Sorting Algorithms

Outline

1) Sorting Definition
2) In-place Sorting and Not In-place Sorting
3) Stable and Not Stable Sorting
4) Important Terms
5) Types of Sorting

Sorting Definition

- Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order.

- The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats.

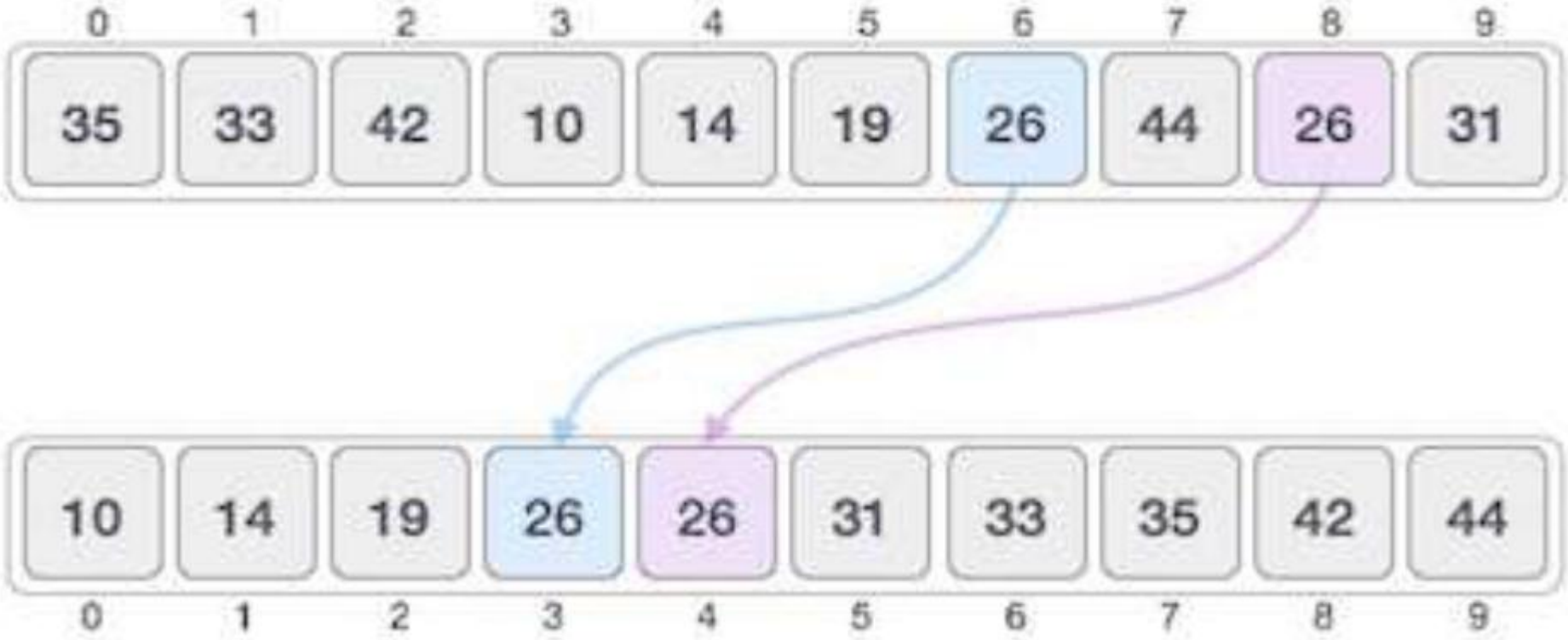
In-place Sorting and Not In-place Sorting

- Sorting algorithms may require some extra space for comparison and temporary storage of few data elements. These algorithms do not require any extra space and sorting is said to happen in-place, or for example, within the array itself. This is called in-place sorting. Bubble sort is an example of in-place sorting.

- However, in some sorting algorithms, the program requires space which is more than or equal to the elements being sorted. Sorting which uses equal or more space is called not in-place sorting. Merge sort is an example of not-in-place sorting.

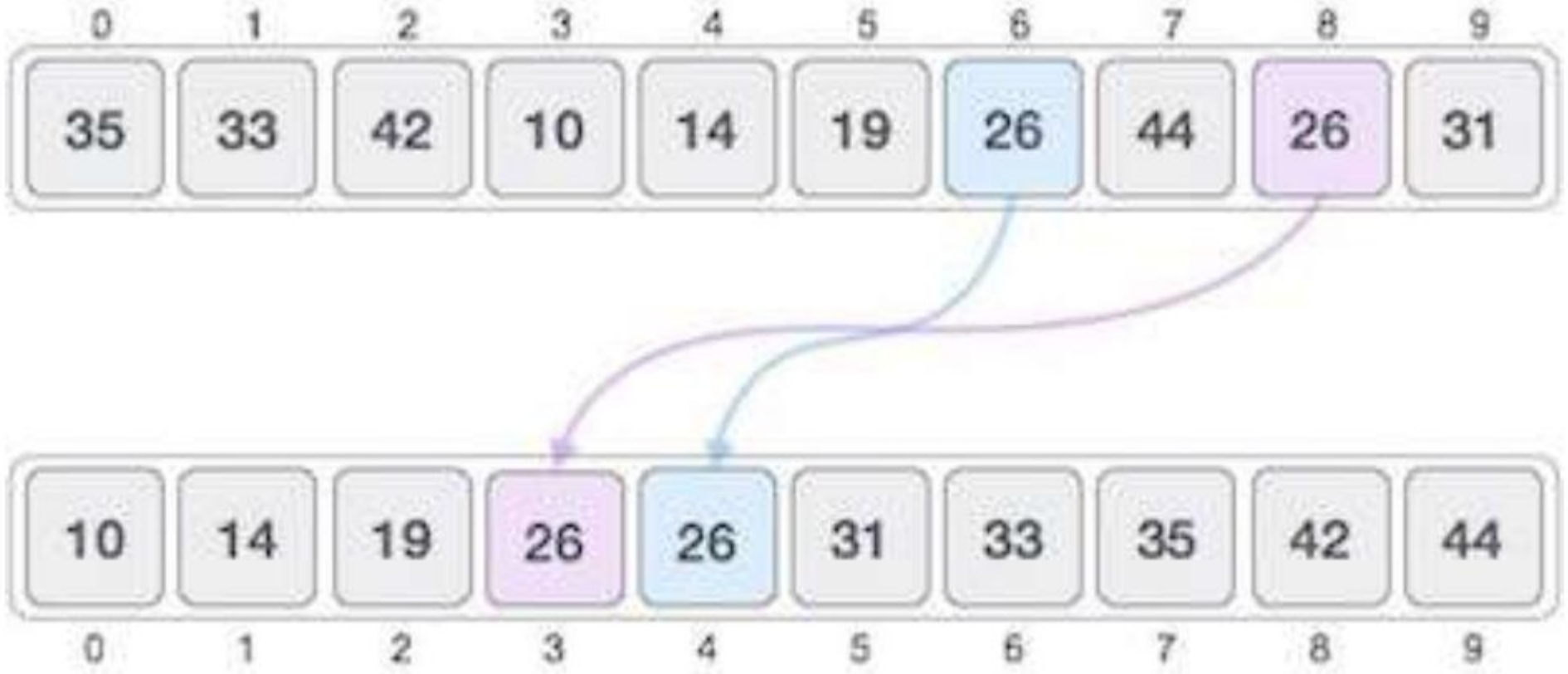
Stable and Not Stable Sorting

- If a sorting algorithm, after sorting the contents, does not change the sequence of similar content in which they appear, it is called stable sorting.



Stable and Not Stable Sorting

- If a sorting algorithm, after sorting the contents, changes the sequence of similar content in which they appear, it is called unstable sorting.



Important Terms

- Increasing Order
  - A sequence of values is said to be in increasing order, if the successive element is greater than the previous one.
  - For example, 1, 3, 4, 6, 8, 9 are in increasing order, as every next element is greater than the previous element.
- Decreasing Order
  - A sequence of values is said to be in decreasing order, if the successive element is less than the current one.
  - For example, 9, 8, 6, 4, 3, 1 are in decreasing order, as every next element is less than the previous element.

Important Terms

- Non-Increasing Order
  - A sequence of values is said to be in non-increasing order, if the successive element is less than or equal to its previous element in the sequence. This order occurs when the sequence contains duplicate values.
  - For example, 9, 8, 6, 3, 3, 1 are in non-increasing order, as every next element is less than or equal to (in case of 3) but not greater than any previous element.
- Non-Decreasing Order
  - A sequence of values is said to be in non-decreasing order, if the successive element is greater than or equal to its previous element in the sequence. This order occurs when the sequence contains duplicate values.
  - For example, 1, 3, 3, 6, 8, 9 are in non-decreasing order, as every next element is greater than or equal to (in case of 3) but not less than the previous one.

Types of Sorting

1) Bubble Sort
2) Selection Sort
3) Insertion Sort
4) Shell Sort
5) Merge Sort
6) Count Sort
7) Heap Sort

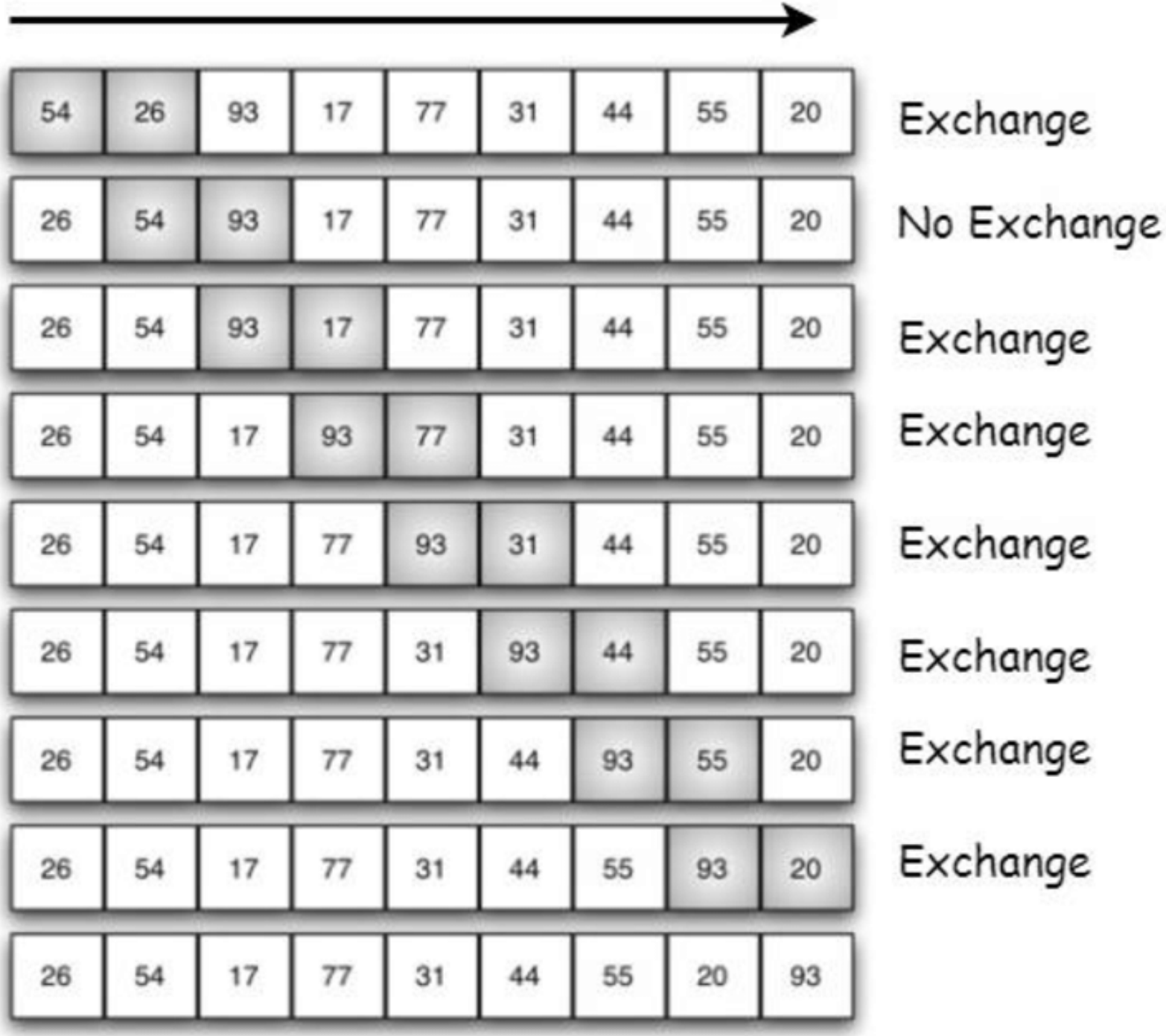
Bubble Sort

- Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.
- This algorithm is not suitable for large data sets as its average and worst case complexity are of O(n^2) where n is the number of items.
- Worst and Average Case Time Complexity: O(n^2). when array is reverse sorted.
- Best Case Time Complexity: O(n). when array is already sorted.
- Sorting In Place: Yes
- Stable: Yes

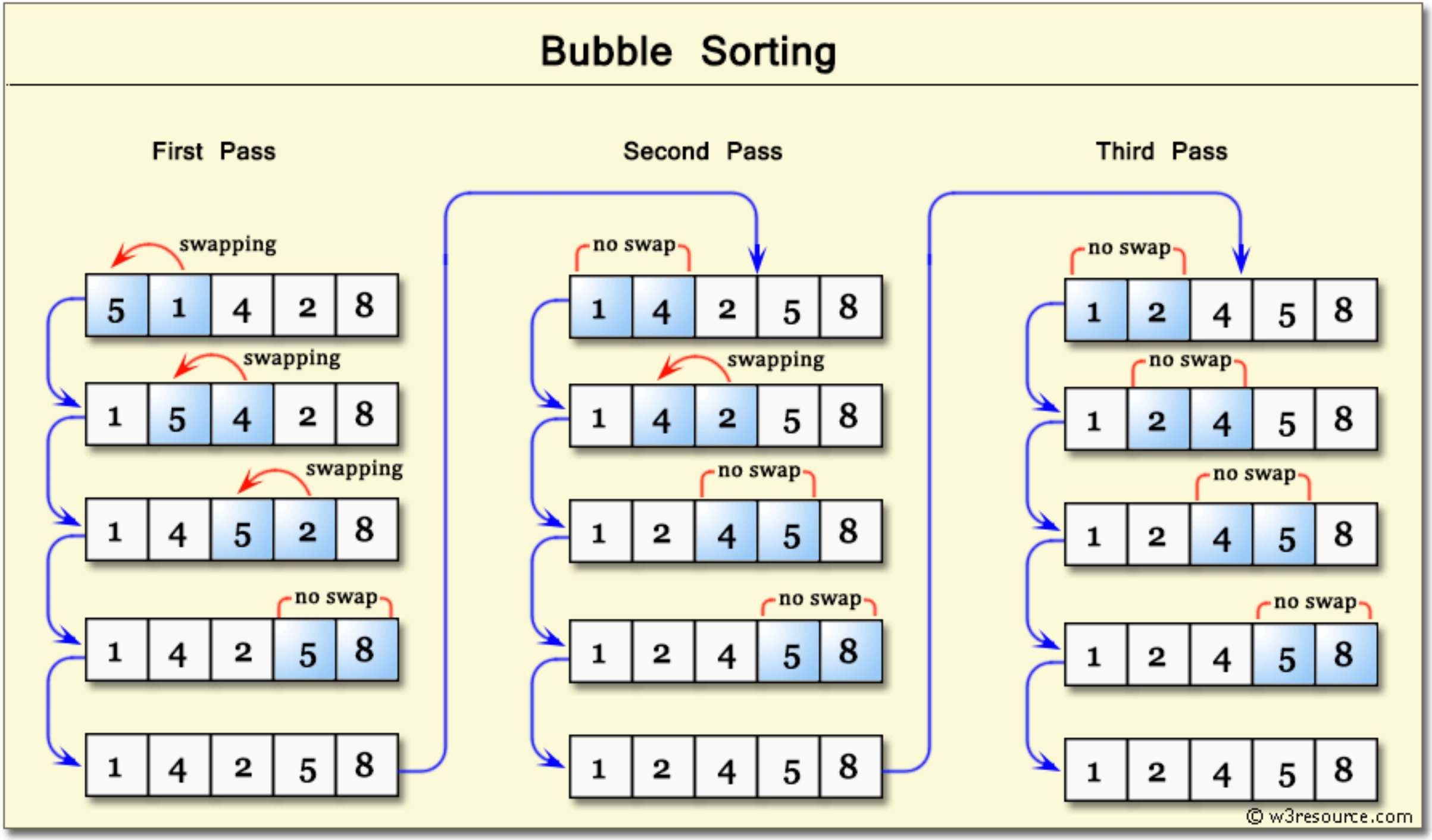
How bubble sort work?

i =	j	0	1	2	3	4	5	6	7
	0	5	3	1	9	8	2	4	7
	1	3	5	1	9	8	2	4	7
	2	3	1	5	9	8	2	4	7
	3	3	1	5	9	8	2	4	7
	4	3	1	5	8	9	2	4	7
	5	3	1	5	8	2	9	4	7
	6	3	1	5	8	2	4	9	7
i = 1	0	3	1	5	8	2	4	7	9
	1	1	3	5	8	2	4	7	
	2	1	3	5	8	2	4	7	
	3	1	3	5	8	2	4	7	
	4	1	3	5	2	8	4	7	
	5	1	3	5	2	4	8	7	
i = 2	0	1	3	5	2	4	7	8	
	1	1	3	5	2	4	7		
	2	1	3	5	2	4	7		
	3	1	3	2	5	4	7		
	4	1	3	2	4	5	7		
i = 3	0	1	3	2	4	5	7		
	1	1	3	2	4	5			
	2	1	2	3	4	5			
	3	1	2	3	4	5			
i = 4	0	1	2	3	4	5			
	1	1	2	3	4				
	2	1	2	3	4				
i = 5	0	1	2	3	4				
	1	1	2	3					
i = 6	0	1	2	3					
		1	2						

How bubble sort work?



Bubble Sorting



Bubble Sort Algorithm

- We assume list is an array of n elements. We further assume that swap function swaps the values of the given array elements.

1. Input n numbers of an array A
2. Initialize i = 0 and repeat through step 4 if (i < n)
3. Initialize j = 0 and repeat through step 4 if (j < n – i – 1)
4. If (A[j] > A[j + 1])
   (a) Swap = A[j]
   (b) A[j] = A[j + 1]
   (c) A[j + 1] = Swap
5. Display the sorted numbers of array A
6. Exit.

Selection Sort

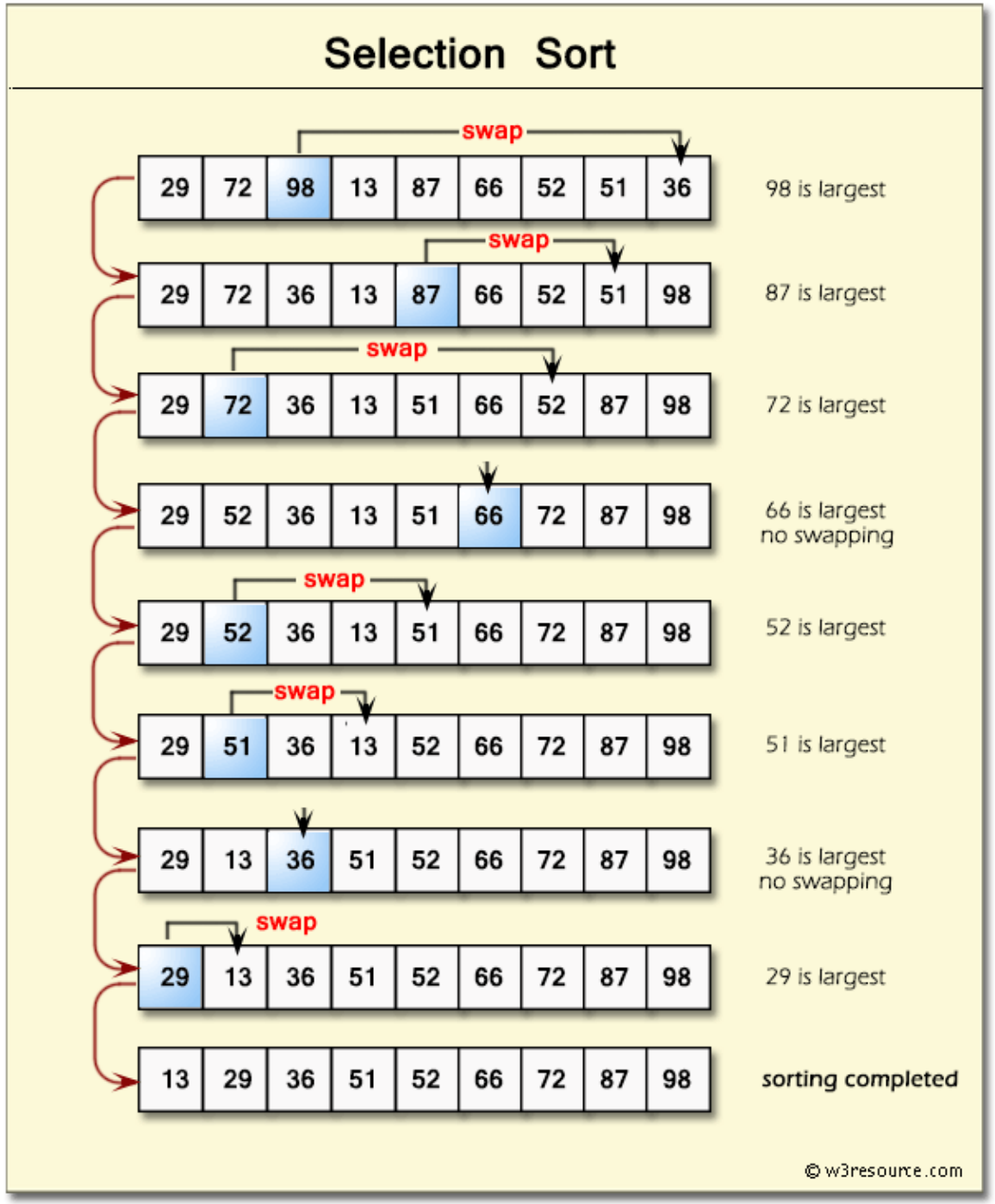
- Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end.
- Initially, the sorted part is empty and the unsorted part is the entire list. The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array.
- This process continues moving unsorted array boundary by one element to the right. This algorithm is not suitable for large data sets as its average and worst case complexities are of O(n^2), where n is the number of items.

How Selection sort work?

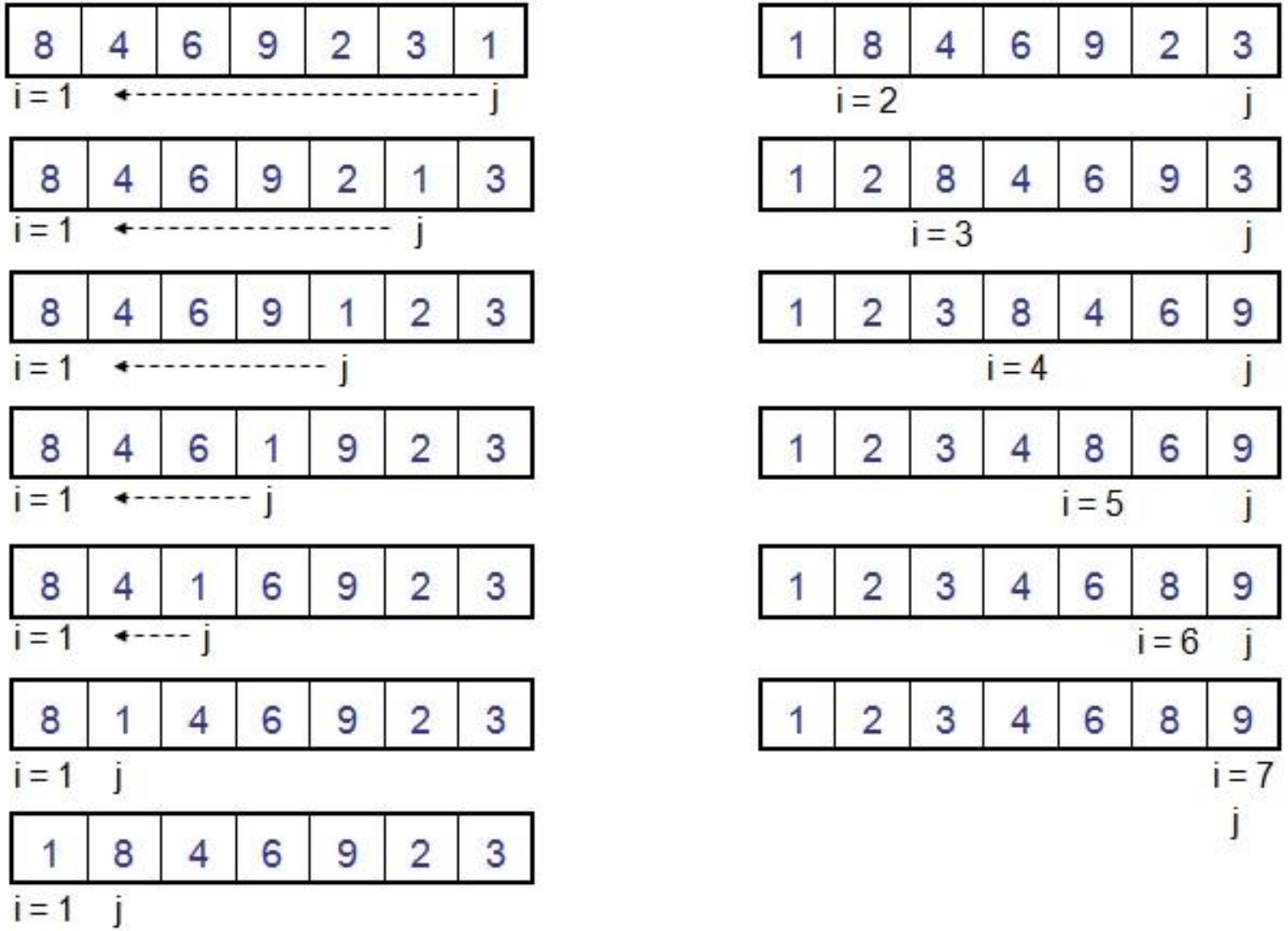
How Selection sort work?

```
arr[] = 64 25 12 22 11

// Find the minimum element in arr[0...4]
// and place it at beginning
11 25 12 22 64

// Find the minimum element in arr[1...4]
// and place it at beginning of arr[1...4]
11 12 25 22 64

// Find the minimum element in arr[2...4]
// and place it at beginning of arr[2...4]
11 12 22 25 64

// Find the minimum element in arr[3...4]
// and place it at beginning of arr[3...4]
11 12 22 25 64
```

How Selection sort work?



How Selection sort work?



Selection Sort Algorithm

- We assume list is an array of n elements. We further assume that swap function swaps the values of the given array elements.

```
1:  function SELECTION-SORT(A, n)
2:      for i = 1 to n-1 do
3:          min ← i
4:          for j = i + 1 to n do
5:              if A[j] < A[min] then
6:                  min ← j
7:              end if
8:          end for
9:          swap A[i], A[min]
10:     end for
11: end function
```

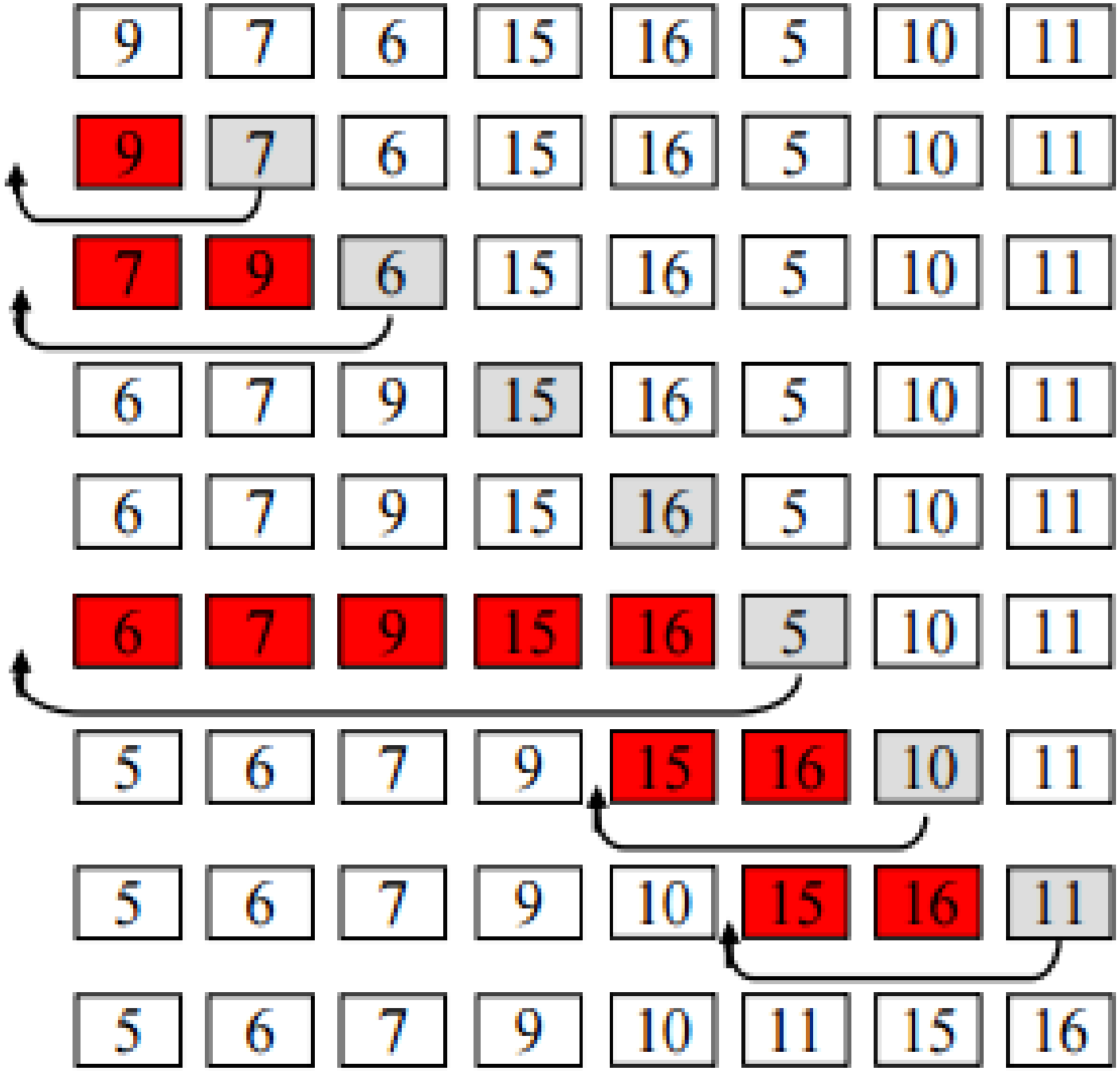
Insertion Sort

- This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted.
- For example, the lower part of an array is maintained to be sorted. An element which is to be inserted in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, insertion sort.
- The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array).

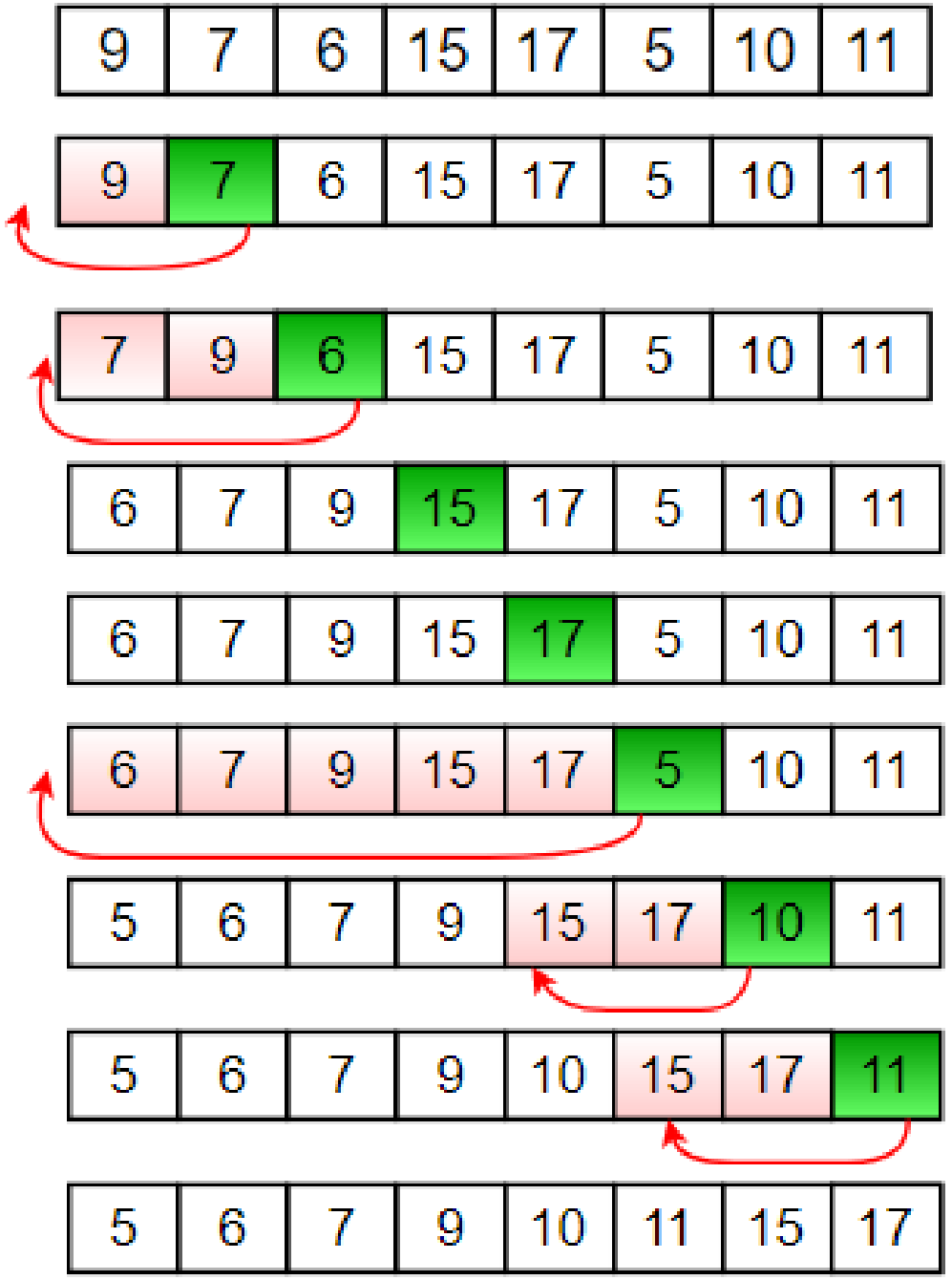
Insertion Sort

- This algorithm is not suitable for large data sets as its average and worst case complexity are of O(n^2), where n is the number of items.
- Time Complexity: O(n^2)
- Boundary Cases: Insertion sort takes maximum time to sort if elements are sorted in reverse order. And it takes minimum time (Order of n) when elements are already sorted.
- Sorting In Place: Yes
- Stable: Yes

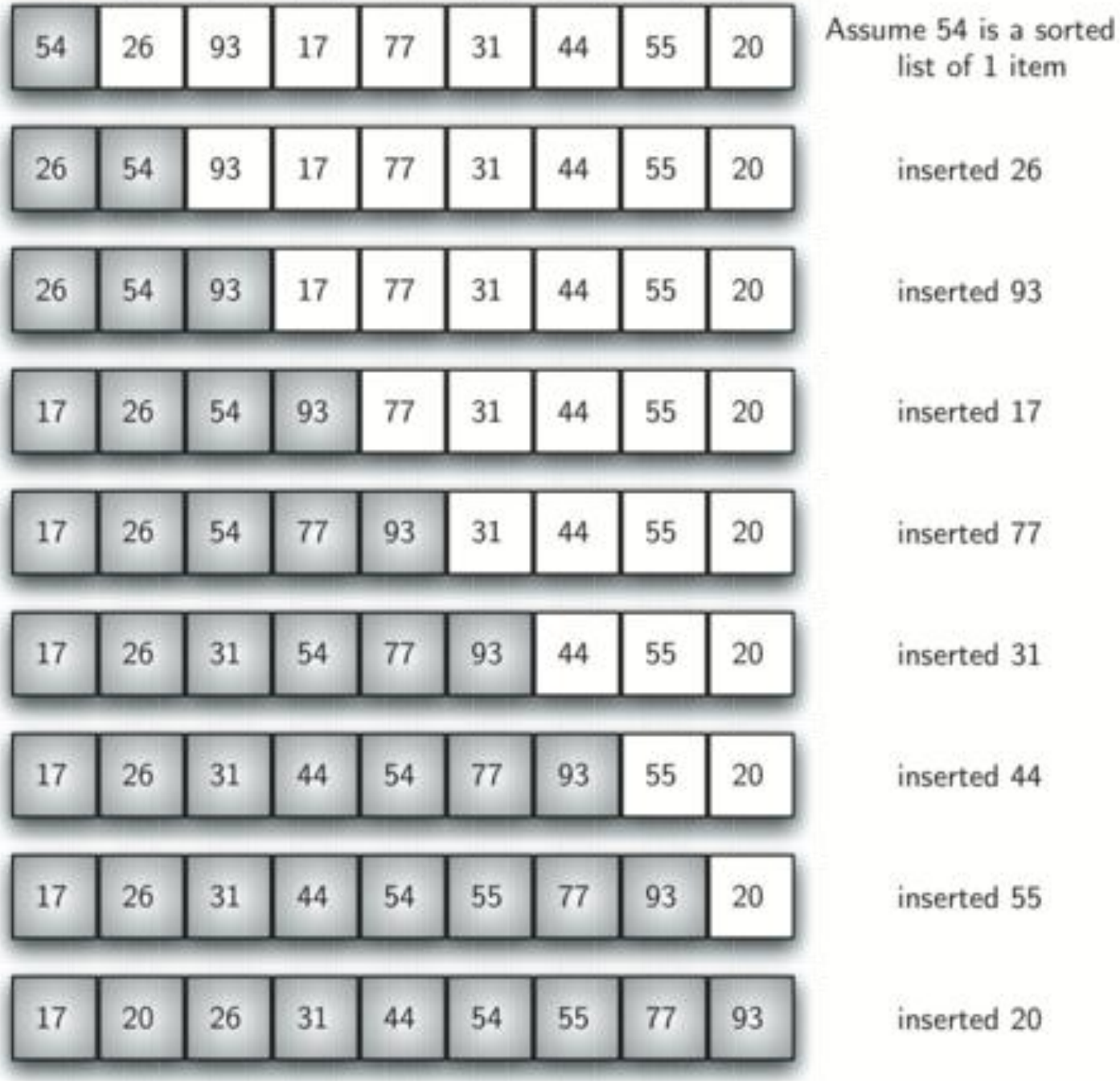
How Insertion sort work?



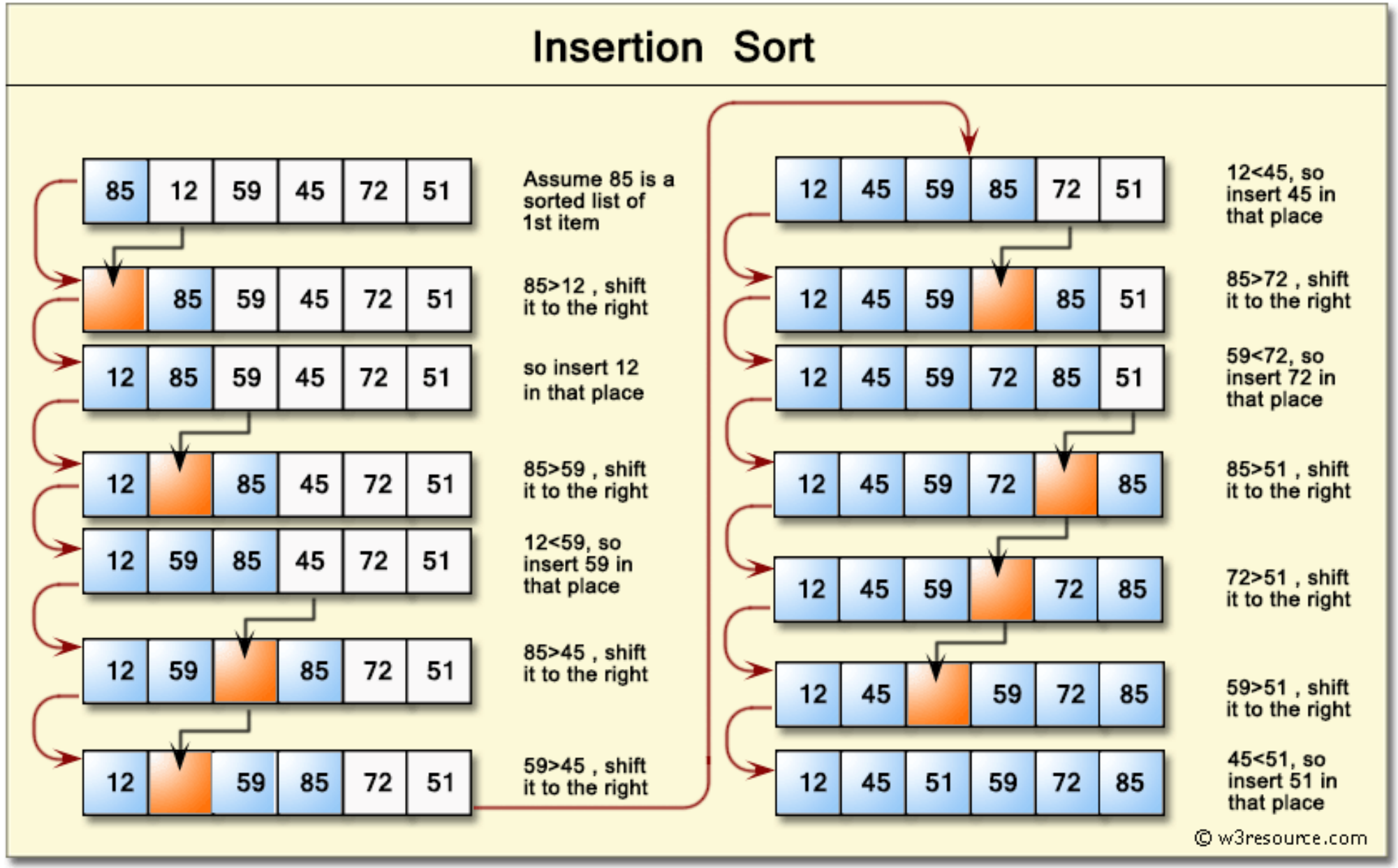
How Insertion sort work?



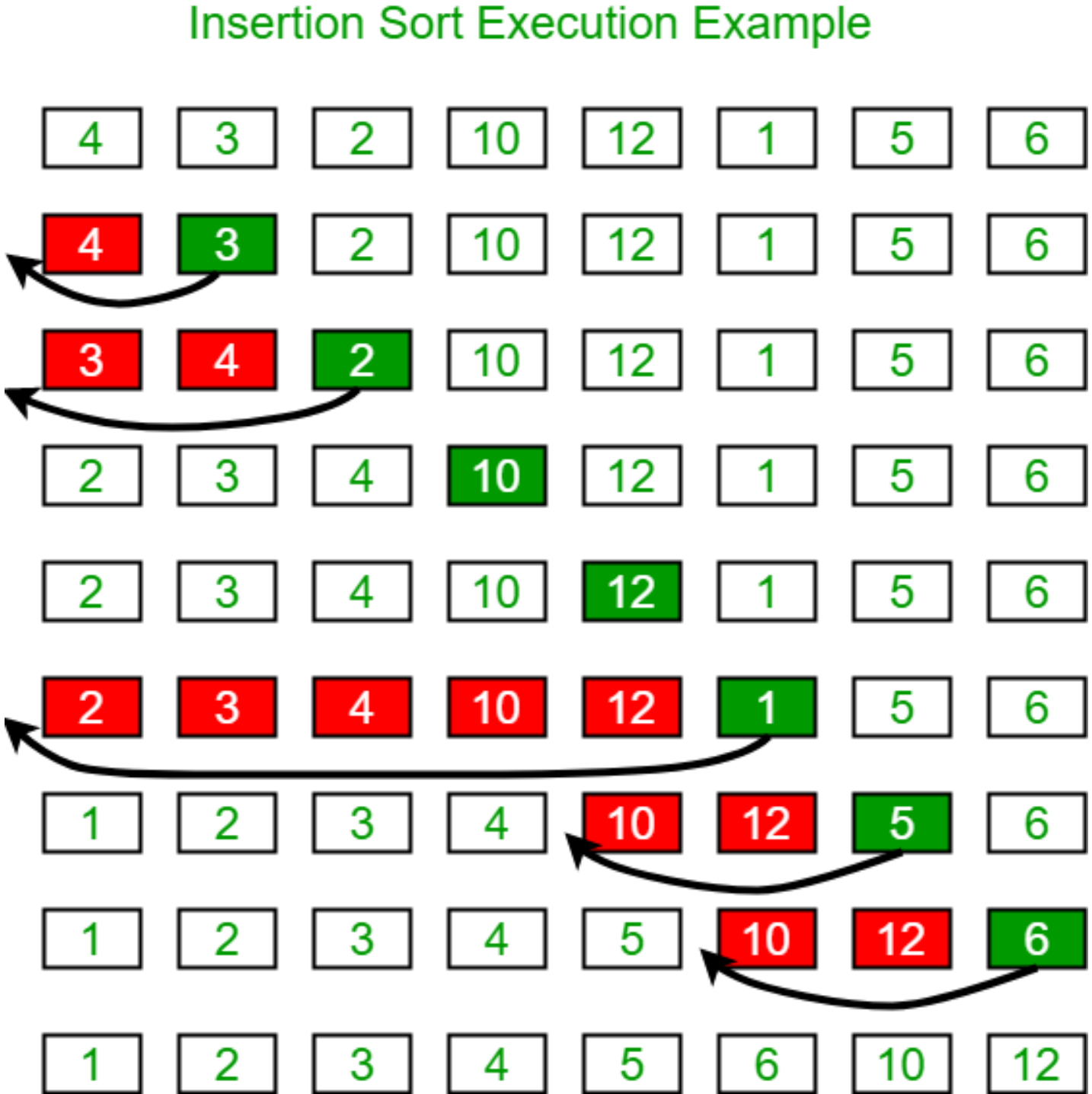
How Insertion sort work?



How Insertion sort work?



How Insertion sort work?



Insertion Sort Algorithm

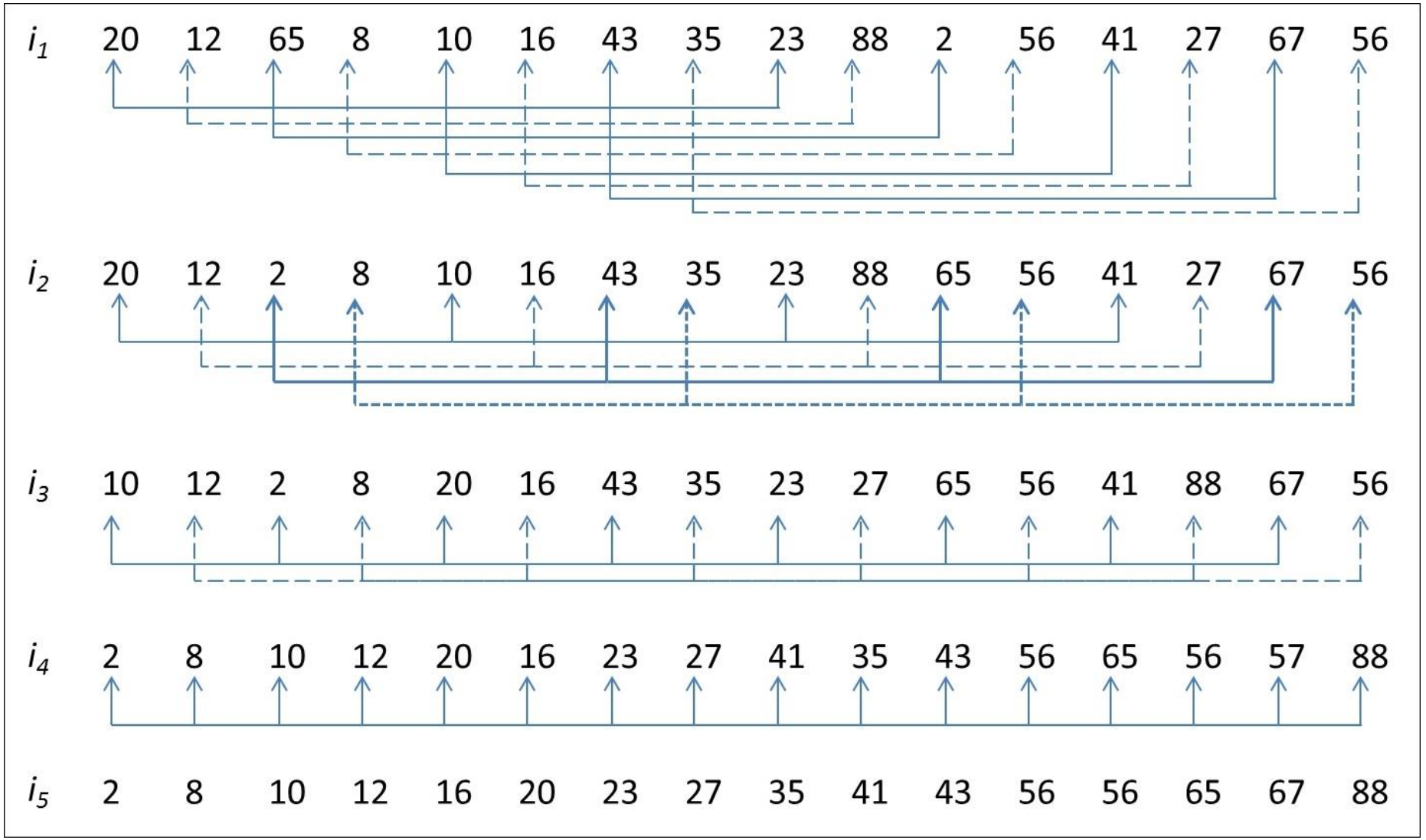
- Now we have a bigger picture of how this sorting technique works, so we can derive simple steps by which we can achieve insertion sort.

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

Shell Sort

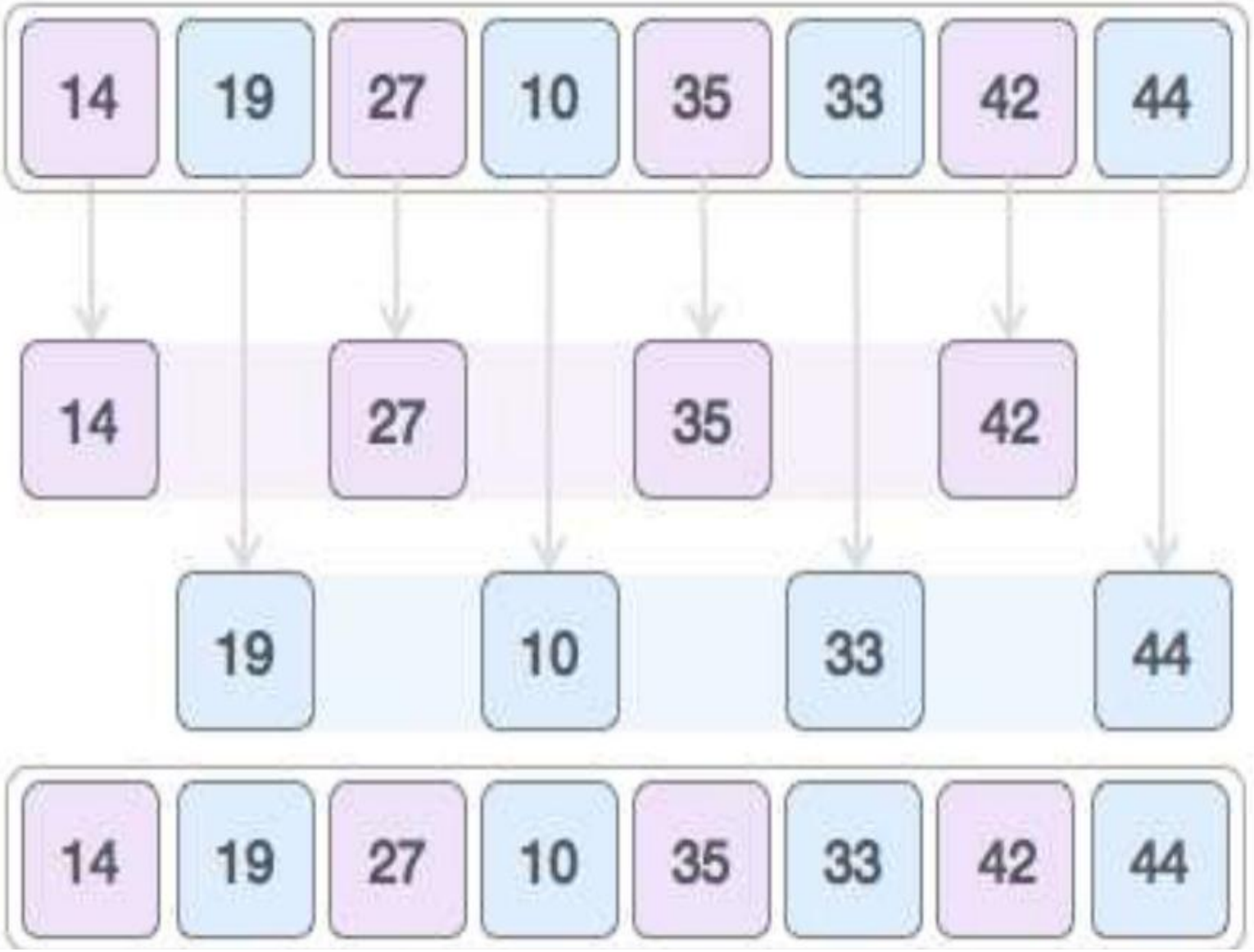
- Shell sort is mainly a variation of Insertion Sort. In insertion sort, we move elements only one position ahead. When an element has to be moved far ahead, many movements are involved. The idea of shell sort is to allow exchange of far items.
- In shell sort, we make the array h-sorted for a large value of h. We keep reducing the value of h until it becomes 1. An array is said to be h-sorted if all sublists of every h-th element is sorted.
- This algorithm avoids large shifts as in case of insertion sort, if the smaller value is to the far right and has to be moved to the far left.
- This algorithm uses insertion sort on a widely spread elements, first to sort them and then sorts the less widely spaced elements. This spacing is termed as interval.
- Time Complexity: between $O(n * log^2(n))$ and $O(n^{1.5})$

How shell sort work?

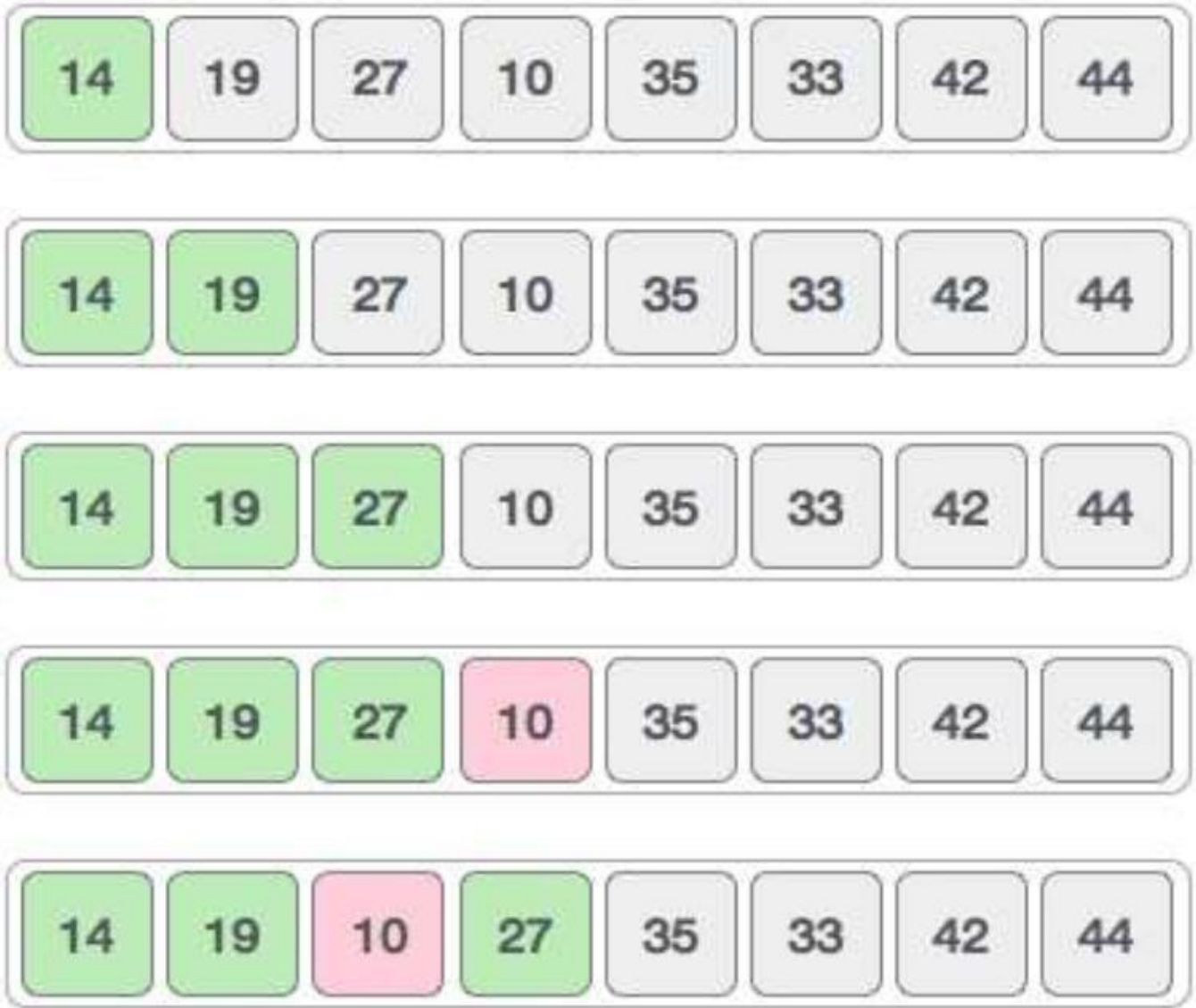


How shell sort work?

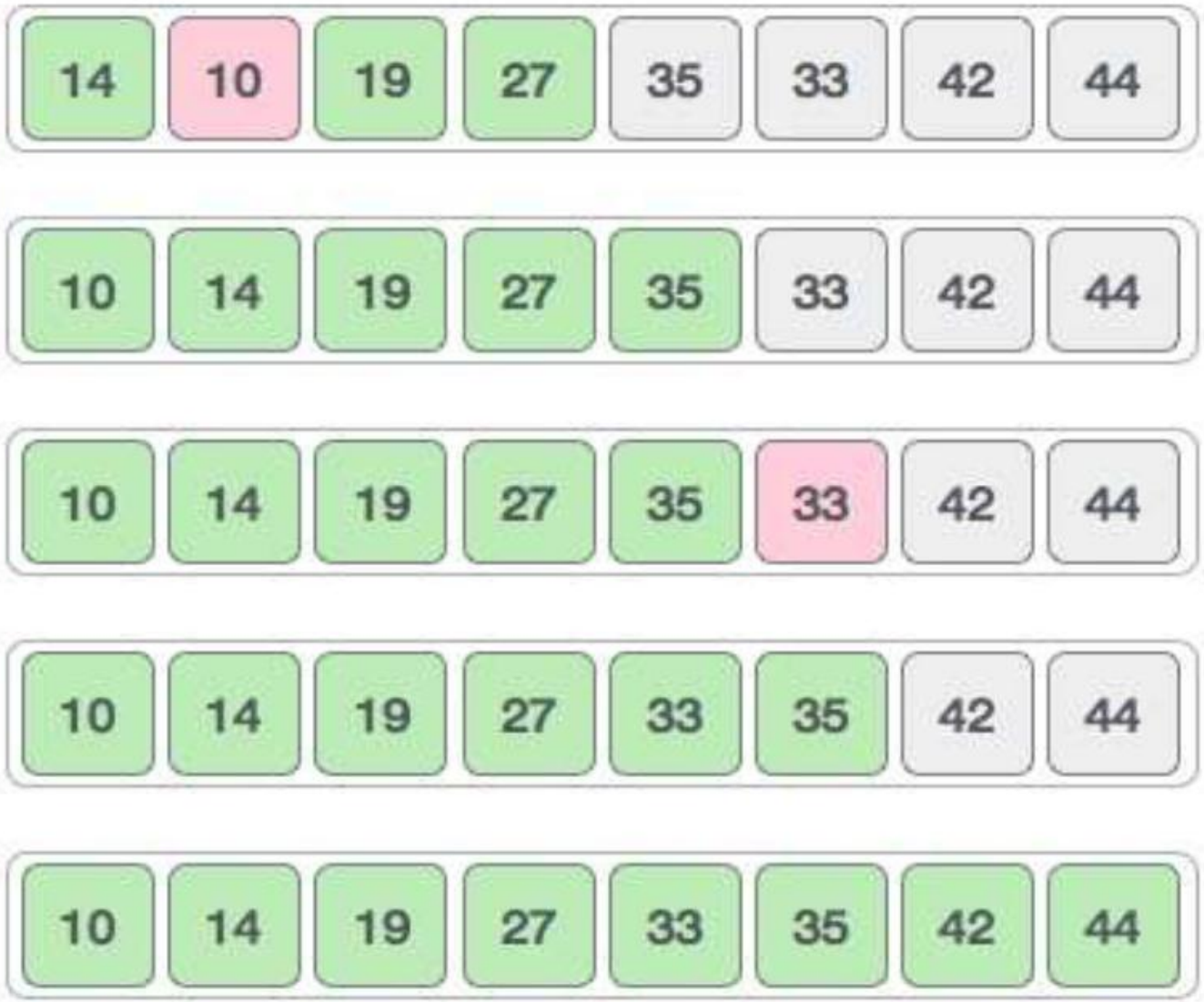
How shell sort work?



How shell sort work?



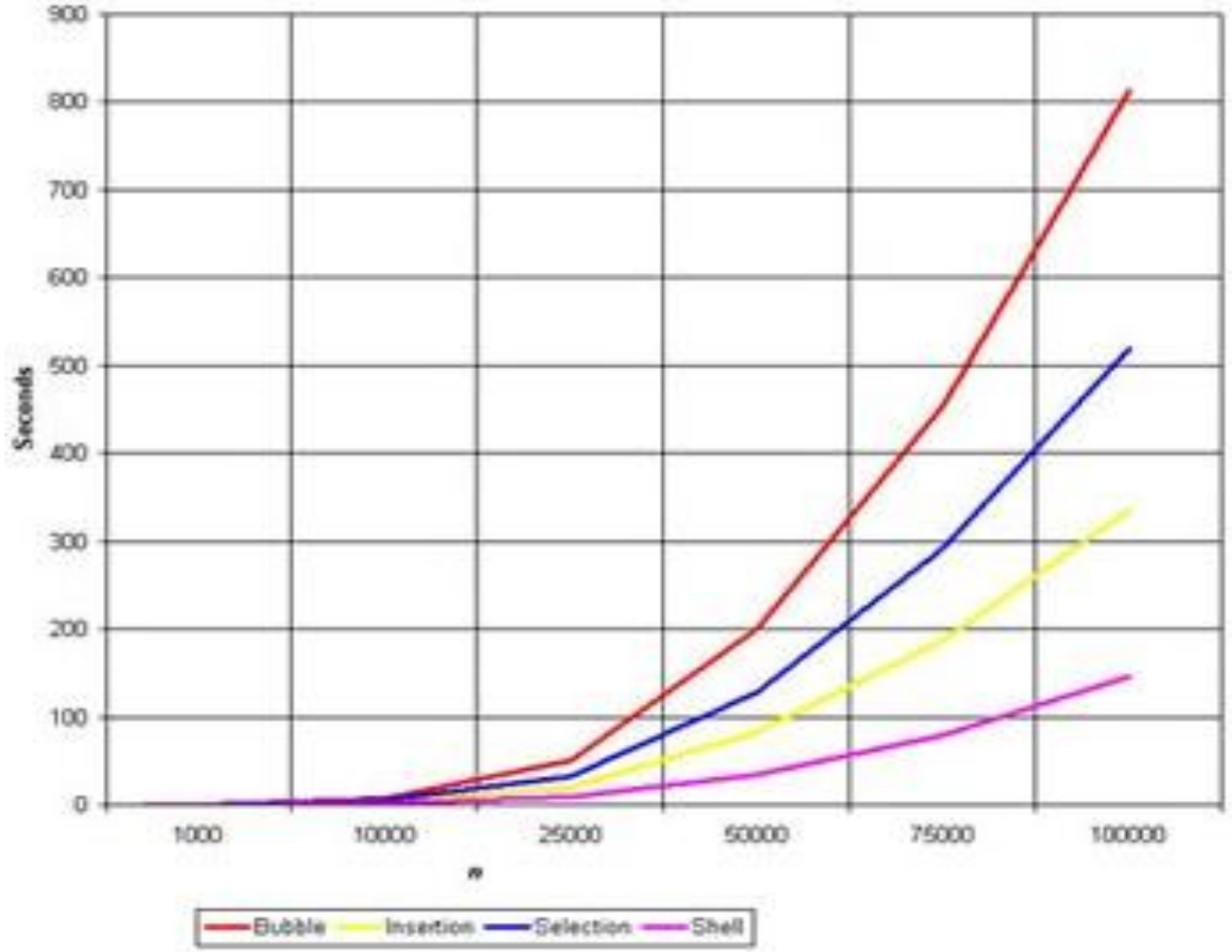
How shell sort work?



Shell Sort Algorithm

```
1.  h ← n
2.  repeat
3.      h ← h/2
4.      for i = h to n do {
5.          key ← A[i]
6.          j ← i
7.          while key < A[j - h] {
8.              A[j] ← A[j - h]
9.              j ← j - h
10.             if j < h then break
11.         }
12.         A[j] ← key
13.     }
14. until h ≤ 1
```

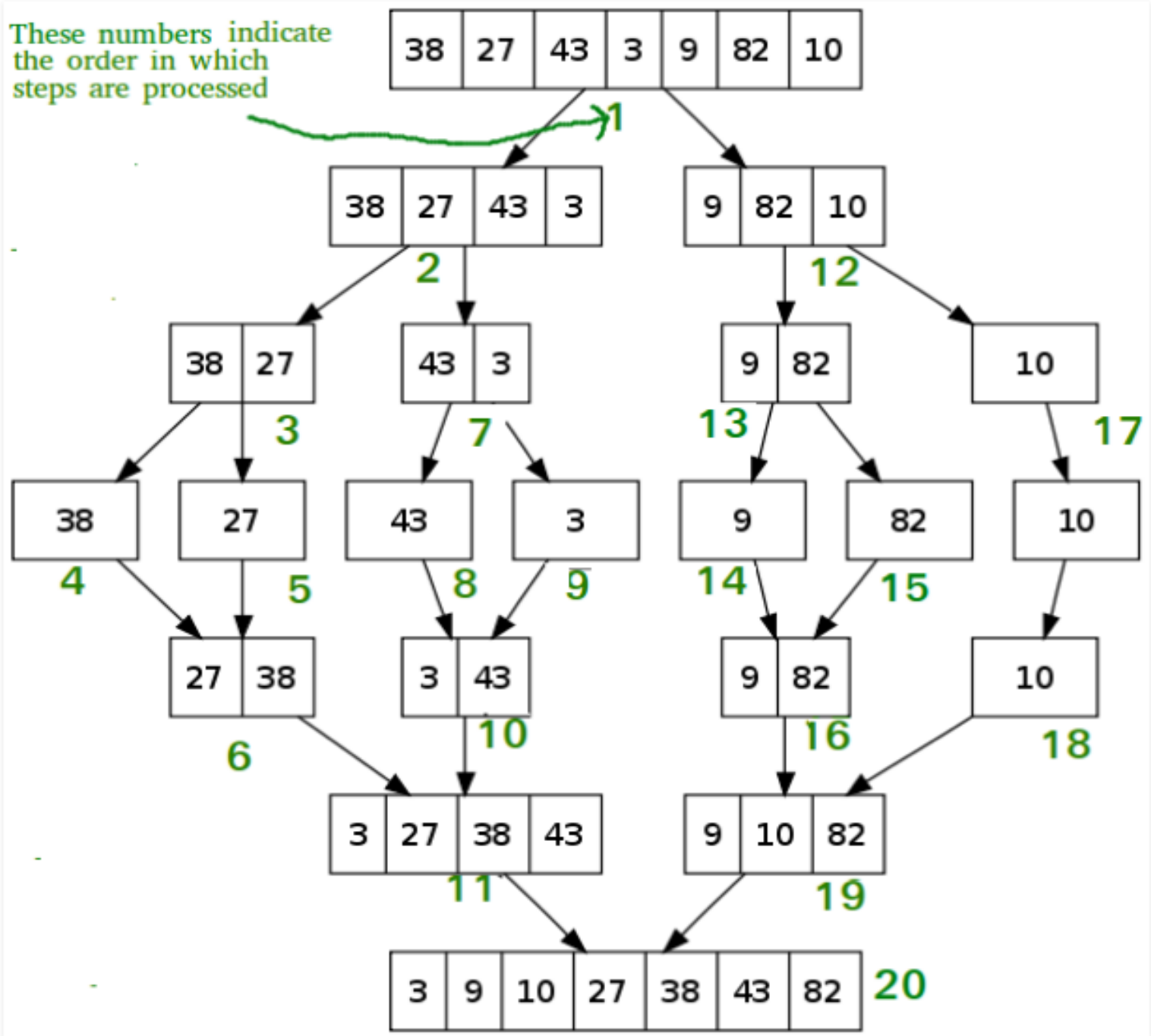
Time Complexity for bubble - selection - insertion - shell



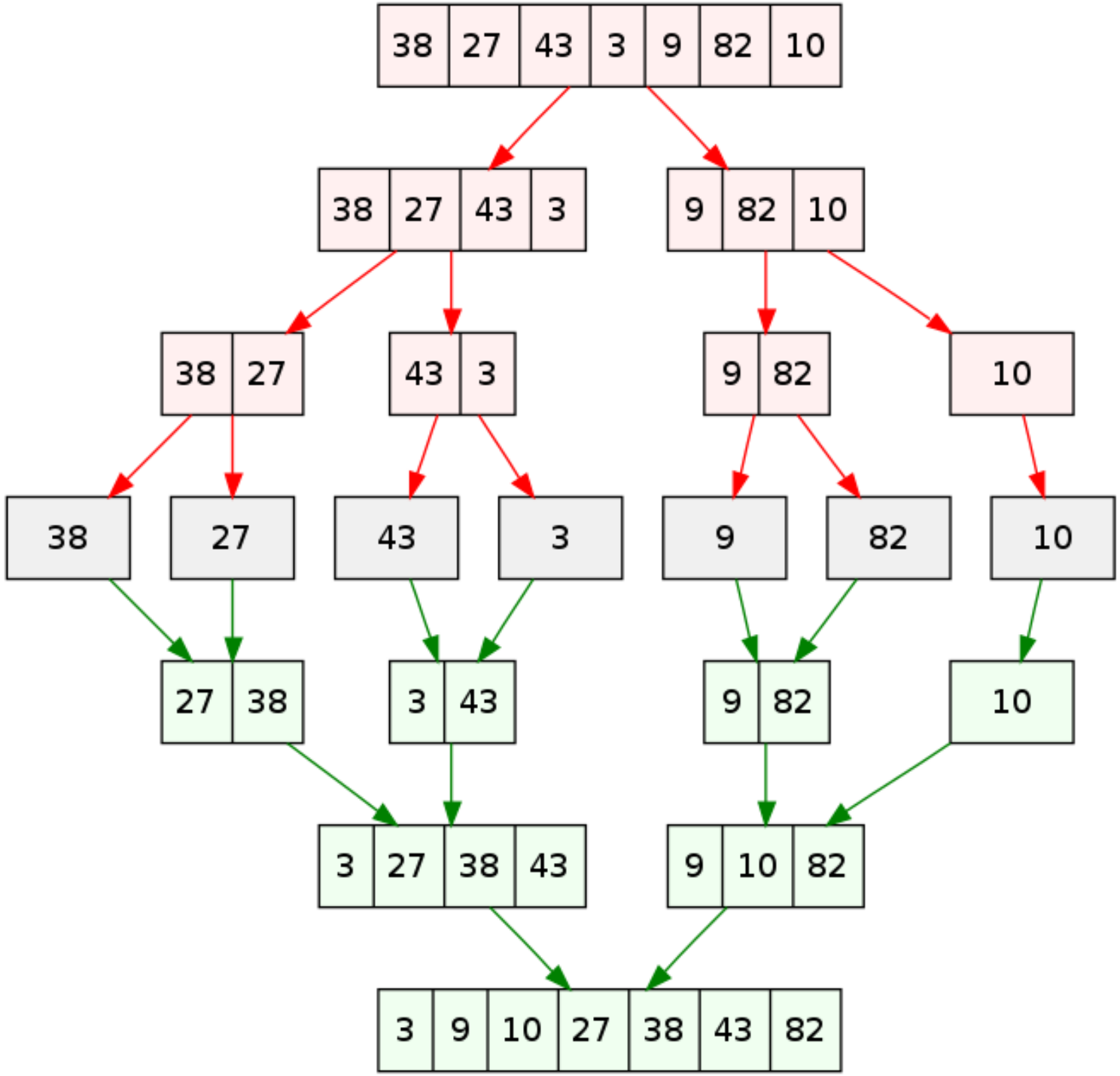
Merge Sort

- Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. The merge() function is used for merging two halves. The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one.
- Time Complexity: O($n * log(n)$)
- Sorting In Place: No in a typical implementation
- Stable: Yes

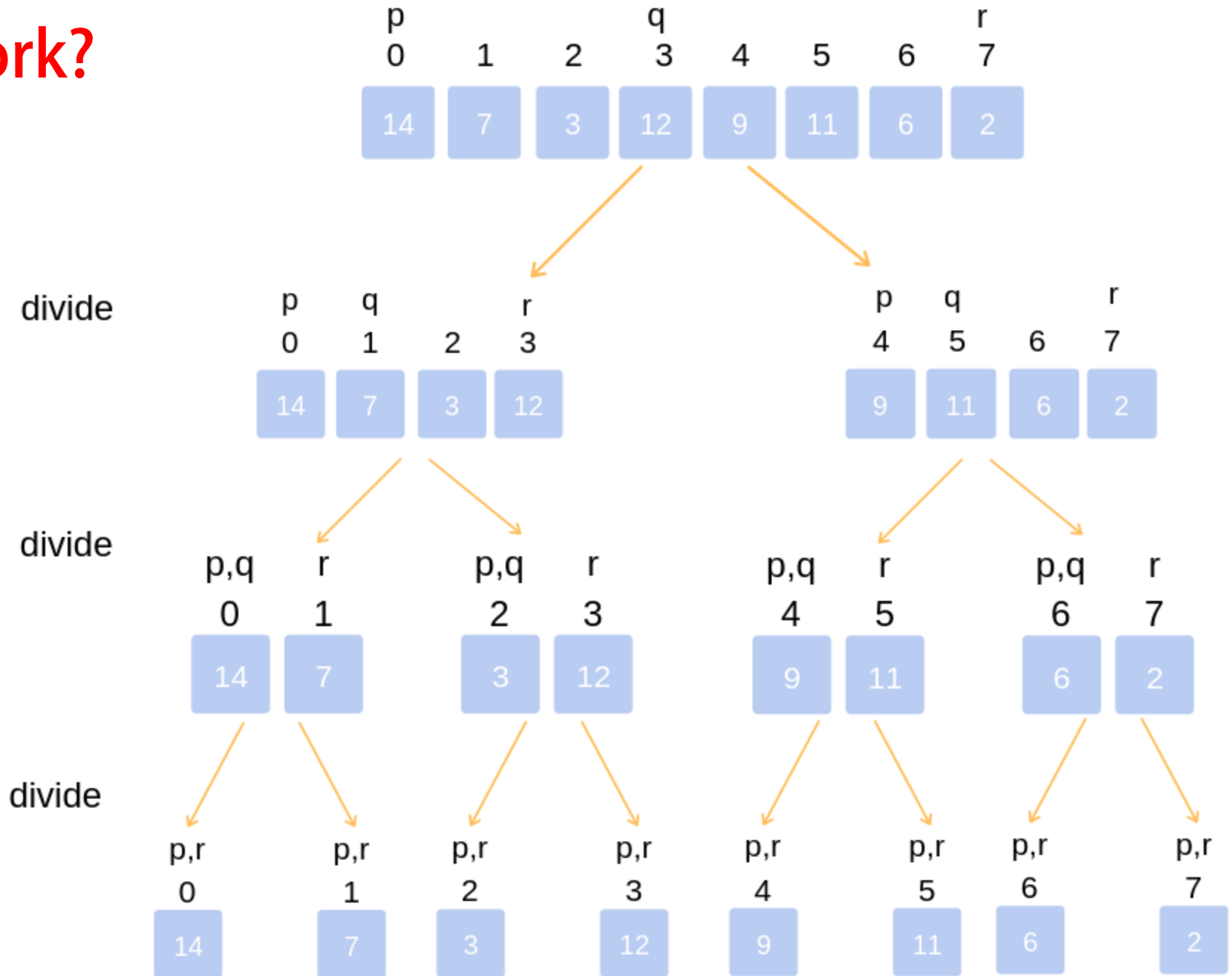
How merge sort work?



How merge sort work?



How merge sort work?



Merge Sort Algorithm

```
MergeSort(arr[], l,  r)
If r > l
     1. Find the middle point to divide the array into two halves:
             middle m = (l+r)/2
     2. Call mergeSort for first half:
             Call mergeSort(arr, l, m)
     3. Call mergeSort for second half:
             Call mergeSort(arr, m+1, r)
     4. Merge the two halves sorted in step 2 and 3:
             Call merge(arr, l, m, r)
```

Merge Sort Algorithm

```
MERGE(A, p, q, r)
 1  n_1 = q − p + 1
 2  n_2 = r − q
 3  let L[1 .. n_1 + 1] and R[1 .. n_2 + 1] be new arrays
 4  for i = 1 to n_1
 5      L[i] = A[p + i − 1]
 6  for j = 1 to n_2
 7      R[j] = A[q + j]
 8  L[n_1 + 1] = ∞
 9  R[n_2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
```

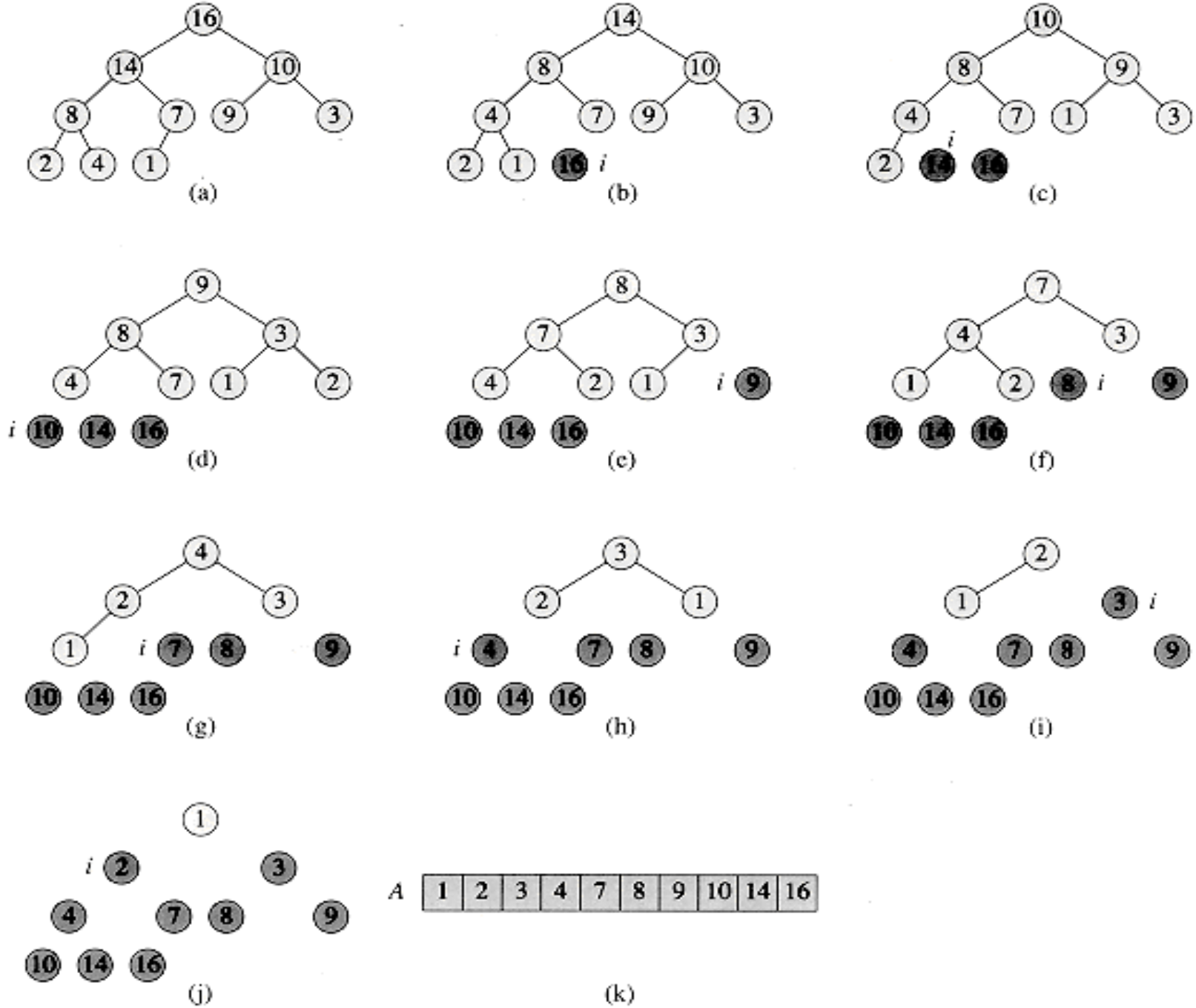
Heap Sort

- Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for remaining element.
- What is Binary Heap?
  - Let us first define a Complete Binary Tree. A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.
  - A Binary Heap is a Complete Binary Tree where items are stored in a special order such that value in a parent node is greater(or smaller) than the values in its two children nodes. The former is called as max heap and the latter is called min heap. The heap can be represented by binary tree or array.

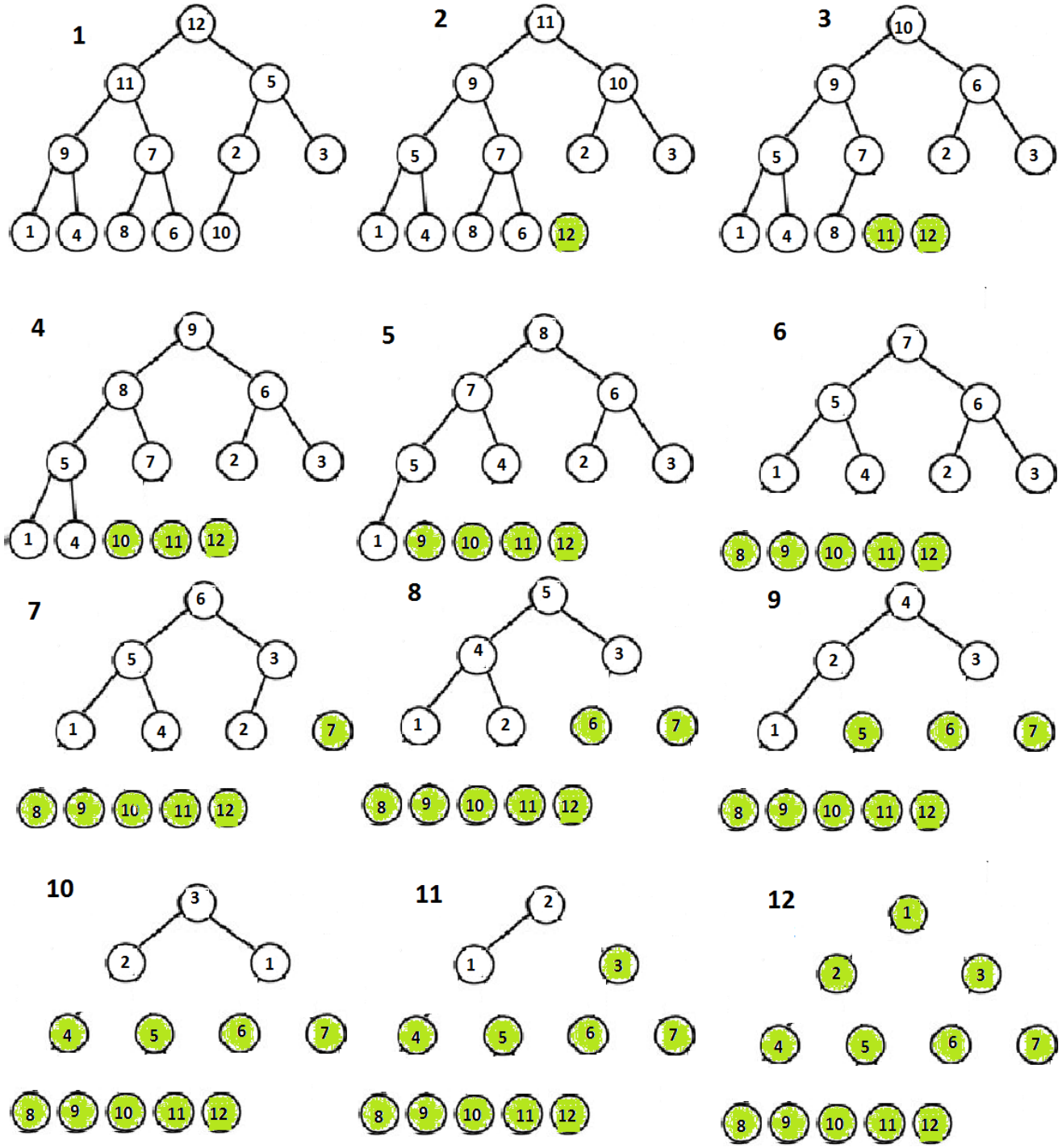
Heap Sort

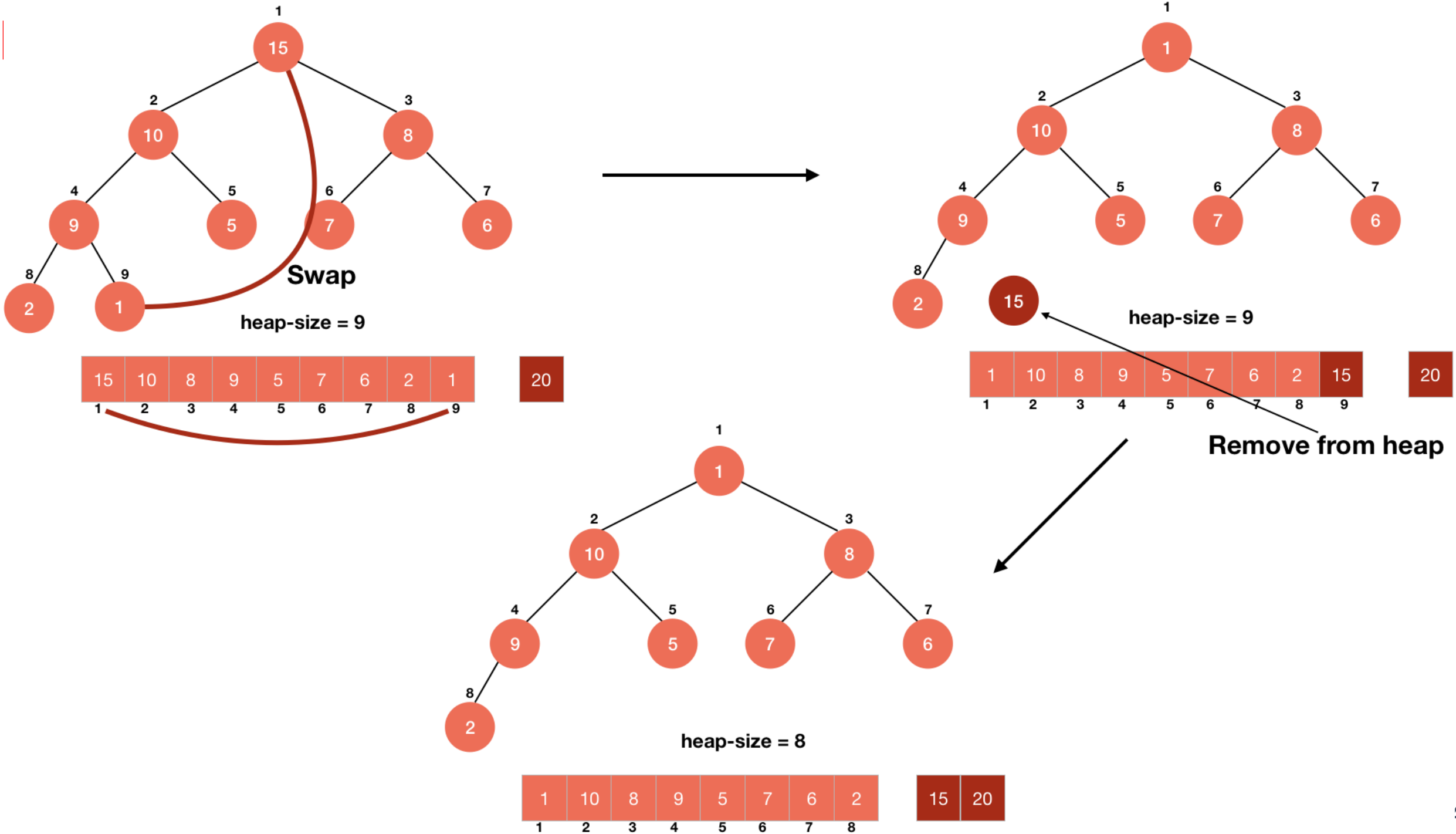
- Why array based representation for Binary Heap?
- Since a Binary Heap is a Complete Binary Tree, it can be easily represented as array and array based representation is space efficient. If the parent node is stored at index i, the left child can be calculated by 2 * i + 1 and right child by 2 * i + 2 (assuming the indexing starts at 0).
- Heap Sort Algorithm for sorting in increasing order:
    1. Build a max heap from the input data.
    2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.
    3. Repeat above steps while size of heap is greater than 1.

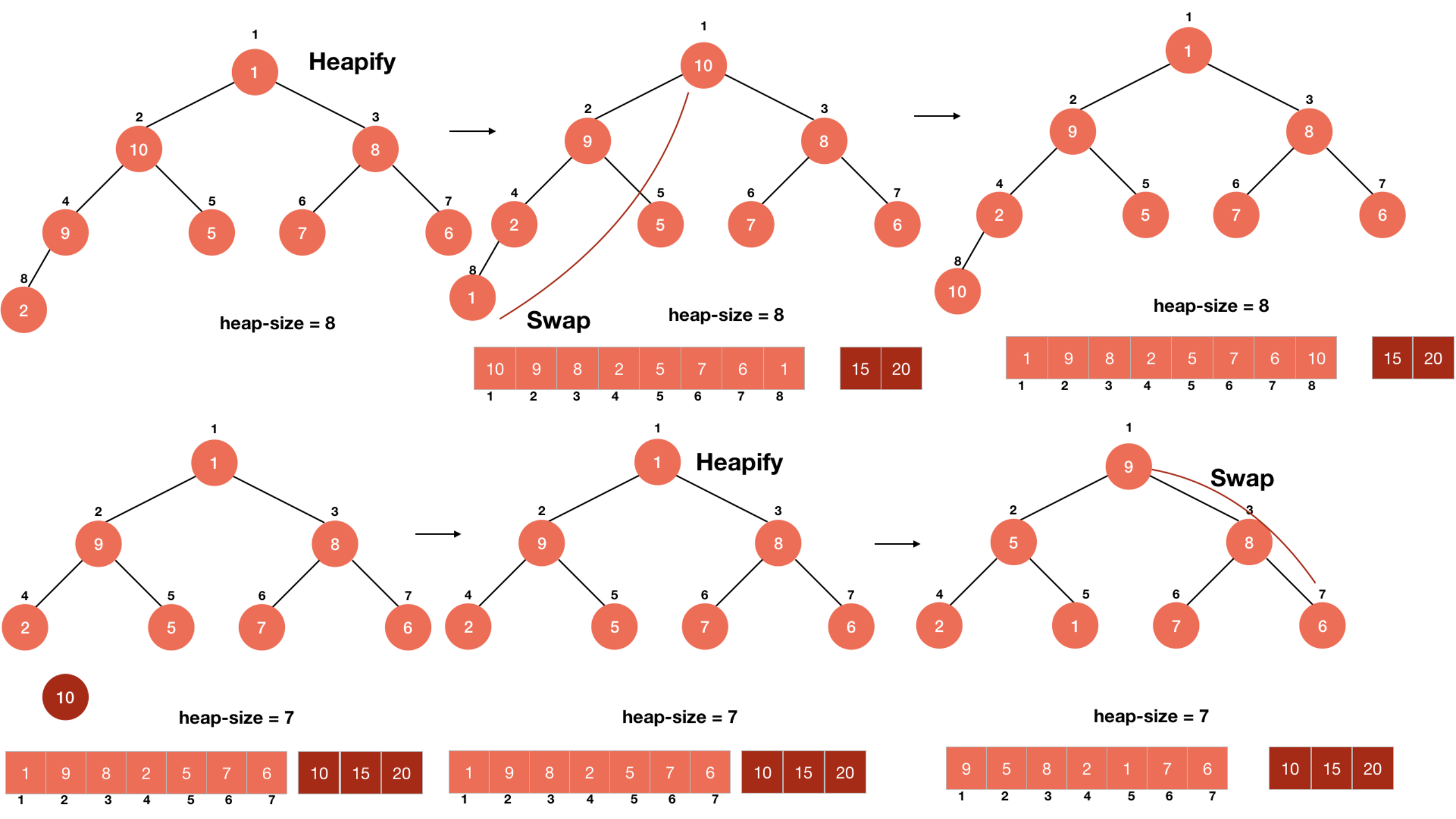
How heap sort work?



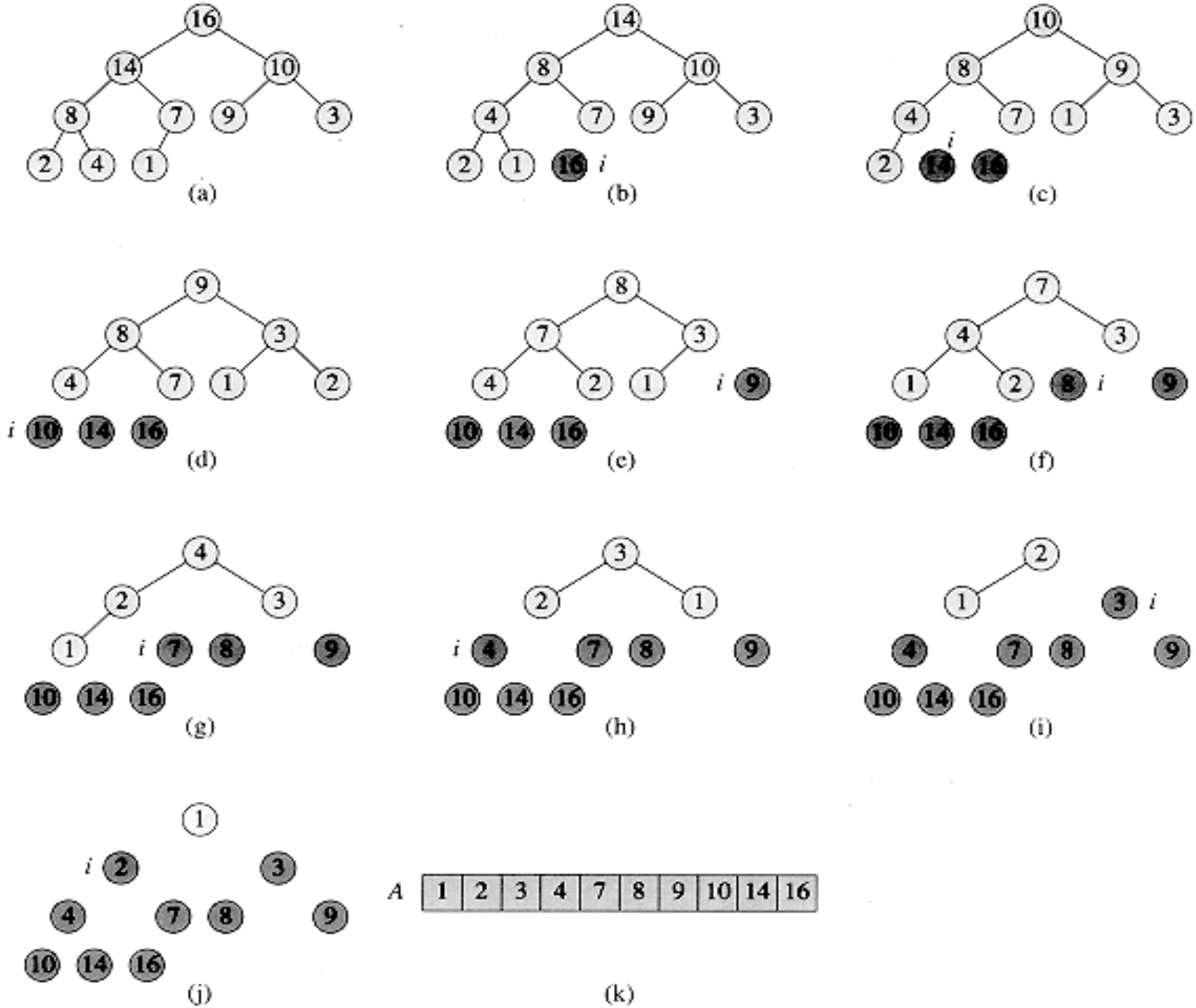
How heap sort work?




Swap
heap-size = 9
15 10 8 9 5 7 6 2 1
20
heap-size = 9
1 10 8 9 5 7 6 2 15
20
Remove from heap
heap-size = 8
1 10 8 9 5 7 6 2
15 20


Heapify
heap-size = 8
Swap
heap-size = 8
heap-size = 8
10 9 8 2 5 7 6 1
15 20
1 9 8 2 5 7 6 10
15 20
Heapify
Swap
10
heap-size = 7
heap-size = 7
heap-size = 7
1 9 8 2 5 7 6
10 15 20
1 9 8 2 5 7 6
10 15 20
9 5 8 2 1 7 6
10 15 20

How heap sort work?



Heap Sort Algorithm

```
MAX-HEAPIFY(A, i)
 1  l = LEFT(i)
 2  r = RIGHT(i)
 3  if l ≤ A.heap-size and A[l] > A[i]
 4       largest = l
 5  else largest = i
 6  if r ≤ A.heap-size and A[r] > A[largest]
 7       largest = r
 8  if largest ≠ i
 9       exchange A[i] with A[largest]
10       MAX-HEAPIFY(A, largest)
```

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3       MAX-HEAPIFY(A, i)
```

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3       exchange A[1] with A[i]
4       A.heap-size = A.heap-size − 1
5       MAX-HEAPIFY(A, 1)
```

Count Sort

- Counting sort is a sorting technique based on keys between a specific range. It works by counting the number of objects having distinct key values (kind of hashing). Then doing some arithmetic to calculate the position of each object in the output sequence.
- Time Complexity: O(n+k) where n is the number of elements in input array and k is the range of input.

Count Sort

- Points to be noted:

  1. Counting sort is efficient if the range of input data is not significantly greater than the number of objects to be sorted. Consider the situation where the input sequence is between range 1 to 10K and the data is 10, 5, 10K, 5K.

  2. It is not a comparison based sorting. It running time complexity is O(n) with space proportional to the range of data.

  3. It is often used as a sub-routine to another sorting algorithm like radix sort.

  4. Counting sort uses a partial hashing to count the occurrence of the data object in O(1).

  5. Counting sort can be extended to work for negative inputs also.

How count sort work?

```
For simplicity, consider the data in the range 0 to 9.
Input data: 1, 4, 1, 2, 7, 5, 2
  1) Take a count array to store the count of each unique object.
  Index:     0  1  2  3  4  5  6  7  8  9
  Count:     0  2  2  0   1  1  0  1  0  0

  2) Modify the count array such that each element at each index
  stores the sum of previous counts.
  Index:     0  1  2  3  4  5  6  7  8  9
  Count:     0  2  4  4  5  6  6  7  7  7

The modified count array indicates the position of each object in
the output sequence.

  3) Output each object from the input sequence followed by
  decreasing its count by 1.
  Process the input data: 1, 4, 1, 2, 7, 5, 2. Position of 1 is 2.
  Put data 1 at index 2 in output. Decrease count by 1 to place
  next data 1 at an index 1 smaller than this index.
```

Count Sort Algorithm

```
1  CountingSort(A, B, k)
2          for i=0 to k
3                  C[i]= 0;
4          for j=1 to n
5                  C[A[j]]= C[A[j]] + 1;
6          for i=2 to k
7                  C[i] = C[i] + C[i-1];
8          for j=n downto 1
9                  B[C[A[j]]] = A[j];
10                 C[A[j]] = C[A[j]] - 1;
```

Thank You