Unit 8 Sorting

The efficiency of data handling can be increased if the data are sorted according to some criteria of order. It is often necessary to sort data before processing. We choose some criteria that is used to order data. The choice will vary from application to application and must be defined by the user. Very often, the sorting criteria are natural, as in case of numbers. A set of numbers can be sorted in ascending or descending order.

The final ordering of data can be obtained in a variety of ways, and only some of them can be considered meaningful and efficient. To decide which method is best, certain criteria of efficiency have to be established and a method for quantitatively comparing different algorithms must be chosen

To make the comparison machine independent, certain critical properties of sorting algorithms should be defined when comparing alternative methods. Two such properties are the number of comparisons and the number of data movements. To sort a set of data, the data have to be compared and moved as necessary; the efficiency of these two operations depends on the size of the data set.

Elementary Sorting Algorithms:

1. Insertion Sort
2. Selection Sort
3. Bubble Sort

Efficient Sorting Algorithm:

1. Heap Sort
2. Quick Sort
3. Merge Sort
4. Radix Sort

1. Insertion Sort

```
void Insertion  sort (int [] data)
{
        int i, j, temp;
        for (i=1; i<data.length; i++)
        {
                temp = data[i];
                for (j=i; j>0 && temp<data[j-1]; j--)
                        data[j] = data[j-1];
                data[j] = temp;
        }
}
```

Example: Consider the following array: 25, 17, 31, 13, 2

First Iteration:

25	17	31	13	2

Since 17 < 25. Hence swap 17 and 25.

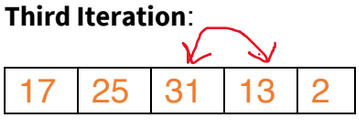
17	25	31	13	2

Second Iteration:

Since 31> 25, no swapping takes place.

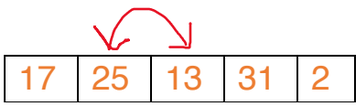
Also, 31> 17, no swapping takes place and 31 remains at its position.

Third Iteration:



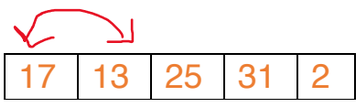
Since 13 < 31, we swap the two.

Array now becomes:



Since, 13 < 25, we swap the two.

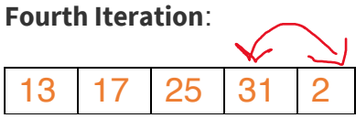
The array becomes:



Since 13 < 17, we swap the two.

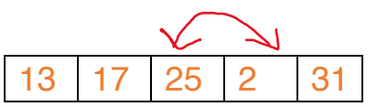
The array now becomes:

Fourth Iteration:

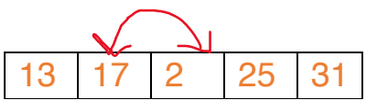


Since 2 < 31. Swap 2 and 31.

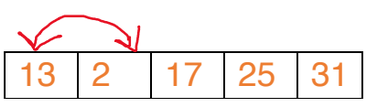
Array now becomes:



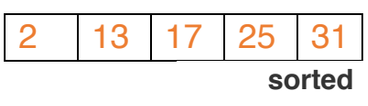
Since 2< 25. Swap 25 and 2. Array now becomes:



Since, 2<17. Swap 2 and 17. Array now becomes:



Since 2< 13. Swap 2 and 13. The array now becomes:



sorted

2. Selection Sort:

The idea of algorithm is quite simple. Array is imaginary divided into two parts- sorted and unsorted one. At the beginning, sorted part is empty, while unsorted one contains whole array. At every step, algorithm finds minimal element in the unsorted part and adds it to the end of the sorted one. When unsorted part becomes empty, algorithm stops.

The pseudocode for Selection sort is:

```
Selection_sort (int data[])
    for i = 0 to data.length-2
     select the smallest element among data[i], ..., data[data.length-1];
     swap it with data[i];
```

Selection Sort Implementation:

```
void selectionsort(int [] data)
{
    int i, j, least;
    for (i=0; i<data.length-1; i++)
    {
        for (j=i+1, least=i; j<data.length; j++)
            if(data[j] < (data[least]))
                least = j;
        if(i != least)
            swap(data, least, i);
    }
}
```

Example: 15, 28, 17, 12, 18, 9, 6

- Find the smallest element (i.e., 6) and swap it with the element at index 0 (i.e., 15)

  6, 28, 17, 12, 18, 9, 15

  - Now, 6 is at correct position. So, start search from index 1 (i.e., 28).

- Find the smallest element (i.e., 9) and swap it with the element at index 1 (i.e., 28)

  6, 9, 17, 12, 18, 28, 15

  - Now, 9 is at correct position. So, start search from index 2 (i.e., 17).

- Find the smallest element (i.e., 12) and swap it with the element at index 2 (i.e., 17)

  6, 9, 12, 17, 18, 28, 15

  - Now, 12 is at correct position. So, start search from index 3 (i.e., 17).

- Find the smallest element (i.e., 15) and swap it with the element at index 3 (i.e., 17)

  6, 9, 12, 15, 18, 28, 17

  - Now, 15 is at correct position. So, start search from index 4 (i.e., 18).

- Find the smallest element (i.e., 17) and swap it with the element at index 4 (i.e., 18)

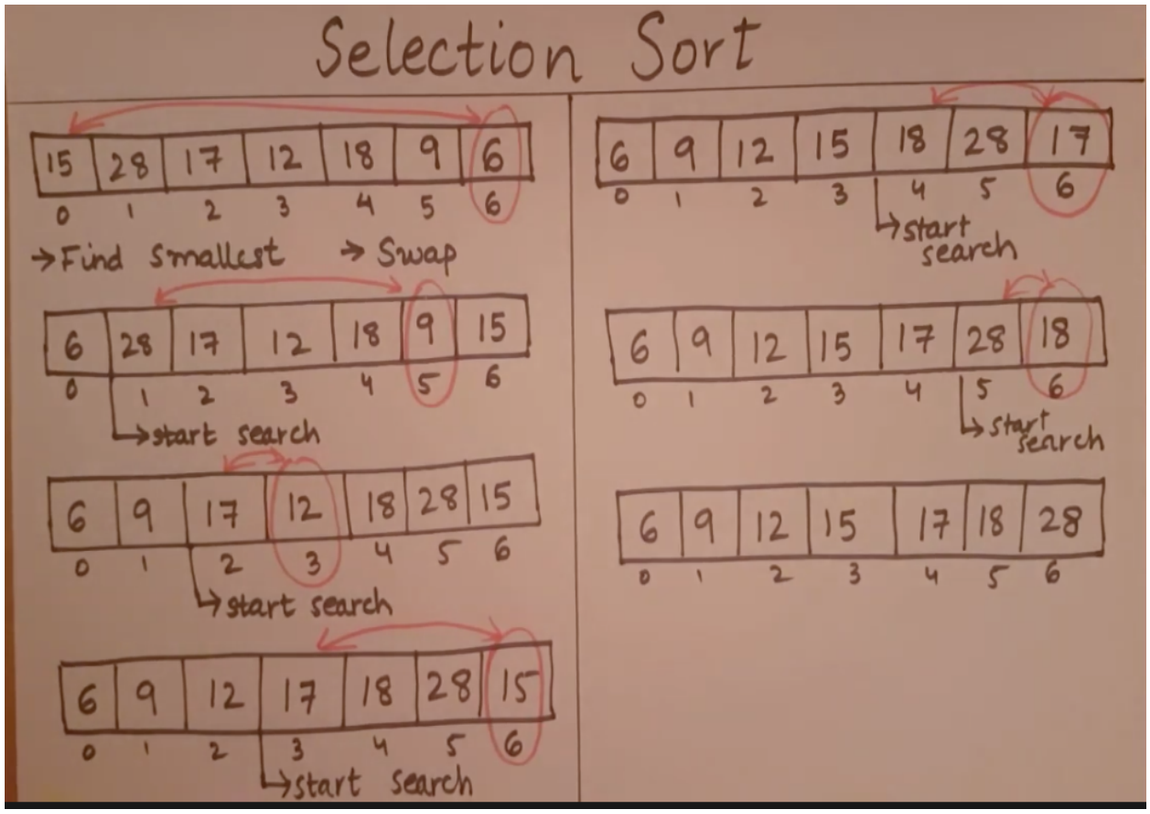
  6, 9, 12, 15, 17, 28, 18

  - Now, 17 is at correct position. So, start search from index 5 (i.e., 28).

- Find the smallest element (i.e. 18) and swap it with the element at index 5 (i.e. 28)

  6, 9, 12, 15, 17, 18, 28

  - Now, 18 is at correct position. Only 28 is left so it is at last index.



2. Bubble Sort:

The bubble sort is the easiest and frequently used sorting algorithm among all the sorting algorithms. The algorithm has got its name as after every pass, the largest element bubbles up and move to end of the array.

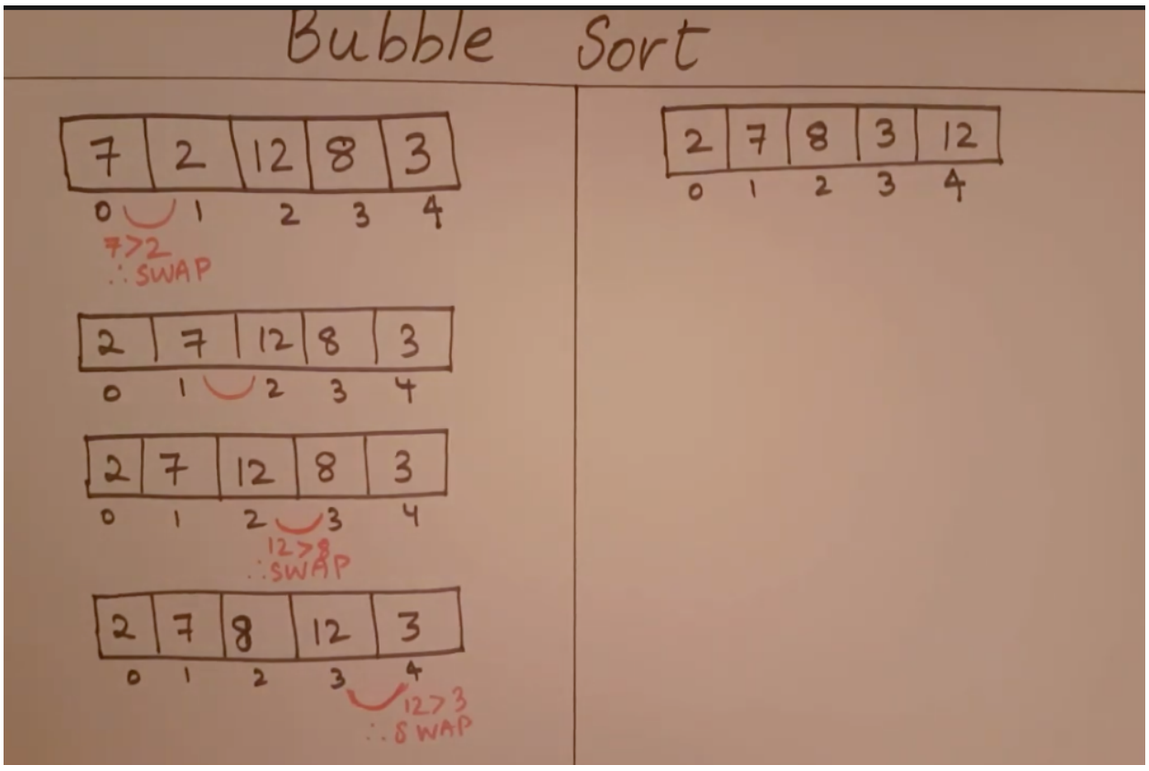
The Java Implementation for bubble sort:

```
void bubblesort(int [] data)
{
        for (int i = 0; i < data.length-1; i++)
                for (int j = data.length-1; j > i; --j)
                        if(data[j]) < (data[j-1]))
                                swap(data, j, j-1);
}
```

Consider the following array: 7, 2, 12, 8, 3

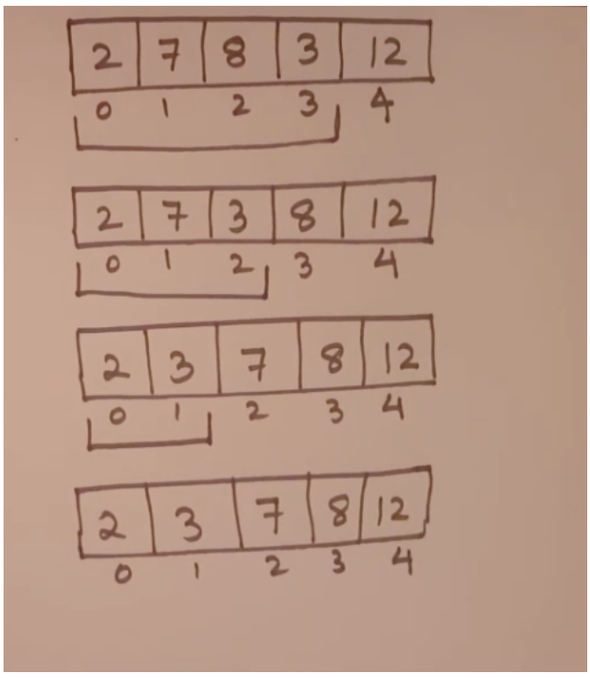
- Try to push the largest element to the last index.
- The basic idea underlying the **bubble sort** is to pass through the file sequentially several times. Each pass consists of comparing each element in the file with its successor (**x[i]** with **x[i+1]**) and interchanging the two elements if they are not in proper order.

First Pass:



- Now, 12 is placed at the last index.
- So, sort element from 0 index to 3 index (2, 7, 8, 3) with the same process as above

Second Pass/ Third Pass/ Fourth Pass:



The complexity of Elementary Sorting algorithm (Insertion, Selection and Bubble Sort) is **$O(n^2)$**

Efficient Sorting Algorithms

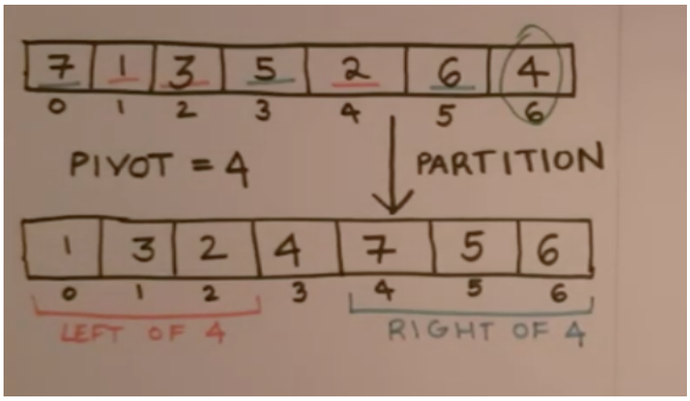
1. Quick Sort:

The quick sort divides the original array into two subarrays, the first of which contains elements less than or equal to a chosen key called pivot or bound. The second subarray includes elements equal to or greater than the bound. The two subarrays can be sorted separately but before this is done, the partition process is repeated for both subarrays. As a result, two new bounds are chosen, one for each subarray. The four subarrays are created because each subarray obtained in first phase is now divided into two segments. This process of partitioning is carried down until there are only one cell arrays that do not need to be sorted at all.

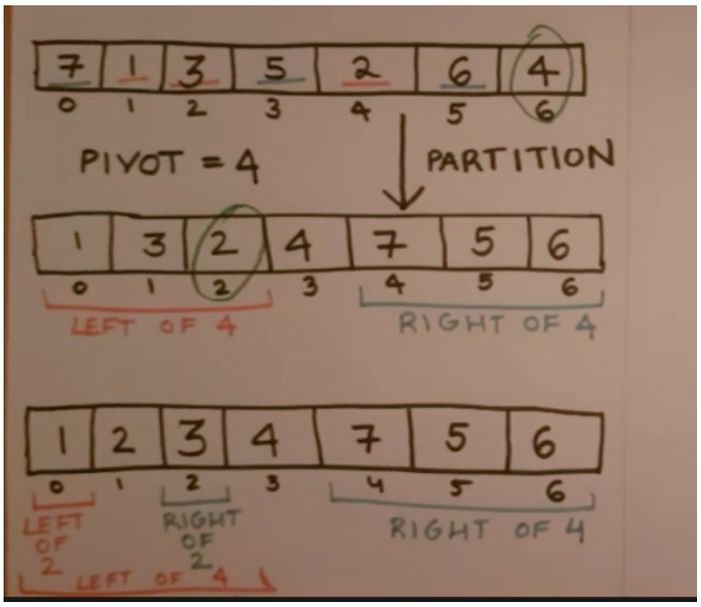
```
void quicksort (a[], lb,ub)
{
        if(lb < ub)
        {
                loc = partition(a, lb, ub);
                quicksort(a, lb, loc-1);
                quicksort(a, loc+1, ub);
        }
}

int partition(a[], lb,ub)
{
        pivot = a[lb];
        start = lb; end = ub;
        while(start <= end)
        {
                while(a[start] <= pivot && start < end)
                {
                        start = start + 1;
                }
                while(a[end] > pivot)
                {
                        end = end - 1;
                }
                if(start < end)
                {
                        swap(a[start], a[end]);
                }
        }
        a[lb] = a[end];
        a[end] = pivot;
        return end;
}
```

Example: Consider the following array: 7, 1, 3, 5, 2, 6, 4

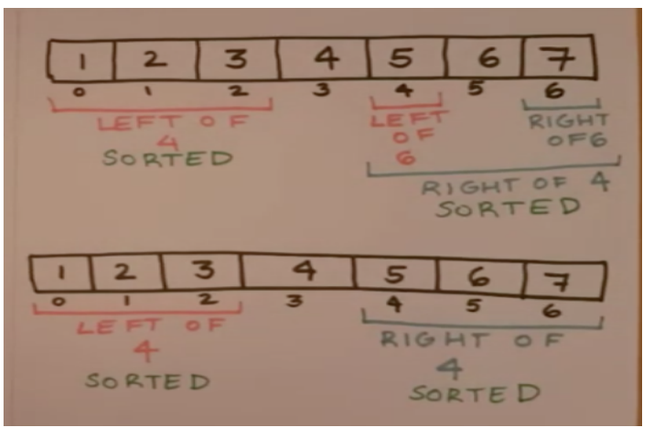
- Choose any element of an array as a pivot element. Say the last element 4 as pivot element. So, Pivot = 4.
- Now, rearrange the array in such a way that all the elements smaller than 4 are placed at the left of 4 and all the elements larger than 4 are placed at the right of 4.



- Apply the partition algorithm again and again until all the element are sorted
- First apply the partition to the left of 4. Pivot = 2 (since 2 is the last element).



- Now apply the partition to the right of 4. Pivot = 6.



The Time complexity of Quick sort is O(nlogn)

2. Heap Sort:

Heap sort was invented by John Williams and uses the approach inherent to selection sort. Selection sort finds among the n elements the one that precedes all other n-1 elements, then the least element among those n-1 items, and so forth, until the array is sorted. To have the array in ascending order, heap sort puts the largest element at the end of the array, then the second largest in front of it, and so on. Heap sort starts from the end of the array by finding the largest elements, whereas selection sort starts from the beginning using the smallest element. The final order in both cases is indeed the same.

A heap is a binary tree with the following two properties.

- The value of each node is not less than the values stored in each of its children.
- The tree is perfectly balanced and the leaves in the last level are all in the leftmost positions.

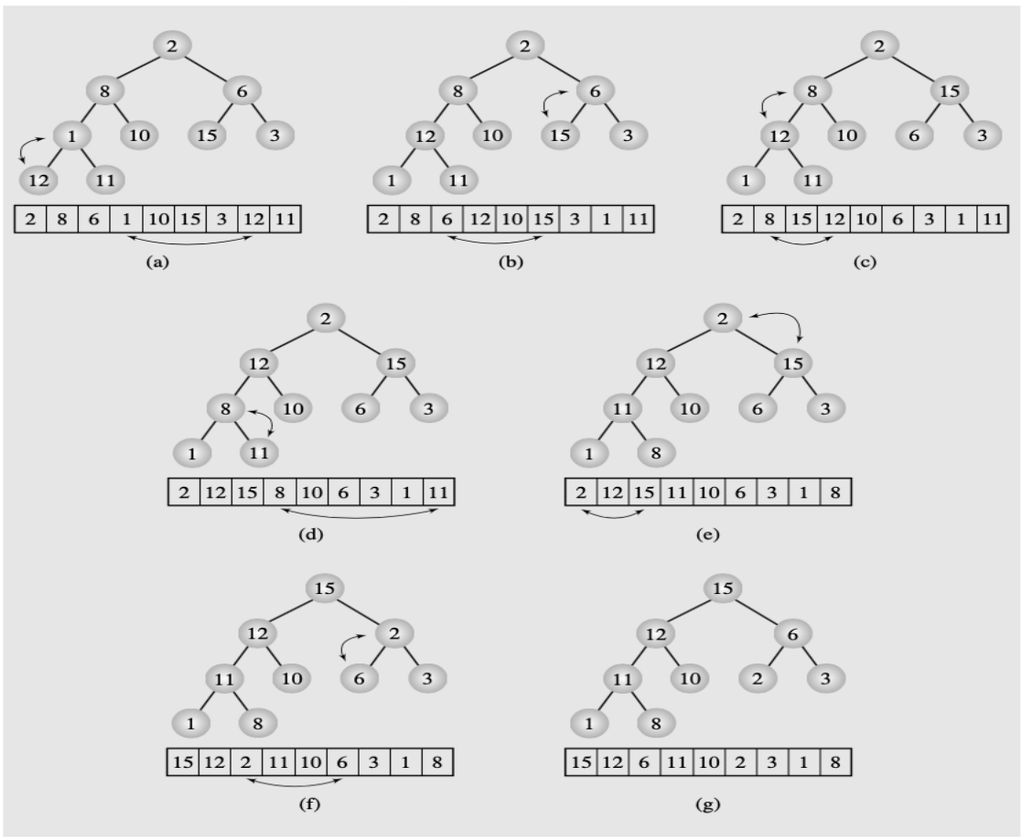
Elements in a heap are not perfectly ordered. It is known only that the largest element is in the root node and that, for each other node, all its descendants are not greater than the element in this node. Heap sort thus starts from the heap, puts the largest element at the end of the array, and restores the heap that now has one less element. From the new heap, the largest element is removed and put in its final position and then the heap property is restored for the remaining elements. Thus, in each round, one element of the array ends up in its final position, and the heap becomes smaller by this one element. The process ends with exhausting all elements from the heap.

Pseudocode for Heap Sort

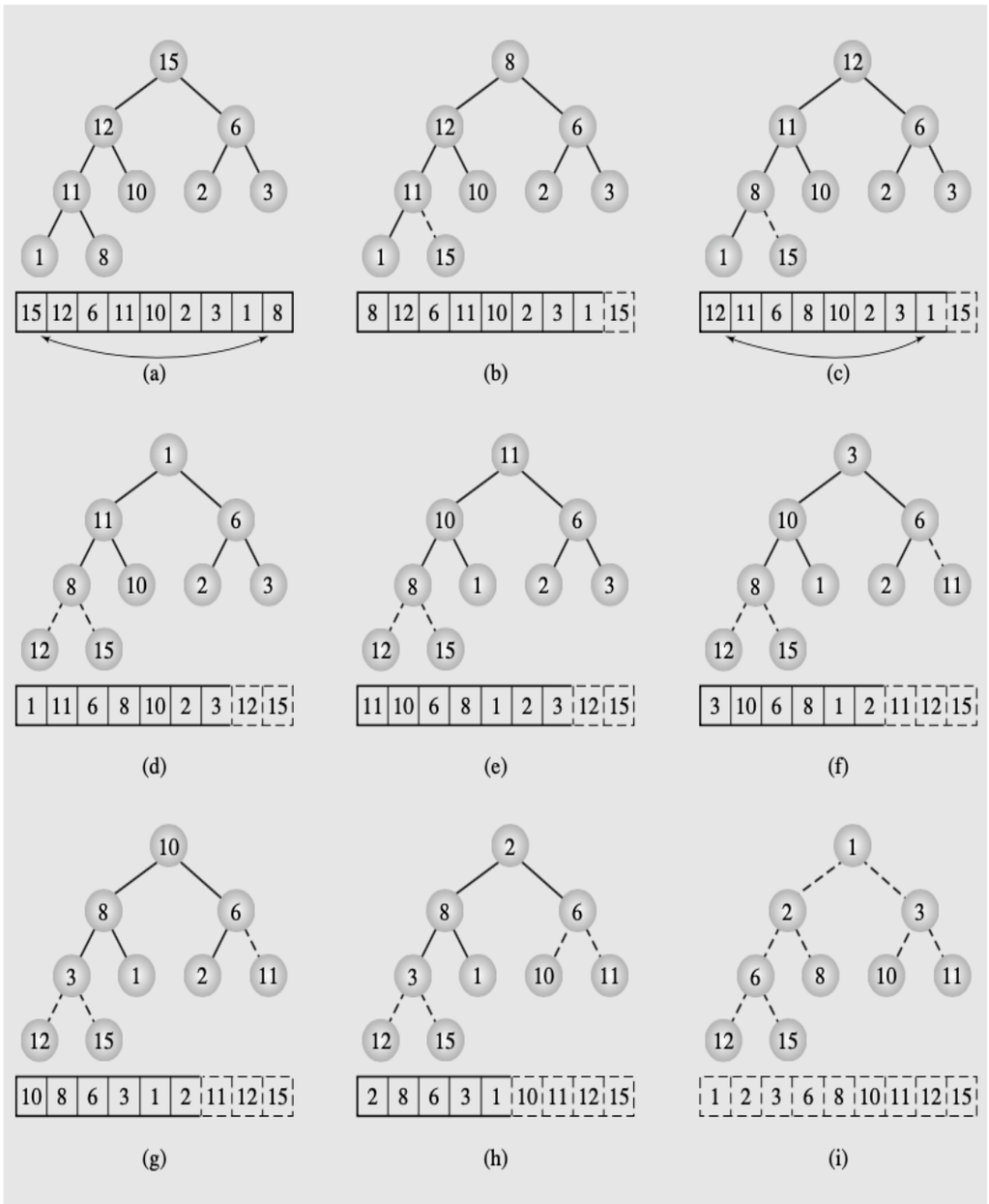
```
heapsort (data[] )
        transform data[] into a heap
        for i = data.length-1 down to 2
                swap the root with the element in position i;
                restore the heap property for the tree data[0],...,data[i-1];
```

Example: data[] = {2, 8, 6, 1, 10, 15, 3, 12, 11}

At first, transform data[] into a heap tree as below:



Then, Sort the above heap tree as below:



- **The Time complexity of Heap Sort is O(nlogn)**

3. Merge Sort:

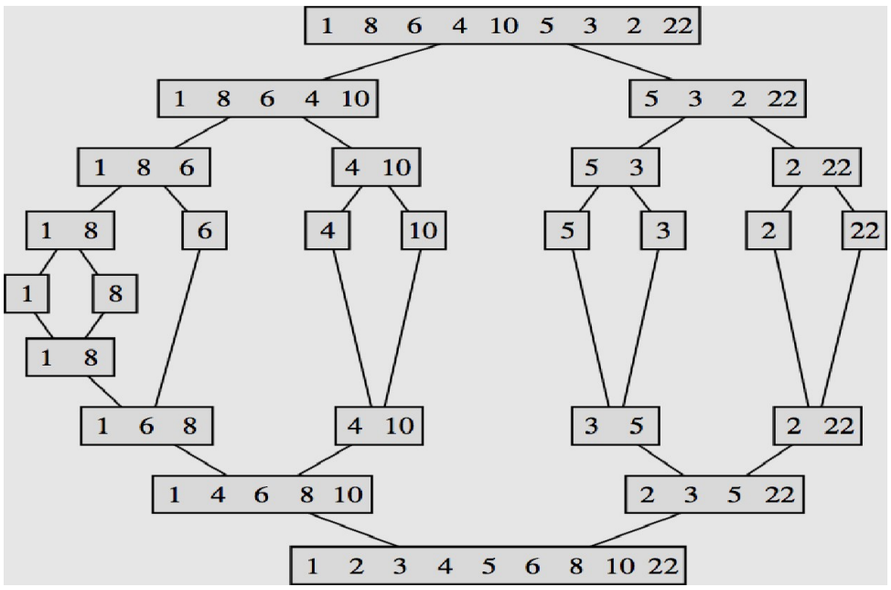
The problem with quick sort is that its complexity in the worst case is $O(n^2)$ because it is difficult to control the partitioning process. Different methods of choosing a bound attempt to make the behavior of this process fairly regular; however, there is no guarantee that portioning results in arrays of approximately the same size. Another strategy is to make portioning as simple as possible and concentrate on merging the two sorted arrays. This strategy is characteristic of merge sort. It was one of the first sorting algorithms used on a computer was developed by John von Neumann.

The key process in merge sort is merging sorted halves of an array into one sorted array. However, these halves have to be sorted first, which is accomplished by merging the already sorted halves of these halves. The process of dividing arrays into two halves stops when the array has fewer than two elements. The algorithm is recursive in nature and can be summarized in the following pseudocode:

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

The complexity of Merge Sort is O(nlogn)

Example: data[]={1, 8, 6, 4, 10, 5, 3, 2, 22}



4. Radix Sort:

Radix sort is a popular way of sorting used in everyday life. To sort library cards, we may create as many piles of cards as letters in the alphabet, each pile containing authors whose names start with the same letter. Then, each pile is sorted separately using the same method; namely, piles are created according to the second letter of the author's names. This process continues until the number of times the piles are divided into smaller piles equals the number of letters of the longest name. This method is actually used when sorting mail in the post office, and it was used to sort 80-column cards of coding information in the early days of computers.

- When sorting integers, 10 piles numbered 0 through 9 are created, and initially, integers are put in a given pile according to their rightmost digit so that 93 is put in pile 3. Then, piles are combined and the process is repeated, this time with the second rightmost digit; in this case, 93 ends up on pile 9. The process ends after the leftmost digit of the longest number is processed.

Algorithm:

radixsort()
for d = 1 to the position of the left most digit of longest number
distribute all numbers among piles 0 through 9 according to the d^{th} digit;
put all integers on one list;

data = [10 1234 9 7234 67 9181 733 197 7 3]

								7		
				3	7234			197		
	10	9181		733	1234			67		9
piles:	0	1	2	3	4	5	6	7	8	9

pass 1

data = [10 9181 733 3 1234 7234 67 197 7 9]

	9			7234						
	7			1234						
	3	10		733			67		9181	197
piles:	0	1	2	3	4	5	6	7	8	9

pass 2

data = [3 7 9 10 733 1234 7234 67 9181 197]

piles:	67									
	10									
	9									
	7	197	7234					773		
	3	9181	1234							
	0	1	2	3	4	5	6	7	8	9

pass 3

data = [3 7 9 10 67 9181 197 1234 7234 733]

piles:	733									
	197									
	67									
	10									
	9									
	7									
	3	1234						7234		9181
	0	1	2	3	4	5	6	7	8	9

pass 4

data = [3 7 9 10 67 197 733 1234 7234 9181]

Question: Sort the following number: 1, 234, 456, 654, 697, 874, 243, 385, 902, 23

The time complexity of Radix sort is O(nlogn)