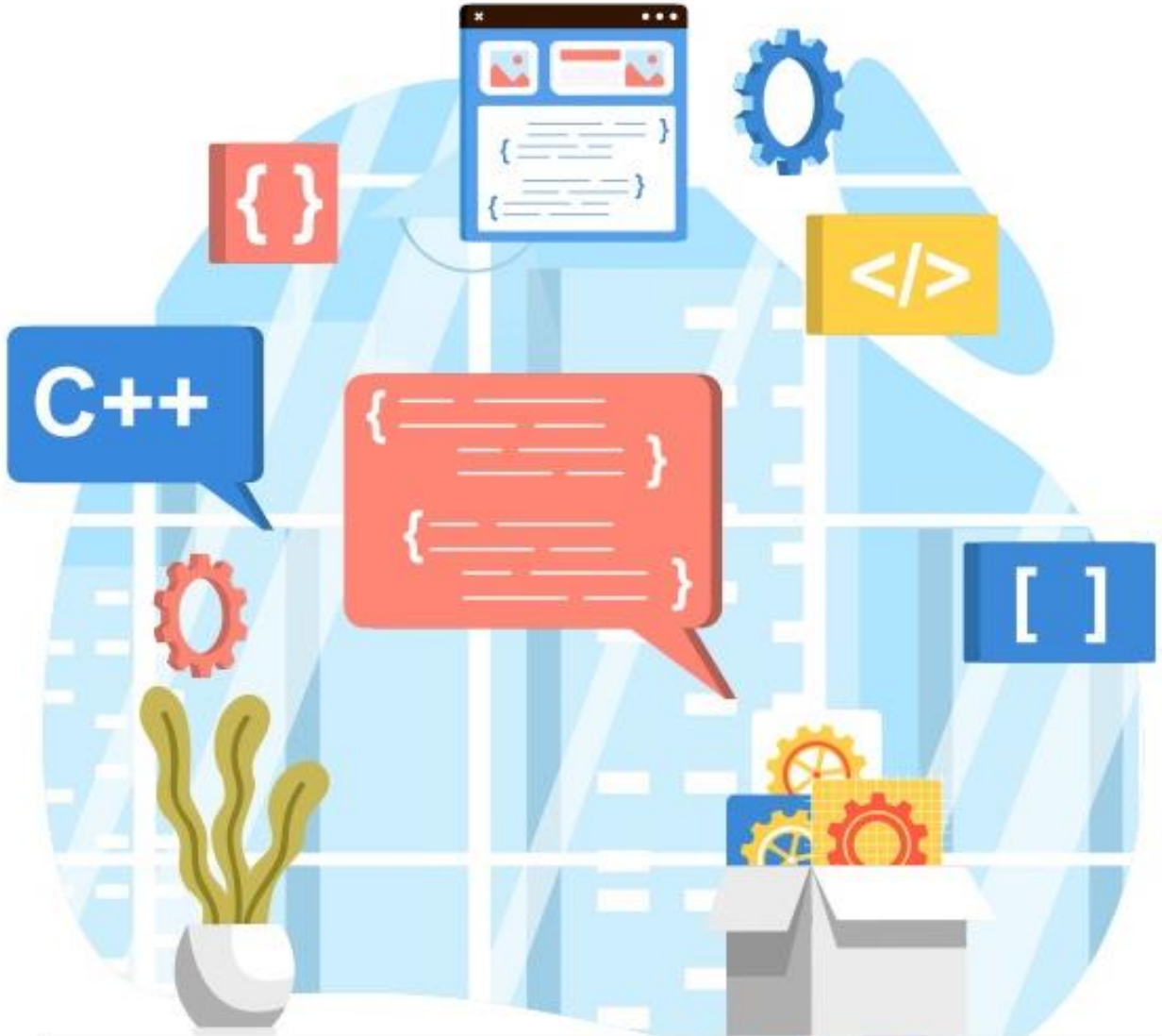
CÁC GIẢI THUẬT SẮP XẾP HIỆU QUẢ

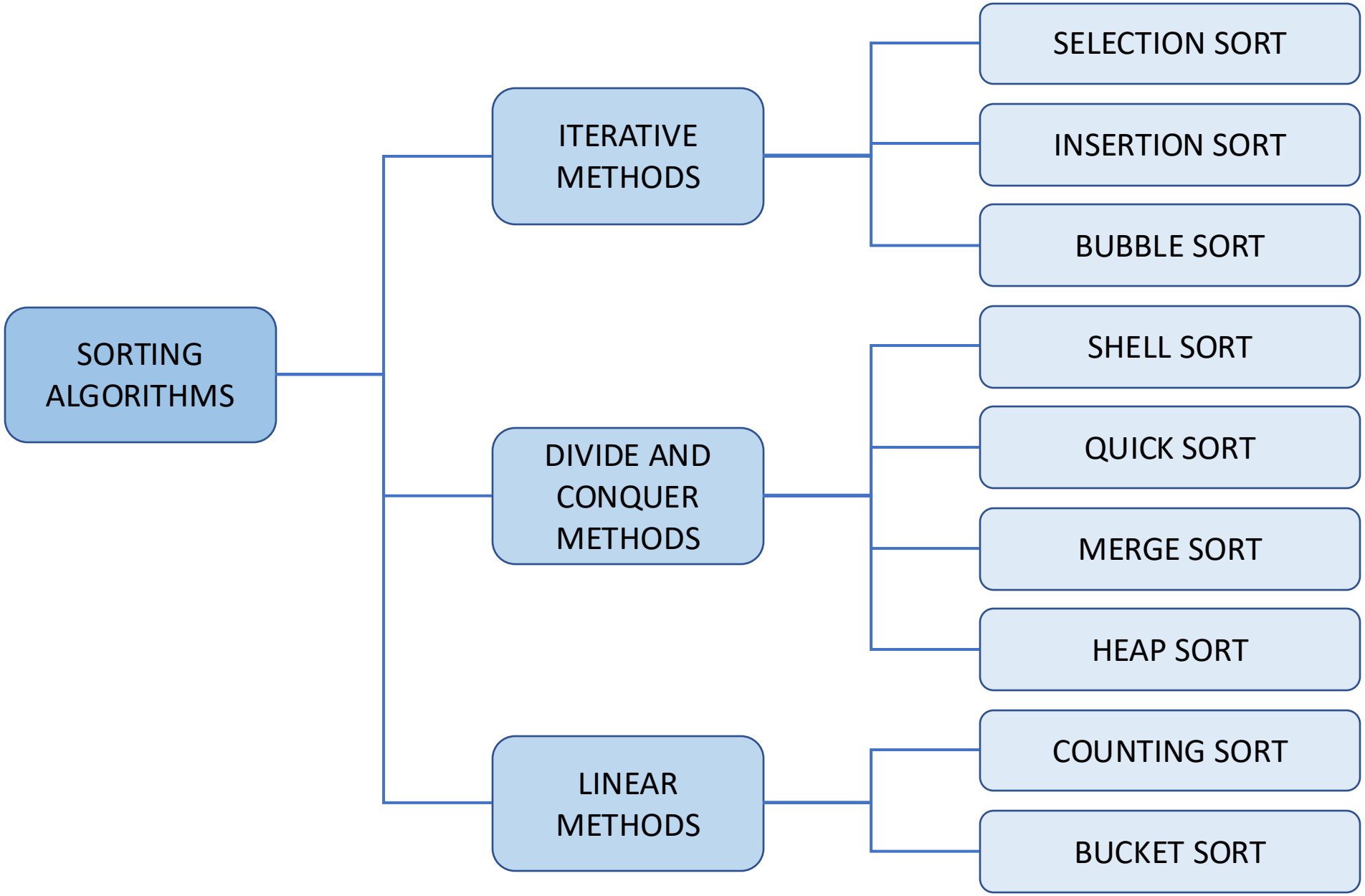


VŨ NGỌC THANH SANG

KHOA CÔNG NGHỆ THÔNG TIN

ĐẠI HỌC SÀI GÒN

I – INTRODUCTION



II – SORTING ALGORITHMS – SHELL SORT

Ý tưởng cho sắp xếp tăng dần một mảng có N phần tử.

- Việc sắp xếp các phần tử của mảng ban đầu có thể hiệu quả hơn nếu mảng đã được sắp xếp một phần nào đó.
- Chia nhỏ mảng ban đầu thành các mảng con và sắp xếp chúng. Như vậy thì việc sắp xếp mảng ban đầu sẽ gần với trường hợp tốt nhất hơn

```
divide data into h subarrays;
for i = 1 to h
   sort subarray data_i;
sort array data;
```

- Nếu h quá nhỏ, thì dữ liệu mảng con của dữ liệu mảng có thể quá lớn và thuật toán sắp xếp cũng có thể không hiệu quả.
- Mặt khác, nếu h quá lớn, thì có quá nhiều mảng con nhỏ được tạo ra và mặc dù chúng đã được sắp xếp, nhưng về cơ bản nó không làm thay đổi thứ tự tổng thể của dữ liệu.

II – SORTING ALGORITHMS – SHELL SORT

- Để giải quyết vấn đề chúng ta chọn vài giá trị của h và thực hiện quy trình như nhau

```
determine numbers h_t ... h_1 of ways of dividing array data into subarrays;
for (h=h_t; t > 1; t--, h=h_t)
    divide data into h subarrays;
    for i = 1 to h
        sort subarray data_i;
sort array data;
```

- Việc sắp xếp có thể được thực hiện bằng một trong các thuật toán sắp xếp cơ bản, tuy nhiên sắp xếp chèn thường được sử dụng nhất.
- Các phần tử ở xa nhau sẽ được so sánh với nhau trước

II – SORTING ALGORITHMS – SHELL SORT

- Có h_t mảng con, và với mỗi mảng $h = 1, ..., h_t$ ta có:

$$data_{h_t h}[i] = data[h_t i + (h - 1)]$$

- Ví dụ: nếu $h_t = 3$, mảng data được chia nhỏ thành 3 mảng con $data_1, data_2, data_3$

$data_{31}[0] = data[0], data_{31}[1] = data[3], ..., data_{31}[i] = data[3*i], ...$
$data_{32}[0] = data[1], data_{32}[1] = data[4], ..., data_{32}[i] = data[3*i+1], ...$
$data_{33}[0] = data[2], data_{33}[1] = data[5], ..., data_{33}[i] = data[3*i+2], ...$

- Sau đó các mảng con được tiếp tục chia nhỏ với $h_{t-1} < h_t$ cho tới khi không thể chia nhỏ nữa.

II – SORTING ALGORITHMS – SHELL SORT

- Sắp xếp mảng [10 8 6 20 4 3 22 1 0 15 16] sử dụng Shell sort

`data` before 5-sort	10	8	6	20	4	3	22	1	0	15	16
Five subarrays before sorting	10	—	—	—	—	3	—	—	—	—	16
		8	—	—	—	—	22				
			6	—	—	—	—	1			
				20	—	—	—	—	0		
					4	—	—	—	—	15	
Five subarrays after sorting	3	—	—	—	—	10	—	—	—	—	16
		8	—	—	—	—	22				
			1	—	—	—	—	6			
				0	—	—	—	—	20		
					4	—	—	—	—	15	
`data` after 5-sort and before 3-sort	3	8	1	0	4	10	22	6	20	15	16

II – SORTING ALGORITHMS – SHELL SORT

- Sắp xếp mảng [10 8 6 20 4 3 22 1 0 15 16] sử dụng Shell sort

Three subarrays before sorting	3	—	—	0	—	—	22	—	—	15	
		8	—	—	4	—	—	6	—	—	16
			1	—	—	10	—	—	20		
Three subarrays after sorting	0	—	—	3	—	—	15	—	—	22	
		4	—	—	6	—	—	8	—	—	16
			1	—	—	10	—	—	20		
`data` after 3-sort and before 1-sort	0	4	1	3	6	10	15	8	20	22	16
`data` after 1-sort	0	1	3	4	6	8	10	15	16	20	22

II – SORTING ALGORITHMS – SHELL SORT

- Bất kì chuỗi nào bắt đầu bằng $h_1 = 1$ cũng có thể sử dụng.

  ví dụ như 1, 2 , 4, 8, ... hoặc 1, 3, 6, 9...

- Dựa trên các phương pháp thực nghiệm (empirical approach), chuỗi nên được chọn dựa theo các điều kiện sau

  1. Bắt đầu bằng $h_1 = 1$
  2. Bước nhảy $h_{i+1} = 3h_i + 1$
  3. Kết thúc bằng h_t với $h_{t+2} \geq n$

- Ví dụ n = 10000, ta có chuỗi sau 1,4, 13, 40, 121, 364, 1093, 3280

II – SORTING ALGORITHMS – SHELL SORT

- Cốt lõi của thuật toán sắp xếp shell là chia một mảng cho trước thành các mảng con bằng cách lấy các phần tử cách nhau h vị trí. Ba yếu tố sau sẽ thay đổi mỗi lần triển khai thuật toán:
    1. Chuỗi ht
    2. Thuật toán sắp xếp cơ bản để sắp xếp tất cả các lần lặp trừ lần cuối cùng.
    3. Thuật toán sắp xếp cơ bản để sắp xếp tất chỉ cho lần cuối cùng

```
void Shellsort(int data[], int n) {
   int i, j, hCnt, h;
   int increments[20], k;
   // create an appropriate number of increments h
   for (h = 1, i = 0; h < n; i++) {
      increments[i] = h;
      h = 3*h + 1;
   }
   // loop on the number of different increments h
   for (i--; i >= 0; i--) {
      h = increments[i];
   // loop on the number of subarrays h-sorted in ith pass
      for (hCnt = h; hCnt < n; hCnt++) {
      // insertion sort for subarray containing every hth element of
         for (j = hCnt; j < n; ) {   // array data
            int tmp = data[j];
            k = j;
            while (k-h >= 0 && tmp < data[k-h]) {
               data[k] = data[k-h];
               k -= h;
            }
            data[k] = tmp;
            j += h;
         }
      }
   }
}
```

Độ phức tạp của thuật toán

- Độ phức tạp của trường hợp xấu nhất: O (n^2)
- Độ phức tạp của trường hợp tốt nhất: O (n * log n)
- Độ phức tạp của trường hợp trung bình: O (n * log n) – Xấp xỉ O ($n^{1,25}$).

II – SORTING ALGORITHMS – QUICK SORT

Ý tưởng cho sắp xếp tăng dần một mảng có N phần tử.

- Tương tự như shell sort bằng cách chia mảng ban đầu thành các mảng con, sắp xếp chúng riêng biệt và sau đó chia chúng một lần nữa để sắp xếp các mảng con mới cho đến khi toàn bộ mảng được sắp xếp.
- Mảng ban đầu được chia thành hai mảng con, mảng đầu chứa các giá trị nhỏ hơn giá trị pivot và mảng sau chứa các giá trị lớn hơn hoặc bằng giá trị pivot.
- Quá trình được lặp lại với các mảng con.

II – SORTING ALGORITHMS – QUICK SORT

```
quicksort(array[])
   if length(array) > 1
      choose pivot; // partition array into subarray1 and subarray2
      while there are elements left in array
         include element either in subarray1 = {el: el ≤ bound}
         or in subarray2 = {element: element ≥ pivot};
      quicksort(subarray1);
      quicksort(subarray2);
```

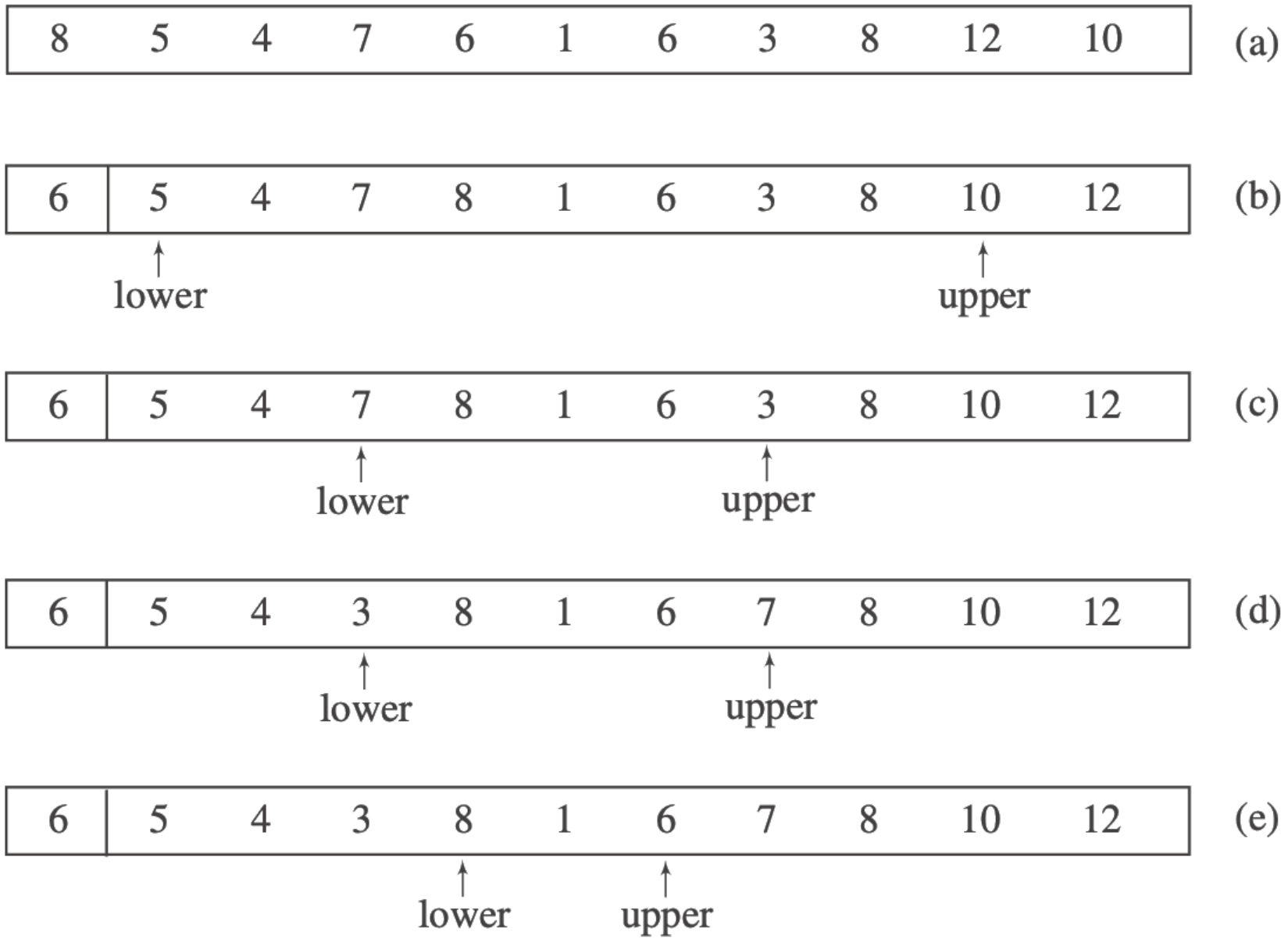
II – SORTING ALGORITHMS – QUICK SORT

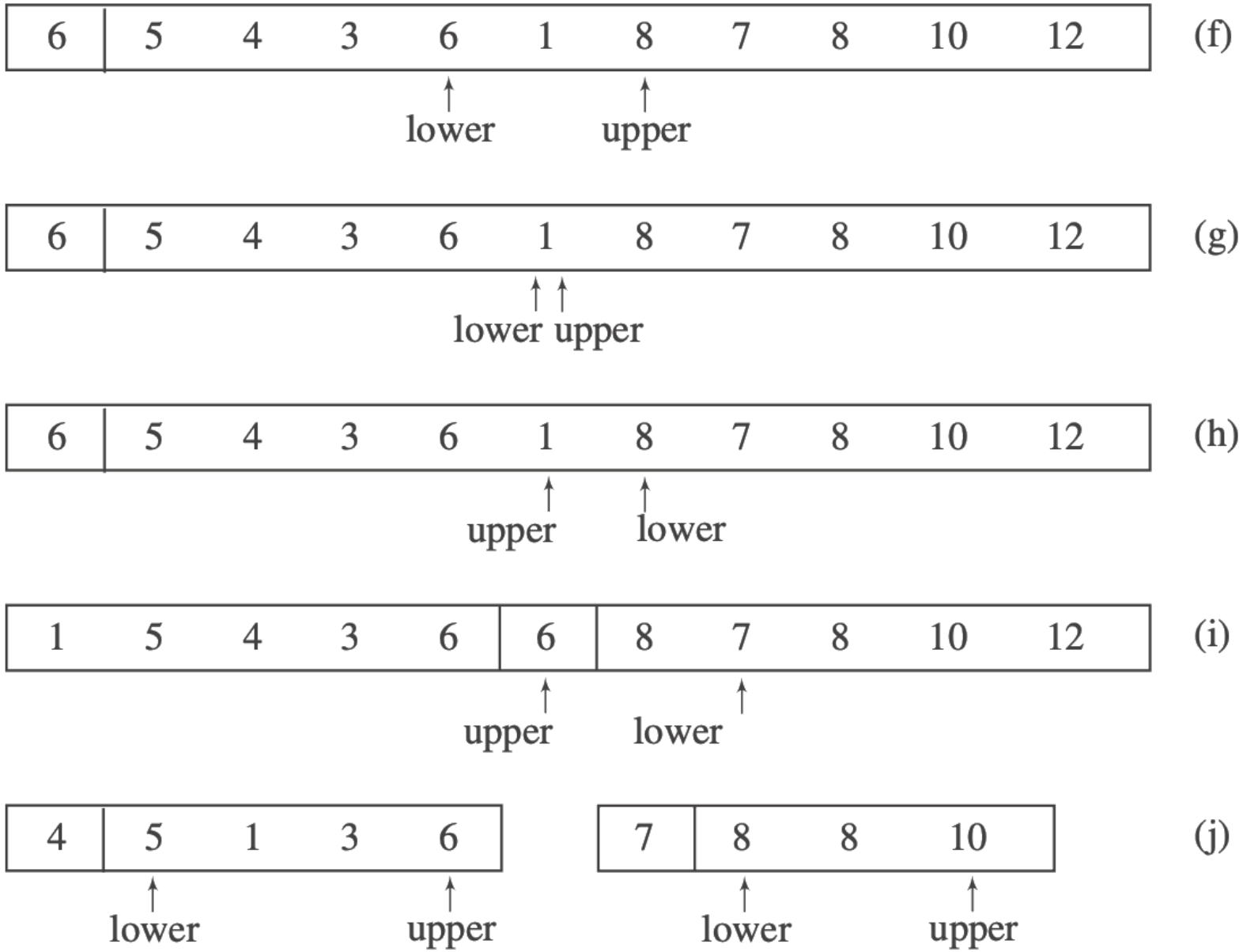
- Chọn giá trị pivot là để mảng được chia thành hai mảng con riêng biệt xấp xỉ nhau là một nhiệm vụ quan trọng để dữ liệu được cân bằng.
- Nhiều các để chọn giá trị pivot đã được đề xuất như chọn phần tử đầu tiên, phần tử ở giữa mảng.

II – SORTING ALGORITHMS – QUICK SORT

```
void quicksort(int data[], int first, int last) {
   int lower = first+1, upper = last;
   swap(data[first],data[(first+last)/2]);
   int pivot = data[first];
   while (lower <= upper) {
      while (pivot > data[lower])
         lower++;
      while (pivot < data[upper])
         upper--;
      if (lower < upper)
         swap(data[lower++],data[upper--]);
      else lower++;
   }
   swap(data[upper],data[first]);
   if (first < upper-1)
      quicksort (data,first,upper-1);
   if (upper+1 < last)
      quicksort (data,upper+1,last);
}
```

II – SORTING ALGORITHMS – QUICK SORT




6 5 4 3 6 1 8 7 8 10 12
(f)
lower
upper
6 5 4 3 6 1 8 7 8 10 12
(g)
lower upper
6 5 4 3 6 1 8 7 8 10 12
(h)
upper
lower
1 5 4 3 6 6 8 7 8 10 12
(i)
upper
lower
4 5 1 3 6
7 8 8 10
(j)
lower
upper
lower
upper

II – SORTING ALGORITHMS – QUICK SORT

```
void quickSort(int arr[], int left, int right) {
    if (left < right) {
        // pi is where the pivot is at
        int pi = partition(arr, left, right);

        // Separately sort elements before and after partition
        quickSort(arr, left, pi - 1);
        quickSort(arr, pi + 1, right);
    }
}
```

II – SORTING ALGORITHMS – QUICK SORT

```
int partition(int arr[], int left, int right) {
    int pivotInd = choosePivot(left, right); // Index of pivot
    swap(arr[right], arr[pivotInd]); // put the pivot at the end
    int pivot = arr[right]; // Pivot
    int i = (left - 1); // All the elements less than or equal to the
    // pivot go before or at i

    for (int j = left; j <= right - 1; j++) {
        if (arr[j] <= pivot) {
            i++; // increment the index
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[right]); // Putting the pivot back in place
    return (i + 1);
}
```

II – SORTING ALGORITHMS – QUICK SORT

- Trường hợp xấu nhất xảy ra khi hàm quicksort() chọn phần tử có giá trị nhỏ nhất hoặc lớn nhất làm pivot. Thuật toán sẽ có độ phức tạp $O(n^2)$
- Trường hợp tốt nhất xảy ra khi giá trị pivot chia mảng thành hai mảng con có độ dài xấp xỉ n/2. Thuật toán sẽ có độ phức tạp O(nlogn)
- Trường hợp trung bình sẽ có độ phức tạp O(nlogn)

II – SORTING ALGORITHMS – MERGE SORT

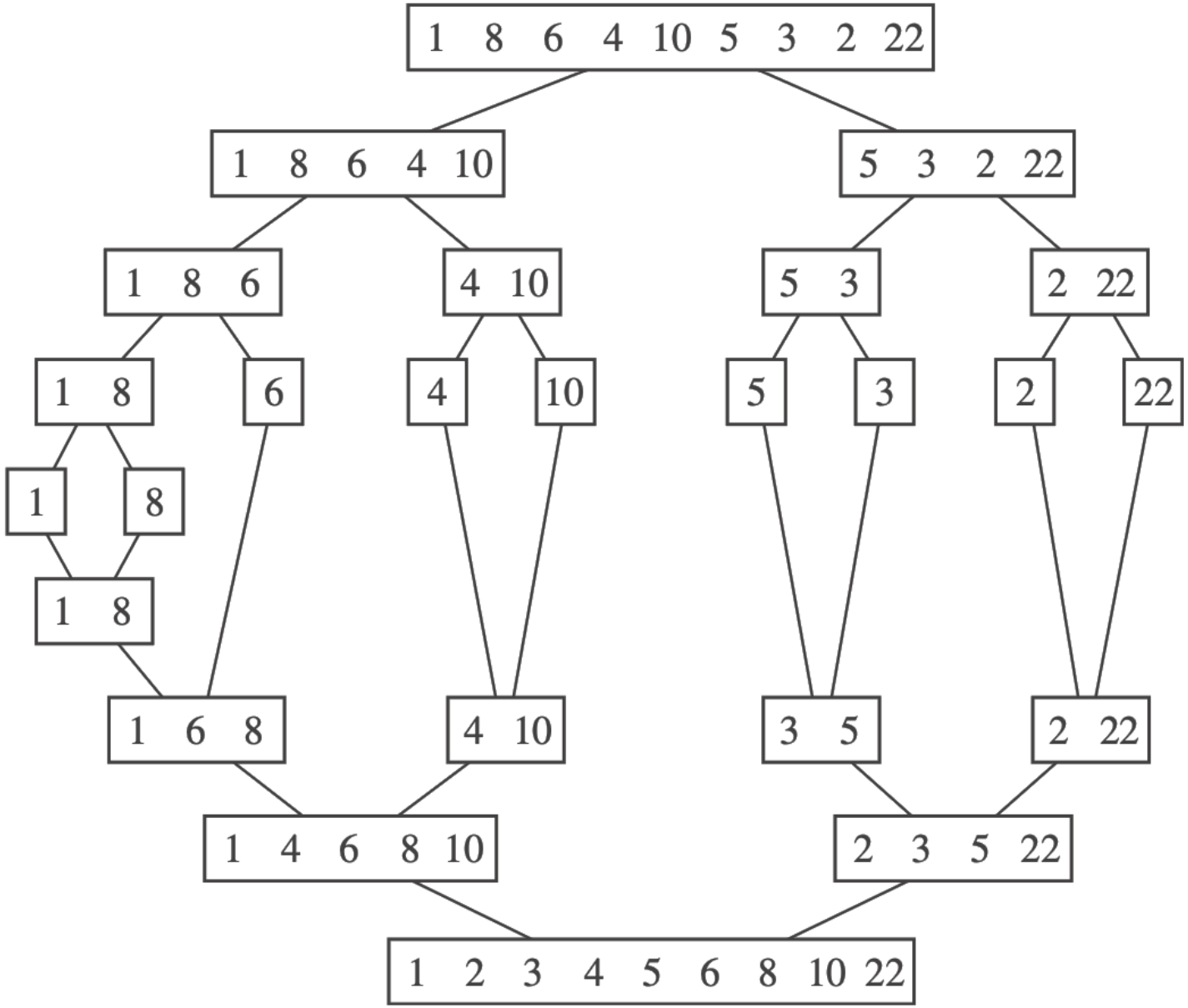
- Một nhược điểm lớn của Quicksort là nó có độ phức tạp $O(n^2)$ trong trường hợp xấu nhất do quá trình phân vùng gặp khó khăn.
- Có rất nhiều kỹ thuật giá trị pivot để cố gắng giải quyết vấn đề này. Tuy nhiên, không có gì đảm bảo rằng các cách tiếp cận sẽ dẫn đến việc phân chia thành công các mảng có kích thước bằng nhau.
- Một cách tiếp cận khác hoàn toàn đơn giản là phân vùng nhiều nhất có thể và tập trung vào việc hợp nhất các mảng đã sắp xếp
- Thao tác quan trọng trong hợp nhất là hợp nhất các nửa đã được sắp xếp của mảng thành một mảng duy nhất
- Tất nhiên, các nửa này phải được sắp xếp, điều này xảy ra bằng cách hợp nhất các nửa đã sắp xếp của các nửa này

II – SORTING ALGORITHMS – MERGE SORT

- Quá trình chia mảng thành hai nửa sẽ dừng lại khi mỗi mảng có ít hơn hai phần tử.
- Do sự tương tự với quá trình phân vùng nhanh chóng, điều này cũng có thể được thực hiện một cách đệ quy, như sau:

```
mergesort (data[], first, last)
   if first < last
      mid = (first + last) / 2
      mergesort(data, first, mid);
      mergesort(data, mid+1, last);
      merge(data, first, last);
```

II – SORTING ALGORITHMS – MERGE SORT



II – SORTING ALGORITHMS – MERGE SORT

```
// Divide the array into two subarrays, sort them and merge them
void mergeSort(int arr[], int l, int r) {
  if (l < r) {
    // m is the point where the array is divided into two subarrays
    int m = l + (r - l) / 2;

    mergeSort(arr, l, m);
    mergeSort(arr, m + 1, r);

    // Merge the sorted subarrays
    merge(arr, l, m, r);
  }
}
```

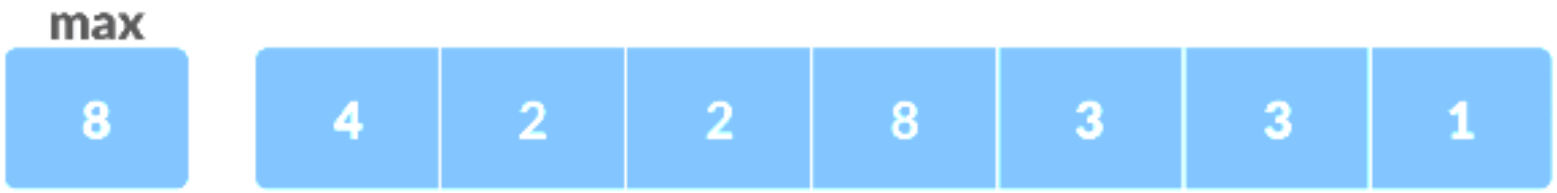
II – SORTING ALGORITHMS – MERGE SORT

- Trường hợp xấu nhất: O(nlogn)
- Trường hợp tốt nhất: O(nlogn)
- Trường hợp trung bình: O(nlogn)

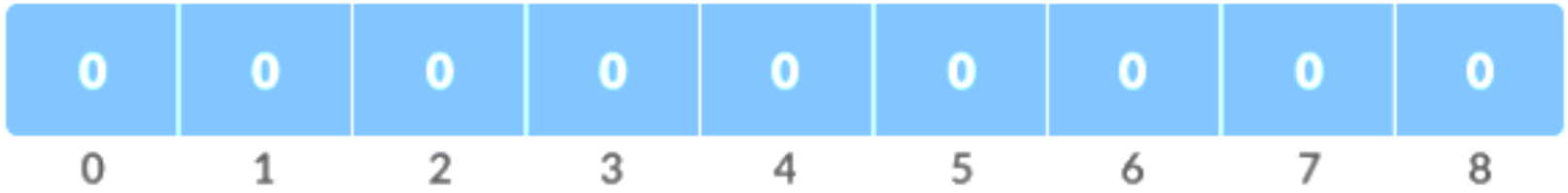
II – SORTING ALGORITHMS – COUNTING SORT

- Đầu tiên, sắp xếp đếm sẽ đếm số lần xuất hiện của mỗi phần tử trong mảng data[] bằng cách sử dụng mảng đếm count [].
- Sau đó, các mảng đếm cho biết số lượng phần tử ≤ i. Theo cách này, số đếm [i] -1 cho biết vị trí chính xác của i trong mảng data[].

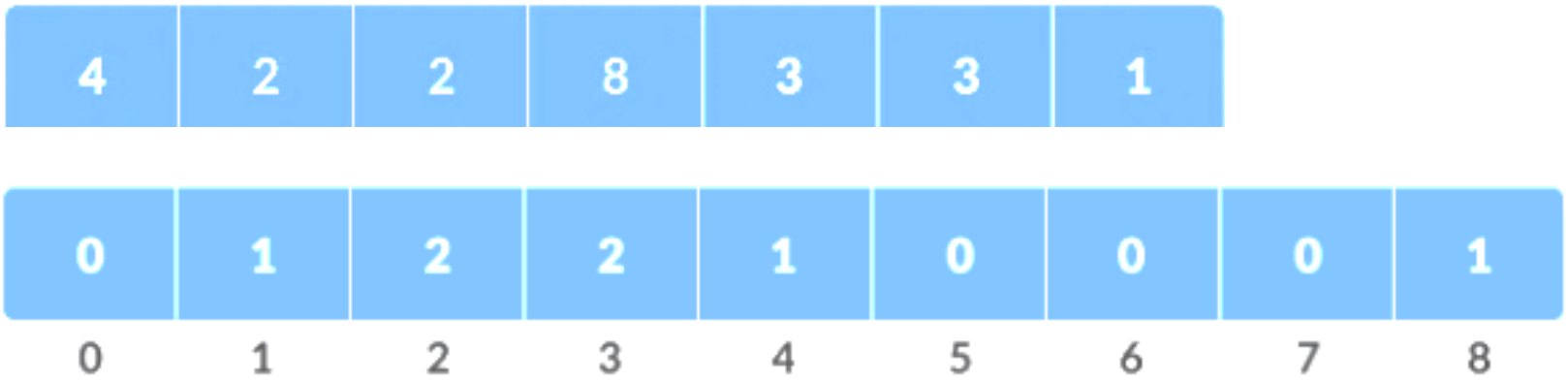
- **Bước 1**: Tìm phần tử lớn nhất (gọi là max) trong mảng đã cho.



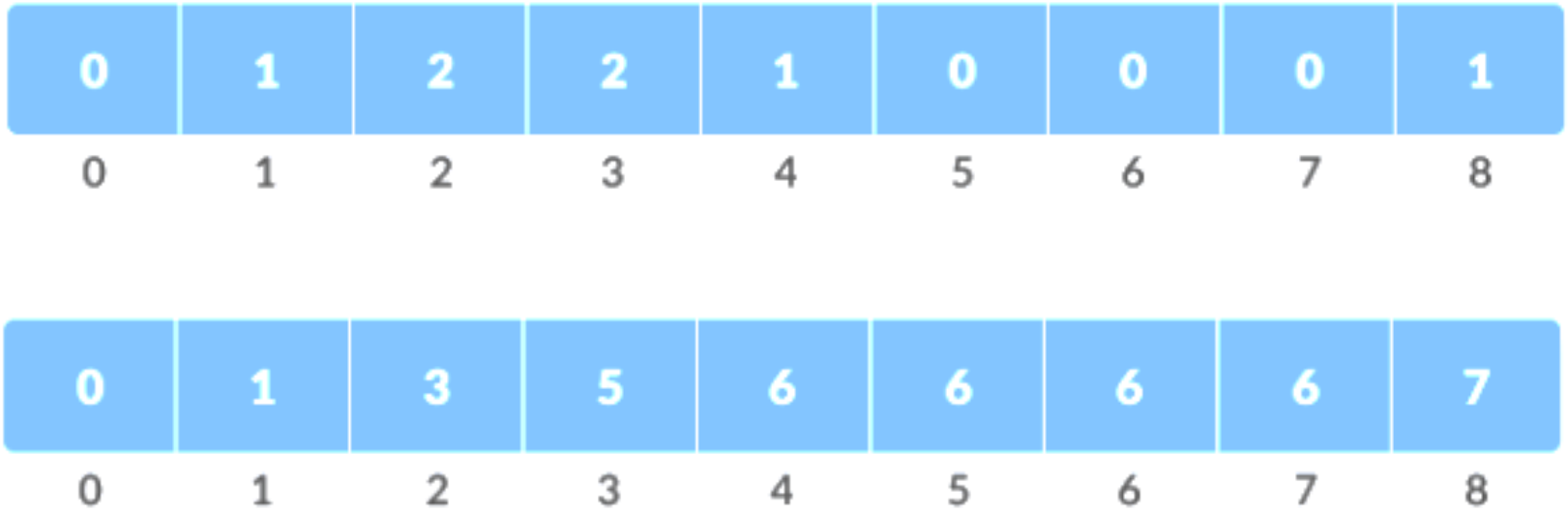
- **Bước 2**: Khởi tạo một mảng có độ dài là max+1 với tất cả các phần tử bằng 0. Mảng này được sử dụng để lưu trữ số lần xuất hiện của từng phần tử trong mảng.



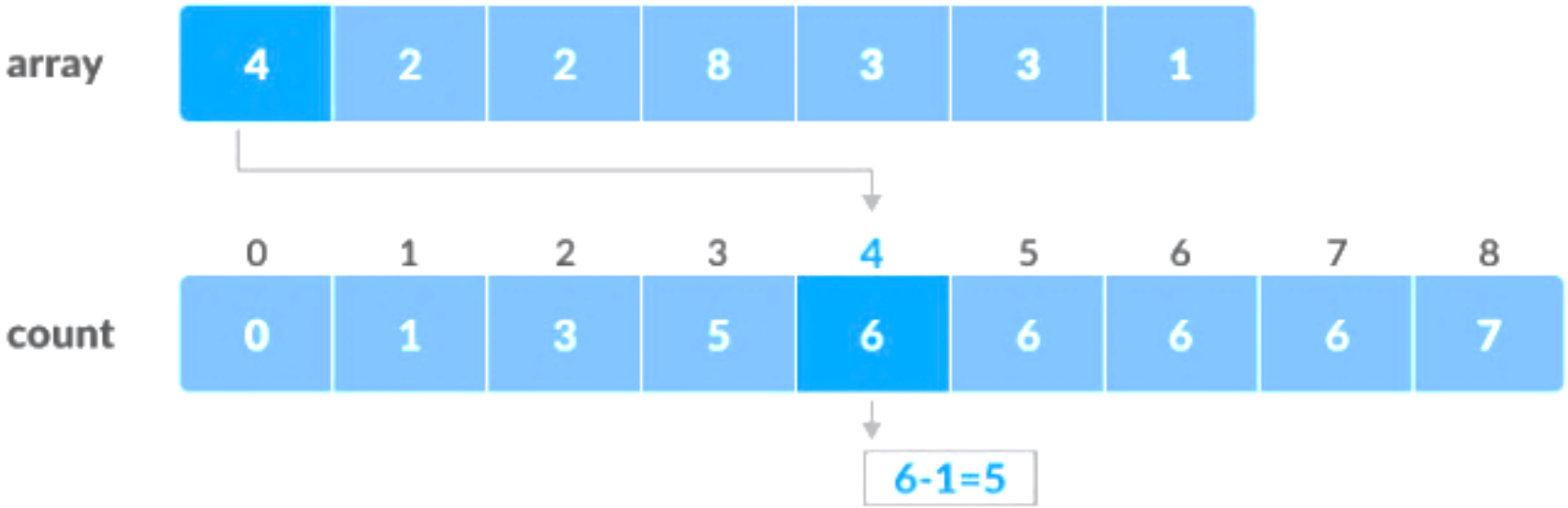
- **Bước 3**: Lưu trữ số lần xuất hiện của từng phần tử trong mảng vào vị trí tương ứng của mảng đếm (count array).
- Ví dụ: nếu số lần xuất hiện của phần tử 3 là 2, thì 2 được lưu trữ tại vị trí thứ 3 của mảng đếm. Nếu phần tử "5" không có trong mảng, thì 0 được lưu trữ tại vị trí thứ 5.



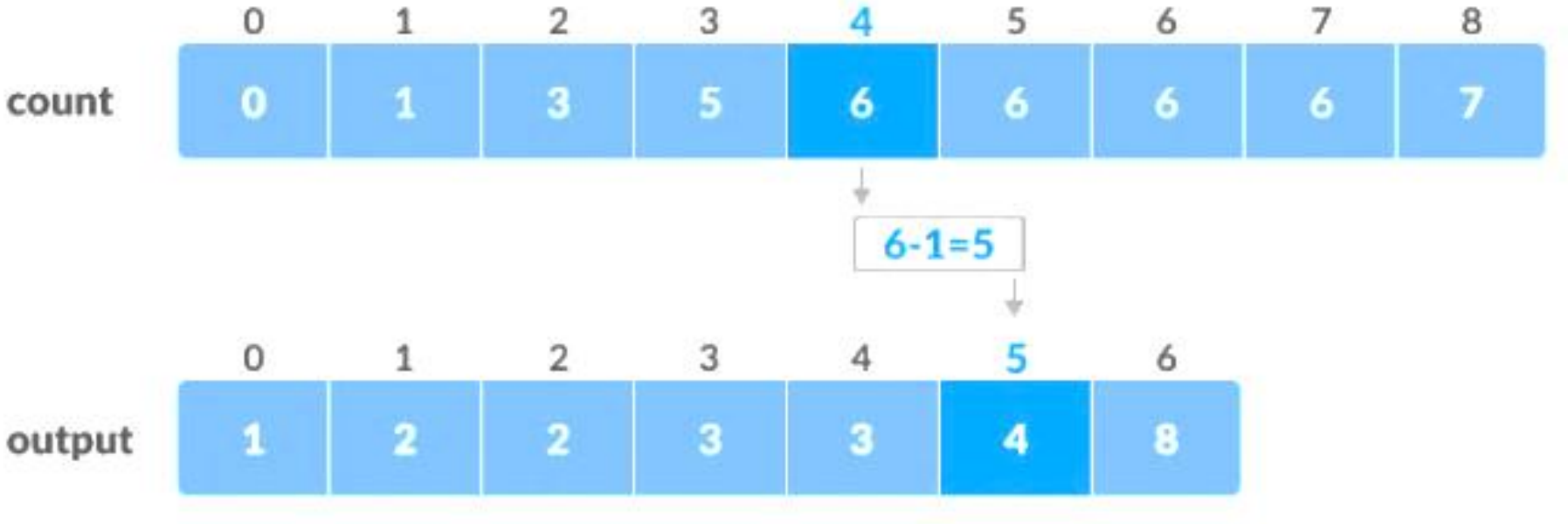
- **Bước 4**: Lưu trữ tổng tích lũy của các phần tử trong mảng đếm. Điều này giúp đặt các phần tử vào vị trí chính xác trong mảng đã sắp xếp.



- **Bước 5**: Tìm chỉ mục của mỗi phần tử trong mảng gốc trong mảng đếm. Giá trị này là số đếm tích lũy.



- **Bước 6**: Sau khi đặt từng phần tử vào vị trí đúng của nó, giảm số lần xuất hiện của nó đi một đơn vị



```
01 function counting_sort(array):
02     // Find the maximum element in the array
03     max_element = max(array)
04
05     // Initialize the count array with all zeros
06     count = [0] * (max_element + 1)
07
08     // Find the total count of each unique element and
09     // store the count at the jth index in the count array
10     for element in array:
11         count[element] += 1
```

```
12
13      // Find the cumulative sum and
14      // store it in the count array itself
15      for i in range(1, max_element + 1):
16          count[i] += count[i - 1]
17
18      // Restore the elements to the array and
19      // decrease the count of each element restored by 1
20      sorted_array = [0] * len(array)
21      for element in array:
22          sorted_array[count[element] - 1] = element
23          count[element] -= 1
```

II – SORTING ALGORITHMS – COUNTING SORT

```
01 void counting_sort(int arr[], int n) {
02     // Find the maximum element in the array
03     int max_element = arr[0];
04     for (int i = 1; i < n; i++) {
05         if (arr[i] > max_element) {
06             max_element = arr[i];
07         }
08     }
09
10     // Initialize the count array with all zeros
11     int count[max_element + 1] = {0};
12
13     // Find the total count of each unique element and
14     // store the count at the jth index in the count array
15     for (int i = 0; i < n; i++) {
16         count[arr[i]]++;
17     }
18
```

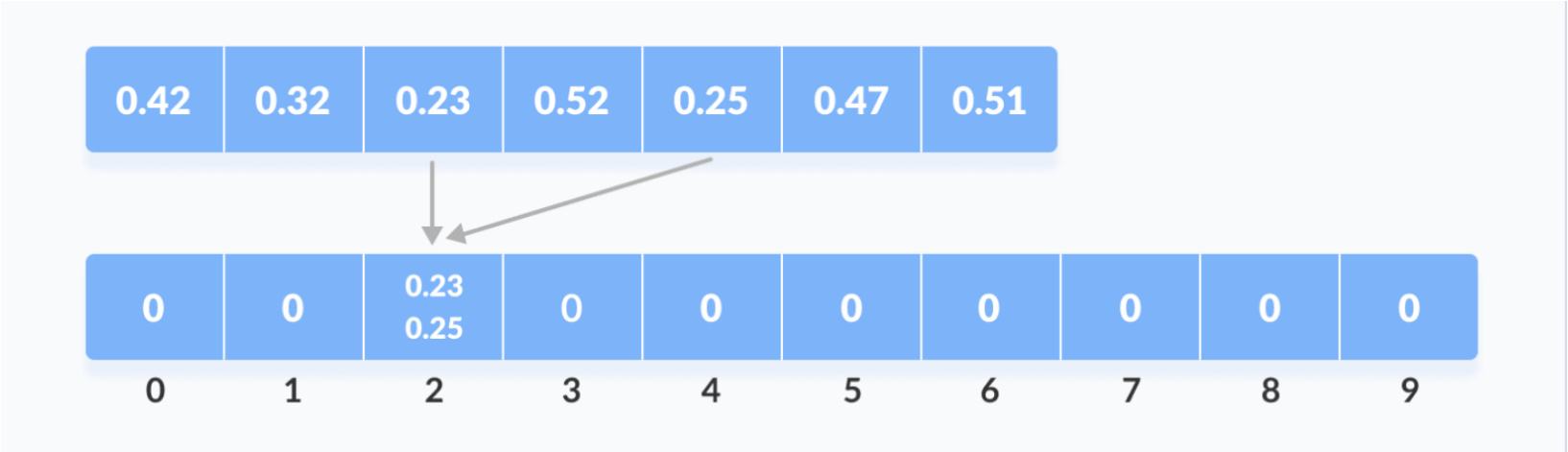
```
19      // Find the cumulative sum and
20      // store it in the count array itself
21      for (int i = 1; i <= max_element; i++) {
22          count[i] += count[i - 1];
23      }
24
25      // Restore the elements to the array and
26      // decrease the count of each element restored by 1
27      int sorted_array[n];
28      for (int i = n - 1; i >= 0; i--) {
29          sorted_array[count[arr[i]] - 1] = arr[i];
30          count[arr[i]]--;
31      }
32
33      // Copy the sorted array back to the input array
34      for (int i = 0; i < n; i++) {
35          arr[i] = sorted_array[i];
36      }
37  } // end counting_sort()
```

- Trường hợp xấu nhất: O(n + k)
- Trường hợp tốt nhất: O(n + k)
- Trường hợp trung bình: O(n + k)

II – SORTING ALGORITHMS – BUCKET SORT

- Quá trình sắp xếp khối (**bucket sort**) có thể được hiểu là một phương pháp phân tán-tập hợp (scatter-gather). Ở đây, các phần tử được phân tán vào các khối, sau đó các phần tử trong mỗi khối được sắp xếp. Cuối cùng, các phần tử được tập hợp theo thứ tự.

II – SORTING ALGORITHMS – BUCKET SORT



II – SORTING ALGORITHMS – BUCKET SORT

- Sắp xếp các phần tử của mảng sau theo thứ tự tăng dần

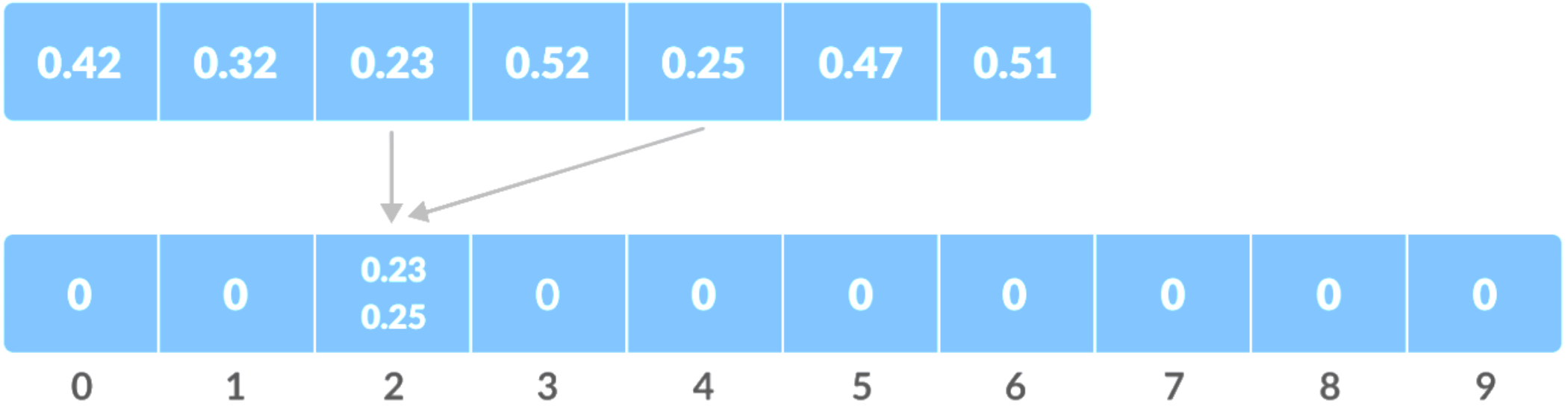
0.42	0.32	0.23	0.52	0.25	0.47	0.51

- **Bước 1**: Tạo một mảng có kích thước 10 và sử dụng mỗi phần tử của mảng này như là một khối để lưu trữ các phần tử cần được sắp xếp.

0	0	0	0	0	0	0	0	0	0

- **Bước 2**: Chèn các phần tử vào các khối tương ứng với phạm vi của khối đó.
- Phạm vi các khối [0,1), [1,2), [2,3), …, [(n-1), n).
- Ví dụ input = 0.23 => input*arraySize = 0.23*10=2.3.

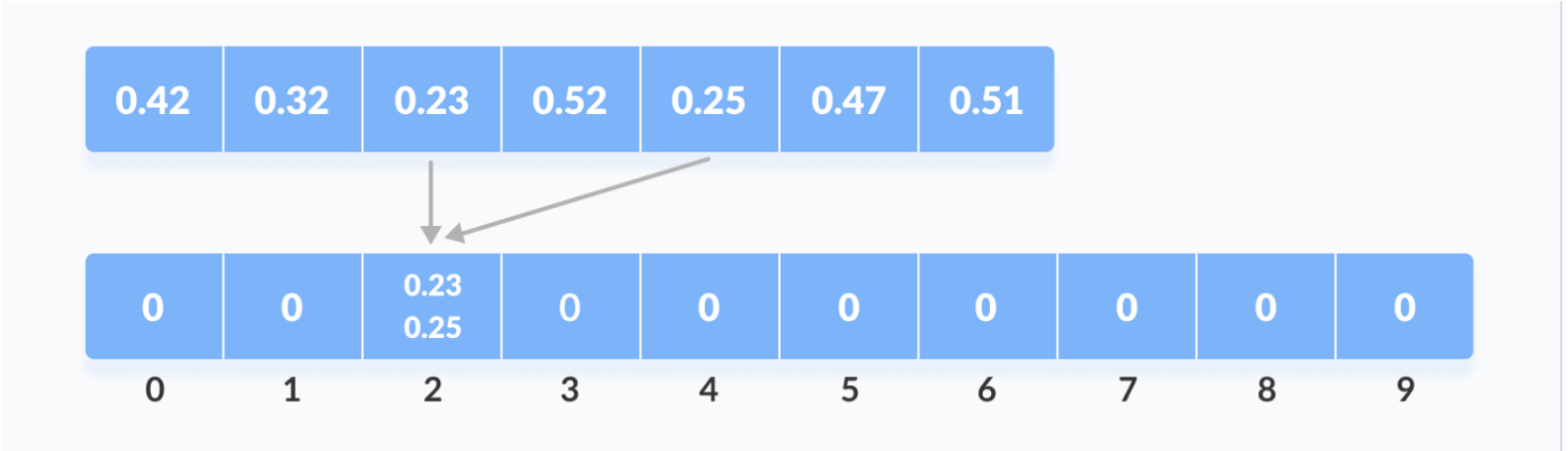
  int(2.3) = 2 => 0.23 được chèn vào khối 2.



- **Bước 2**: Chèn các phần tử vào các khối tương ứng với phạm vi của khối đó.

0	1	2	3	4	5	6	7	8	9
0	0	0.23 0.25	0.32	0.42 0.47	0.52 0.51	0	0	0	0

II – SORTING ALGORITHMS – BUCKET SORT



EXERCISES

Bài 1: Tìm hiểu thêm về cách chọn các chuỗi giá trị cho thuật toán shell sort như: Knuth's increments, Sedgewick's increments, Hibbard's increments, Papernov & Stasevich increment, Pratt.

Bài 2: Đề xuất cách chọn giá trị pivot để hạn chế trường hợp xấu nhất xảy ra khi triển khai thuật toán quick sort

Bài 3: Viết chương trình thực hiện shell sort, và quick sort.

EXERCISES

Bài 4: Viết chương trình thực hiện merge sort, và count sort sử dụng đệ quy và không sử dụng đệ quy.

Bài 5: Viết chương trình thực hiện bucket sort để sắp xếp tăng dần mảng

```
arr = {0.897, 0.565, 0.656, 0.1234, 0.665, 0.3434}
```

THANK YOU FOR YOUR ATTENTIONS