SORTING ALGORITHM

INTRODUCTION TO SORTING

- Sorting is nothing but arranging the data in ascending or descending order. The term **sorting** came into picture, as humans realised the importance of searching quickly.

- Different Sorting Algorithms
- Bubble Sort
- Insertion Sort
- Selection Sort
- Quick Sort
- Merge Sort
- Becket Sort
- Shell Sort

BUBBLE SORT

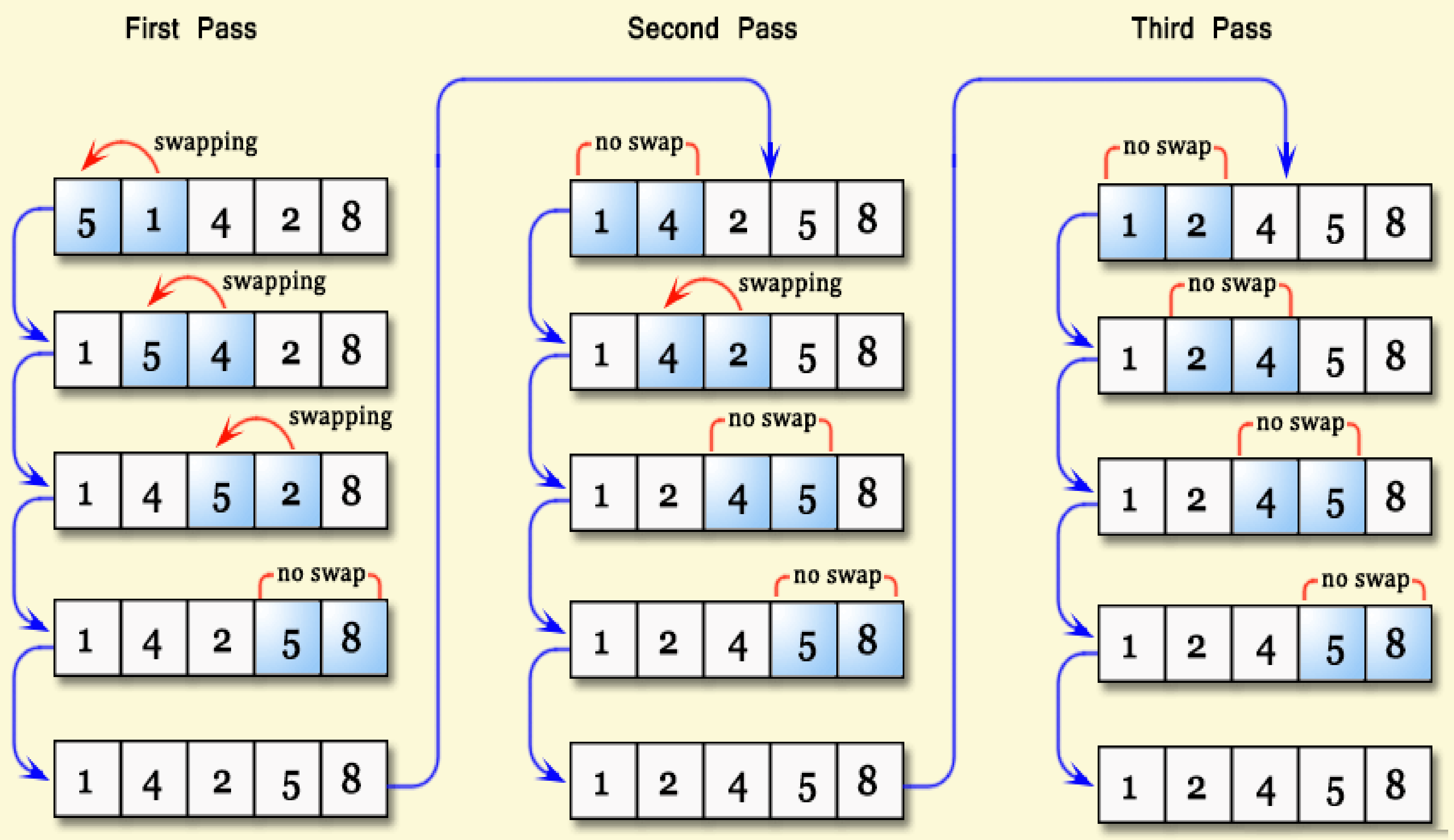
- **Bubble Sort** is a simple algorithm which is used to sort a given set of n elements provided in form of an array with n number of elements.

- Bubble Sort compares all the element one by one and sort them based on their values.

- This sorting algorithm is based on comparing and exchanging pairs of adjecent element in array. The bubble sort method derives it name from the fact that the smallest data item bubbles up to the top of the array.

Bubble Sort Example

Codingcompiler.com



BUBBLE SORT ALGO.

```
Step-1  [initialize]
         i=0 , j=0
Step-2  Repeat through step-6 while (i=n-1)
Step-3  Repeat through step-5 while (j<i)
Step-4  if(a[j]>a[i+1])
              t=a[j];
              a[j]=a[j+1]
              a[j+1]=t;
Step-5  j=j+1
Step-6  i=i-1
Step-7  exit
```

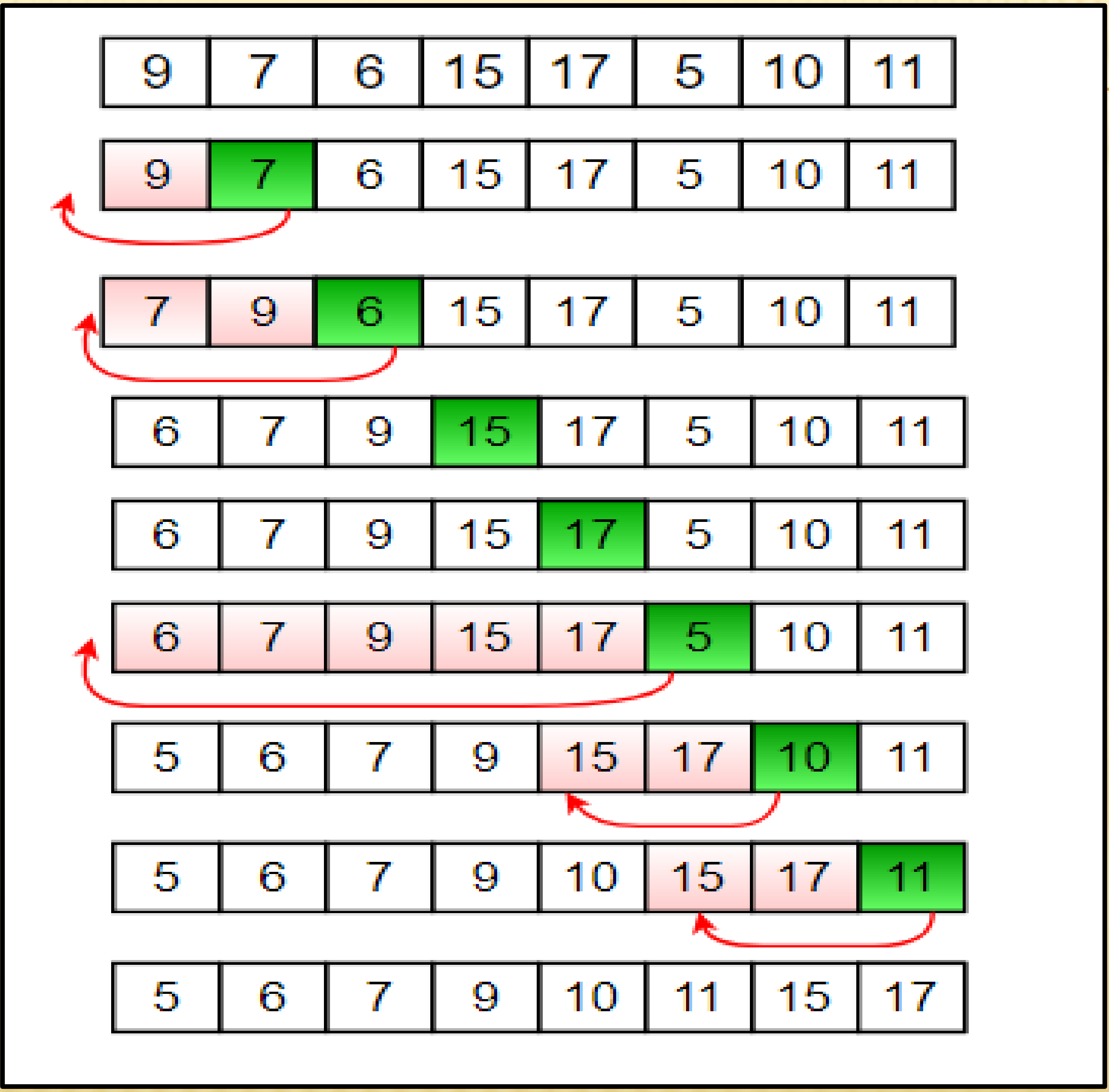
INSERTION SORT

- they are sorted, or arranged in the ascending order of their numbers.

- Its space complexity is less. Like bubble Sort, insertion sort also requires a single additional memory space.

- Insertion sort is a simple algorithm that is relatively efficient for small and mostly sorted list, and often Is used as part of more sophisticated algorithms.

- It works by taking element from the list one by one and inserting them in their correct position into a new sorted list. Insertion sort works just like its name suggest – it inserts each element into its proper place in the final list.


9 7 6 15 17 5 10 11
9 7 6 15 17 5 10 11
7 9 6 15 17 5 10 11
6 7 9 15 17 5 10 11
6 7 9 15 17 5 10 11
6 7 9 15 17 5 10 11
5 6 7 9 15 17 10 11
5 6 7 9 10 15 17 11
5 6 7 9 10 11 15 17

INSERTION SORT ALGO

```
Step-1  [initialize]
            i=0 , t=0
Step-2  Repeat through step-8 for while (i<size)
Step-3  j=0
Step-4  Repeat through step-7 for while (j<i)
Step-5  if(a[j]>a[i] then
                t=a[j];
                a[j]=a[i]
                k=i;
                Repeat through step-8 while (k>j)
                a[k]=a[k-1]
step-6  k--
Step-7  j=j+1
Step-8  i=i-1
Step-9  exit
```

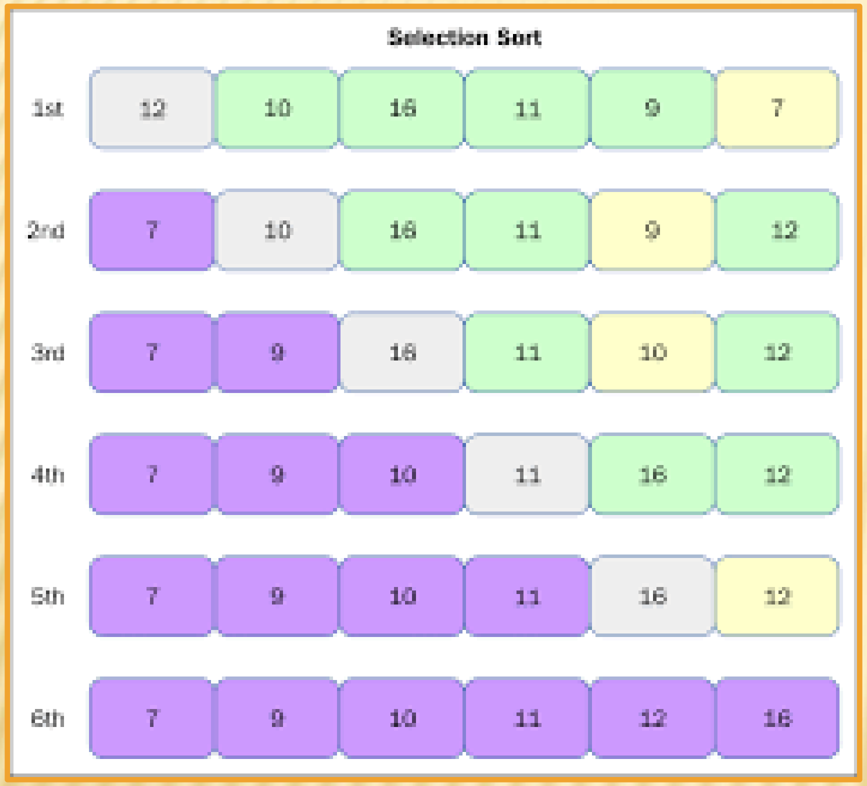
SELECTION SORT

- Selection sort is conceptually the most simplest sorting algorithm.

  - This algorithm will first find the **smallest** element in the array and swap it with the element in the **first** position,

  - then it will find the **second smallest** element and swap it with the element in the **second** position,

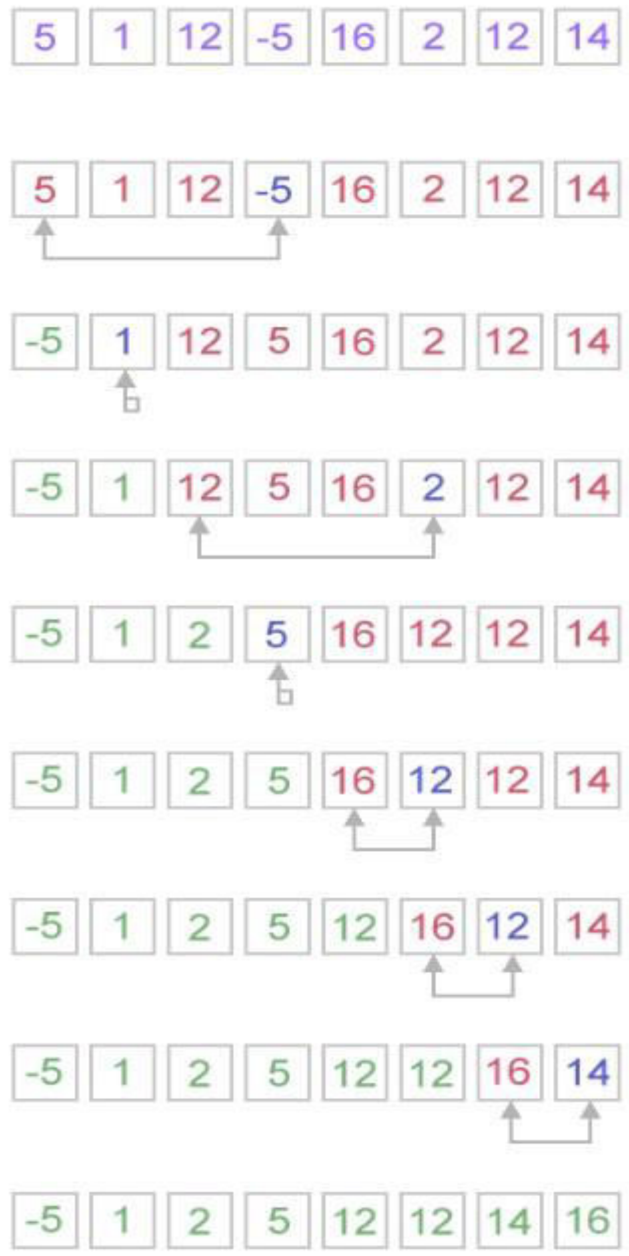
  - and it will keep on doing this until the entire array is sorted.


Selection Sort

1st	12	10	16	11	9	7
2nd	7	10	16	11	9	12
3rd	7	9	16	11	10	12
4th	7	9	10	11	16	12
5th	7	9	10	11	16	12
6th	7	9	10	11	12	16

Let us see an example of sorting an array to make the idea of selection sort cle

Example. Sort {5, 1, 12, -5, 16, 2, 12, 14} using selection sort.



5	1	12	-5	16	2	12	14
5	1	12	-5	16	2	12	14
-5	1	12	5	16	2	12	14
-5	1	12	5	16	2	12	14
-5	1	2	5	16	12	12	14
-5	1	2	5	16	12	12	14
-5	1	2	5	12	16	12	14
-5	1	2	5	12	12	16	14
-5	1	2	5	12	12	14	16

SELECTION SORT ALGO

```
Step-1  [initialize]
         i=0 ,j=0, t=0
Step-2  Repeat through step-6 for while (i<size)
Step-3  Repeat through step-5 for while (j<size)
Step-4  if(a[i]>a[j] then
             t=a[i];
             a[i]=a[j]
             a[j]=t
step-5  j=j+1
Step-6  i=i+1
Step-7  exit
```

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.

MARGE SORT ALGO

```
Step-1  [initialize]
        i=0,j=0,k=0
Step-2  Repeat through step-9 while (i<n)
Step-3  Repeat through step-8  while (j<i)
Step-4  if(a[j] > a[i])
              t=a[j]
              a[j]=a[i]
Step-5  Repeat through step-7 while (k>j)
Step-6  a[k]=a[k-1]
       a[k+1=t;
Step-7  K- -
Step-8  j=j+1
Step-9  i=i+1
Step-10  exit
```

QUICK SORT

- **Quick sort** is a highly efficient **sorting** algorithm and is based on partitioning of array of **data** into smaller arrays.

- This algorithm is quite efficient for large-sized **data** sets as its average and worst case complexity are of $O(n^2)$, where n is the number of items.

QUICK_SORT ALGO.

```
Quick_sort(a,first,last)
Step-1  [initialize]
          low=first,  high=last,  pivot=a[(first+last)/2]
Step-2   Repeat through step-7 while (low<=high)
Step-3   Repeat through step-4 while (a[low]<pivot)
Step-4   low=low+1
Step-5   Repeat through step-6 while (a[high]>pivot)
Step-6   high=high-1
Step-7   If(low<=high)
                 t=a[low]    a[low]=a[high]
                 a[high]=t   low=low+1     high=high-1
Step-8   If(first < high)
                 quick_sort(a,first,high)
Step-9   if(low<last)
                 quick_sort(a,low,last)
Step-10  exit
```

LINEAR SEARCH ALGORITHM

```
Linear(a,n)
Where a→ Represent the unsorted list
            n→found the key from the list
Step-1 [initialize]
         i=0 , found=0
Step-2 repeat step-4 for i<size
Step-3 if a[i]=key)
           --found=1
           --o/p→search is successful
           --o/p→key is searching at position(i+1)
Step-4 i=i+1
Step-5 if found=0
            --o/p→no searching
Step-6 exit
```

BINARY SEARCH ALGORITHM

```
Binary_search(a,n)
Where a→ represents unsorted list.
        n→search the key
Step-1 [intialize]
        low=0 ,  high=n-1 ,  flag=0
Step-2 Repeat through step-4 while (low<=high)
Step-3 mid=(low+high)/2
Step-4 if(n<arr[mid]) then
                high=mid-1
        else if (n>arr[mid]) then
                low=mid+1
        else if(n==arr[mid])
                o/p--search successful location element "mid+1"
        flag=1
Step-5 if flag==0 then
        o/p—search is un-successful
Step-6  exit
```