thể coi như gồm 5 dãy con không giảm (12); (2, 8); (5); (1, 6); (4, 15).
  - Dãy đã có thứ tự coi như có 1 dãy con.

➔ Hướng tiếp cận: tìm cách làm giảm số dãy con không giảm của dãy ban đầu.

➔ https://viblo.asia/p/cung-on-lai-nhung-thuat-toan-sap-xep-va-xay-dung-ham-sort-su-dung-

Merge Sort – thuật toán

Bước 1 : // Chuẩn bị

k = 1; // k là chiều dài của dãy con trong bước hiện hành

Bước 2 :

Tách dãy $a_0, a_1, ., a_{n-1}$ thành 2 dãy b, c theo nguyên tắc luân phiên từng nhóm k phần tử:

$b = a_0, ., a_k, a_{2k}, ., a_{3k}, .$

$c = a_{k+1}, ., a_{2k+1}, a_{3k+1}, .$

Bước 3 :

Trộn từng cặp dãy con gồm k phần tử của 2 dãy b, c vào a.

Bước 4 :

k = k*2;

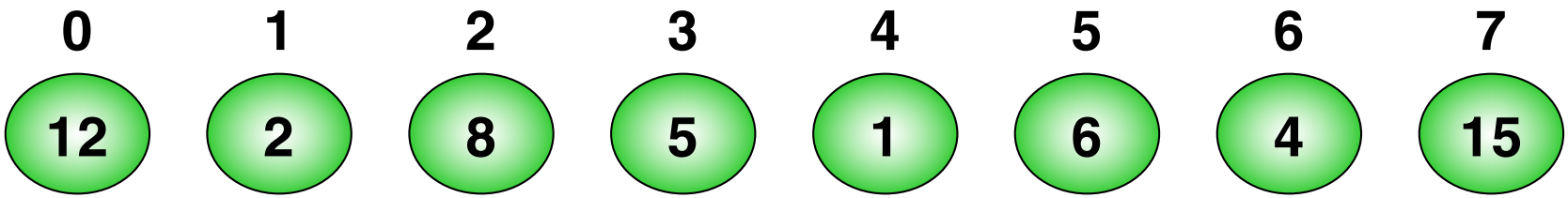
Nếu k < n thì trở lại bước 2.

Ngược lại: Dừng

Merge Sort – Ví Dụ

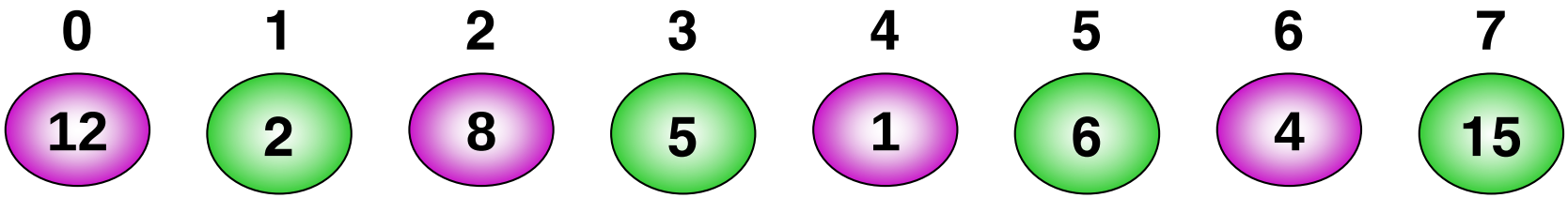
- k=1

- Phân phối luân phiên



Merge Sort – Ví Dụ

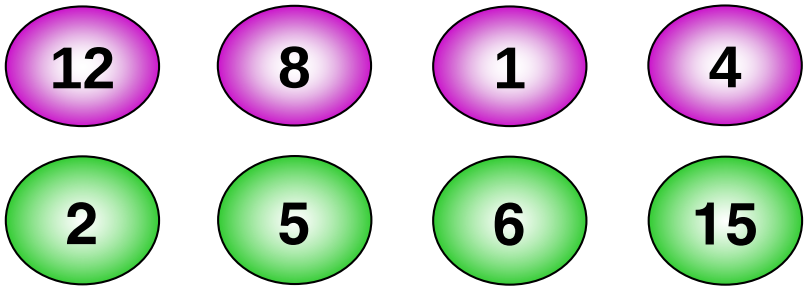
- k = 1
- Phân phối luân phiên



Merge Sort – Ví Dụ

- Trộn từng cặp đường chạy

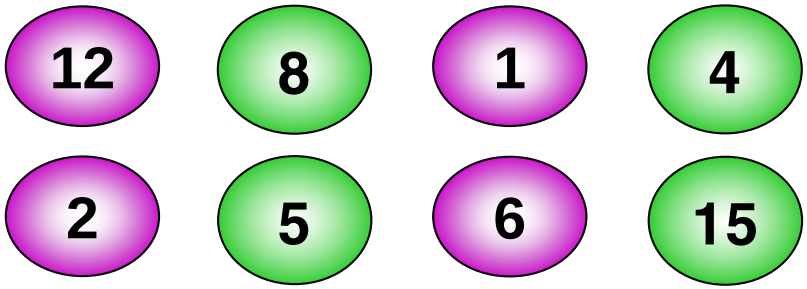
0	1	2	3	4	5	6	7



Merge Sort – Ví Dụ

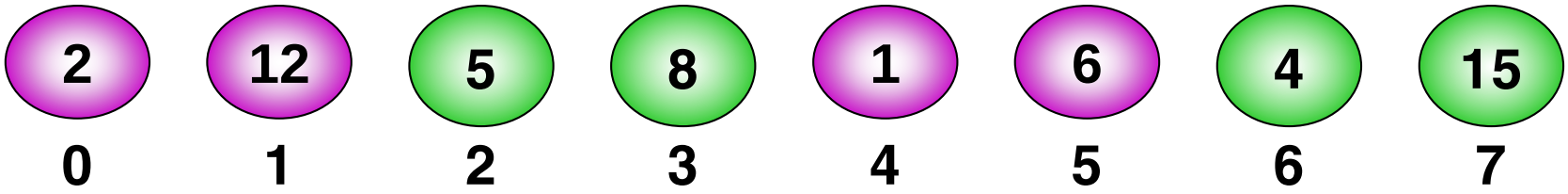
- k = 1
- Trộn từng cặp đường chạy

0	1	2	3	4	5	6	7



Merge Sort – Ví Dụ

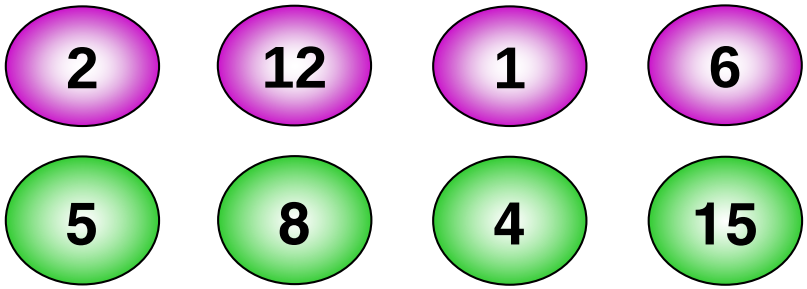
- k = 2
- Phân phối luân phiên



Merge Sort – Ví Dụ

- k = 2
- Trộn từng cặp đường chạy

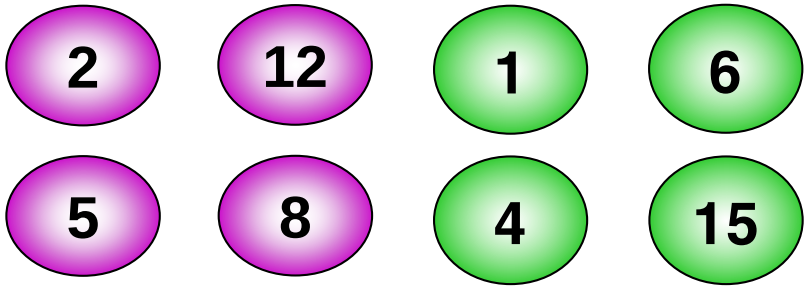
0 1 2 3 4 5 6 7



Merge Sort – Ví Dụ

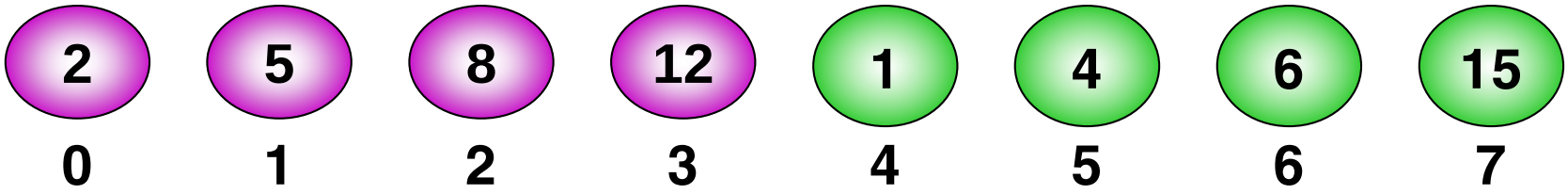
- k = 2
- Trộn từng cặp đường chạy

0	1	2	3	4	5	6	7



Merge Sort – Ví Dụ

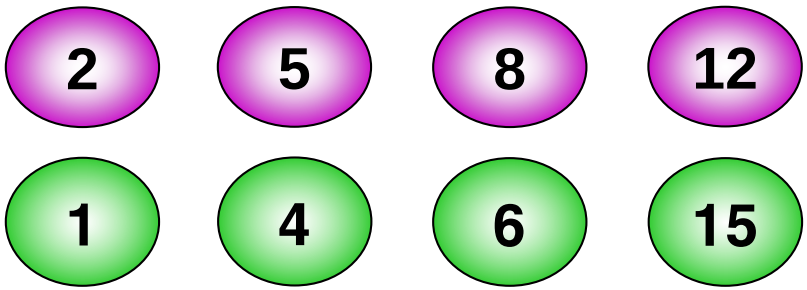
- k = 4
- Phân phối luân phiên



Merge Sort – Ví Dụ

- k = 4
- Trộn từng cặp đường chạy

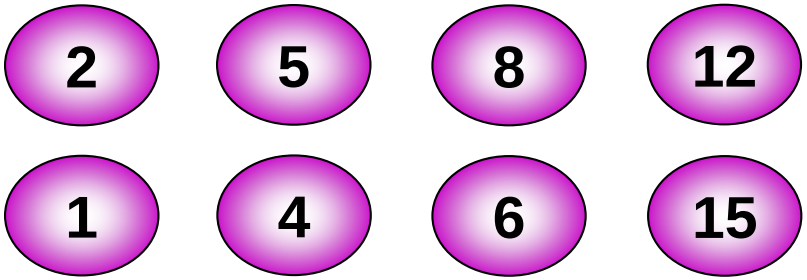
0	1	2	3	4	5	6	7



Merge Sort – Ví Dụ

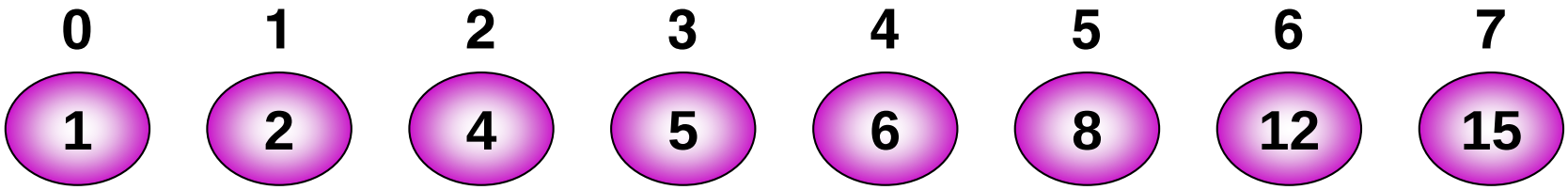
- k = 4
- Trộn từng cặp đường chạy

0 1 2 3 4 5 6 7

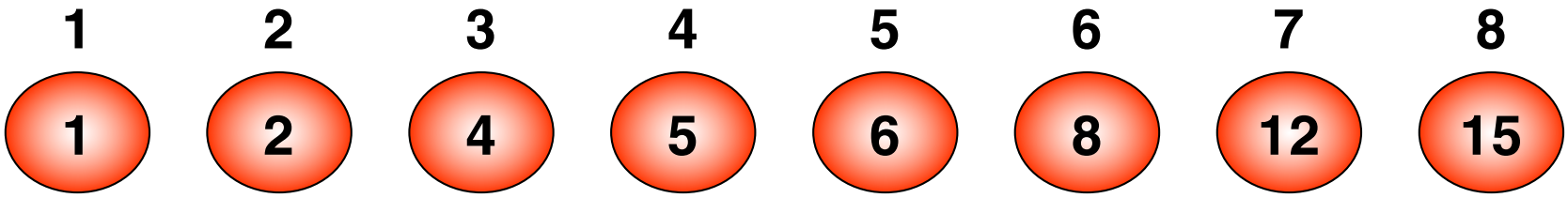


Merge Sort – Ví Dụ

- k = 8



Merge Sort – Ví Dụ



Merge Sort – Cài Đặt

- Dữ liệu hỗ trợ: 2 mảng b, c:

  int b[MAX], c[MAX], nb, nc;

- Các hàm cần cài đặt:
  - **void MergeSort(int a[], int N);** : Sắp xếp mảng (a, N) tăng dần
  - **void Distribute(int a[], int N, int &nb, int &nc, int k);** Phân phối đều luân phiên các dãy con độ dài k từ mảng a vào hai mảng con b và c
  - **void Merge(int a[], int nb, int nc, int k);** : Trộn mảng b và mảng c vào mảng a
  - **void MergeSubarr(int a[], int nb, int nc, int &pa, int &pb, int &pc, int k);** : Trộn một cặp dãy con từ b và c vào a

Merge Sort – Cài Đặt

- Thuật toán phân chia và trị này chia một danh sách thành một nửa và tiếp tục chia danh sách cho 2 cho đến khi nó chỉ có các phần tử số ít.
- Các phần tử liền kề trở thành các cặp được sắp xếp, sau đó các cặp được sắp xếp cũng được merge và sắp xếp với các cặp khác.
- Quá trình này tiếp tục cho đến khi chúng ta nhận được một danh sách được sắp xếp với tất cả các yếu tố của danh sách đầu vào chưa được sắp xếp.

Merge Sort – Cài Đặt

- Chúng ta đệ quy chia danh sách thành một nửa cho đến khi chúng ta có danh sách với kích thước = một. Sau đó chúng ta merge từng nửa được tách ra, sắp xếp chúng trong quá trình.
- Sắp xếp được thực hiện bằng cách so sánh các yếu tố nhỏ nhất của mỗi nửa. Yếu tố đầu tiên của mỗi danh sách là yếu tố đầu tiên được so sánh. Nếu nửa đầu bắt đầu với giá trị nhỏ hơn, thì chúng ta thêm nó vào danh sách đã sắp xếp. Sau đó, chúng ta so sánh giá trị nhỏ thứ hai của nửa đầu với giá trị nhỏ nhất đầu tiên của nửa thứ hai.
- Mỗi lần chúng tôi chọn giá trị nhỏ hơn ở đầu một nửa, chúng ta sẽ di chuyển chỉ mục của mục nào cần so sánh với một mục.

Merge Sort – Cài Đặt

```
123  print("===========MERGESORT===========")
124  def MERGE_SORT(a):
125      n=len(a)
126      if n>1:
127          mid = n//2
128          left_half = a[:mid]
129          right_half =a[mid:]
130          #Đệ quy sắp xếp từng phần
131          MERGE_SORT(left_half)
132          MERGE_SORT(right_half)
133          i=j=k=0
134          #Hợp nhất hai danh sách con đã sắp xếp
135          while i<len(left_half) and j<len(right_half):
136              if left_half[i]<right_half[j]:
137                  a[k]=left_half[i]
138                  i+=1
139              else:
140                  a[k]=right_half[j]
141                  j+=1
142              k+=1
```

Chú ý

Merge Sort – Cài Đặt

```
139                 else:
140                     a[k]=right_half[j]
141                     j+=1
142                 k+=1
143             while i<len(left_half):
144                 a[k]=left_half[i]
145                 i+=1
146                 k+=1
147             while j<len(right_half):
148                 a[k] =right_half[j]
149                 j+=1
150                 k+=1
151     print("6. MERGESORT")
152     a=[6,5,3,1,8,7,2,4]
153     print("Mảng chưa sắp xếp:")
154     print(a)
155     MERGE_SORT(a)
156     print("Mảng đã sắp xếp")
157     print(a)
```

Merge Sort – Cài Đặt

- Trước tiên chúng ta hãy xem chức năng merge. Phải mất hai danh sách và lặp lại n lần, trong đó n là kích thước của đầu vào kết hợp của chúng. Hàm merge_sort chia mảng đã cho thành 2 và sắp xếp đệ quy các mảng con. Vì đầu vào đang được đệ quy là một nửa của những gì đã được đưa ra, giống như cây nhị phân, điều này làm cho thời gian cần thiết để xử lý tăng logarit theo n.
- Do đó, độ phức tạp thời gian tổng thể của thuật toán merge sort là O (nlog (n)).

Shell Sort – Cài Đặt

- Shell sort là 1 kỹ thuật phân chia 1 danh sách nhất định thành các danh sách con và sau đó sắp xếp chúng bằng cách sử dụng insertion sort (sắp xếp chèn). Thuật toán khoảng trống n để chọn các mục cách nhau n khoảng trống để tạo thành danh sách con.
- Các danh sách con sau đó được sắp xếp bằng cách sử dụng insertion sort được kết hợp với nhau. Danh sách kết hợp không được sắp xếp hoàn toàn nhưng giúp thuật toán này có lợi thế là các mục gần vị trí cuối cùng hơn

Shell Sort – Cài Đặt

Mô tả ở trên có thể không có nhiều ý nghĩa, nhưng một ví dụ sẽ hữu ích. Giả sử bạn có danh sách: [39, 6, 2, 51, 30, 42, 7, 4, 16] và giá trị khoảng cách (gap value) là 3.

Danh sách phụ đầu tiên sẽ có các mục: 39, 51, 7

Danh sách phụ thứ hai: 6, 30, 4

Danh sách phụ thứ ba & cuối cùng: 2, 42, 16

Sau khi insertion sort, mỗi danh sách con sẽ được sắp xếp như sau:

Đầu tiên: 7, 39, 51

Thứ hai: 4, 6, 30

Thứ ba: 2, 16, 42

Shell Sort – Cài Đặt

Danh sách con đã được sắp xếp hiện được kết hợp theo một cách nhất định. Mỗi mục (ite
trong danh sách con được đưa vào chỉ mục (index) mà từ đó giá trị danh sách con chưa c
sắp xếp ban đầu được tập hợp.

Do đó, bạn sẽ kết thúc với trình tự bên dưới:

[7, 4, 2, 39, 6, 16, 51, 30, 42]
Lưu ý rằng danh sách vẫn chưa được sắp xếp nhưng các mục đã ở gần vị trí cuối cùng củ
chúng hơn. Sau khi thực hiện insertion sort trên tổ hợp danh sách này, danh sách cuối cù
cũng được sắp xếp:

[2, 4, 6, 7, 16, 30, 39, 42, 51]

```python
print("===========7.SHELLSORT===========")
def SHELL_SORT(a):
    n=len(a)
    gap =n//2
    #lặp đến khi gap =0
    while gap >0:
        #Lặp qua tất cả các phần tử trong mảng
        for i in range(gap,n):
            #Lấy giá trị phần tử đang xét
            temp =a[i]
            #So sánh với tất cả các phần tử trước nó
            j= i
            while j>=gap and a[j-gap]>temp:
                a[j] =a[j-gap]
                j=j-gap
            #Gán giá trị vừa tìm được vào vị trí đúng
            a[j]=temp
        gap=gap//2
    return a
print("7. SHELLSORT")
a=[6,5,3,1,8,7,2,4]
print("Mảng chưa sắp xếp:")
print(a)
print("Mảng đã sắp xếp")
print(SHELL_SORT(a))
```