Data Structures and Analysis

UNIT-1

Analysis of Algorithms: Efficiency of algorithms, A priori Analysis, Asymptotic notations.
Searching: Introduction, linear search, binary search, Fibonacci search.
Sorting: Introduction, bubble sort, insertion sort, selection sort, quick sort, merge sort.

Analysis of algorithms:

Algorithm is a step by step procedure to solve a particular problem. Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

Characteristics of algorithm:

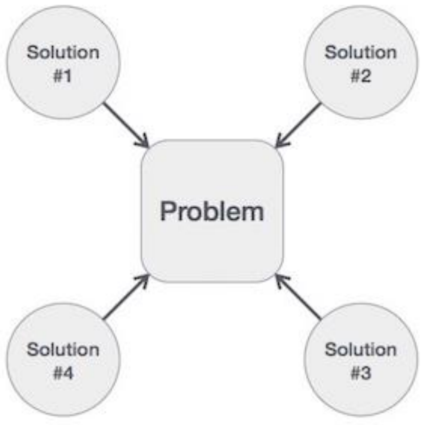
1. **Unambiguous**: Algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.
2. **Input** : An algorithm should have 0 or more well-defined inputs.
3. **Output**: An algorithm should have 1 or more well-defined outputs, and should match the desired output.
4. **Finiteness** − Algorithms must terminate after a finite number of steps.
5. **Feasibility** − Should be feasible with the available resources.

Example:

Problem − Design an algorithm to add two numbers and display the result.

Step 1 − START
Step 2 − declare three integers **a**, **b** & **c**
Step 3 − define values of **a** & **b**
Step 4 − add values of **a** & **b**
Step 5 − store output of step 4 to **c**
Step 6 − print **c**
Step 7 − STOP

We design an algorithm to get a solution of a given problem. A problem can be solved in more than one ways.



Hence, many solution algorithms can be derived for a given problem. The next step is to analyze those proposed solution algorithms and implement the best suitable solution.

Efficiency of algorithm:

Efficiency of an algorithm is a measure of the average execution time necessary for an algorithm to complete work on a set of data. That means

➔ How efficiency the problem has been solved
➔ How much computational resources are used (memory/CPU)

Efficiency of an algorithm can be analyzed at two different stages, before implementation and after implementation. They are the following –

- ***A Priori* Analysis** − This is a theoretical analysis of an algorithm. Efficiency of an algorithm is measured by assuming that all other factors, for example, processor speed, are constant and have no effect on the implementation.
- ***A Posterior* Analysis** − This is an empirical analysis of an algorithm. The selected algorithm is implemented using programming language. This is then executed on target computer machine. In this analysis, actual statistics like running time and space required are collected.

Differences between A priori analysis and A posterior analysis:

A priori analysis	A posteriori analysis
Priori analysis is an absolute analysis.	Posteriori analysis is a relative analysis.
It is independent of language of compiler and types of hardware.	It is dependent on language of compiler and type of hardware.
It will give approximate answer	It will give exact answer
It uses the asymptotic notations to represent how much time the algorithm will take in order to complete its execution.	It doesn't use asymptotic notations to represent the time complexity of an algorithm.
The time complexity of an algorithm using a priori analysis is same for every system	The time complexity of an algorithm using a posteriori analysis differ from system to system
If the program running faster, credit goes to the programmer.	If the time taken by the algorithm is less, then the credit will go to compiler and hardware.

Asymptotic Analysis:

As we know that data structure is a way of organizing the data efficiently and that efficiency is measured either in terms of time or space. So, the ideal data structure is a structure that occupies the least possible time to perform all its operation and the memory space. Our focus would be on finding the time complexity rather than space complexity, and by finding the time complexity, we can decide which data structure is the best for an algorithm. The main question arises in our mind that on what basis should we compare the time complexity of data structures?. The time complexity can be compared based on operations performed on them.

How to find the Time Complexity or running time for performing the operations?

The measuring of the actual running time is not practical at all. The running time to perform any operation depends on the size of the input. Let's understand this statement through a simple example.

Suppose we have an array of five elements, and we want to add a new element at the beginning of the array. To achieve this, we need to shift each element towards right, and suppose each element takes one unit of time. There are five elements, so five units of time would be taken. Suppose there are 1000 elements in an array, then it takes 1000 units of time to shift. It concludes that time complexity depends upon the input size.

Therefore, "if the input size is n, then f(n) is a function of n that denotes the time complexity".

Asymptotic Notations:

Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

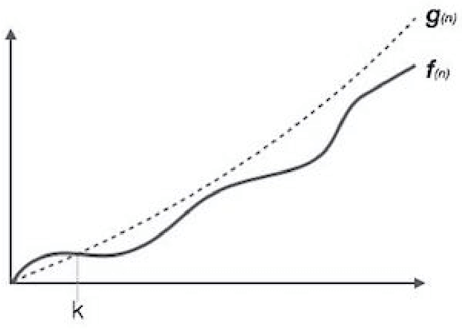
- O Notation
- Ω Notation
- θ Notation

Big Oh Notation, O:

- Big O notation is an asymptotic notation that measures the performance of an algorithm by simply providing the order of growth of the function.
- This notation provides an upper bound on a function which ensures that the function never grows faster than the upper bound. So, it gives the least upper bound on a function so that the function never grows faster than this upper bound.

F(n) <= c . g(n) or f(n) = O(g(n))
After some 'n'
$n >= n_0$ where c, n are real numbers $c>0$, $n_0 >=1$

The notation O(n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.



Example:
If **f(n)** and **g(n)** are the two functions defined for positive integers, then **f(n)** = **O(g(n))** as **f(n) is big oh of g(n)** or f(n) is on the order of g(n)) if there exists constants c and no such that:

f(n)≤c.g(n) for all n≥no

This implies that f(n) does not grow faster than g(n), or g(n) is an upper bound on the function f(n). In this case, we are calculating the growth rate of the function which eventually calculates the worst time complexity of a function, i.e., how worst an algorithm can perform.

Let us consider
f(n) = 3n+2,   g(n) = n
Can we say that f(n) = O(g(n)) ?
Initially we need to find **c, n_0**
f(n) <= c.g(n)   c>0 and $n_0 >=1$

3n+2 <= c.n
c=1   3n+2 <= (1).n => 3n+2<=1n
c=2   3n+2 <= (2).n => 3n+2<=2n
c=3   3n+2 <= (3).n => 3n+2<=3n

c=4 3n+2 <= (4).n => 3n+2<=4n

2 <= 4n-3n

$\boxed{c=4}$ $\boxed{n>=2}$

For every n>=2 at c=4 f(n)<=c.g(n) such that

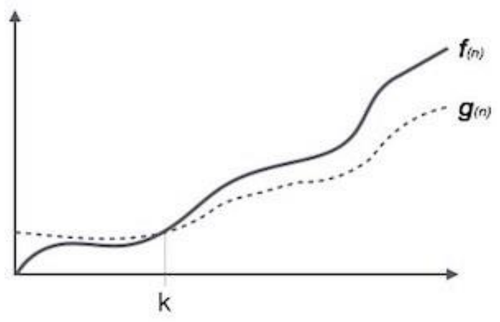
$\boxed{f(n) = O(c.g(n)) \text{ or } f(n) = O(g(n))}$

Omega Notation, Ω:

- It basically describes the best-case scenario which is opposite to the big o notation.
- It is the formal way to represent the lower bound of an algorithm's running time. It measures the best amount of time an algorithm can possibly take to complete or the best-case time complexity.
- It determines what the fastest time that an algorithm can run is.

F(n) >= c . g(n) or f(n) = Ω (g(n))
After some 'n'
n>= n_0 where c, n are real numbers c>0 , n_0 >=1

If we required that an algorithm takes at least certain amount of time without using an upper bound, we use big- Ω notation i.e. the Greek letter "omega". It is used to bound the growth of running time for large input size.
If **f(n)** and **g(n)** are the two functions defined for positive integers, then **f(n) = Ω (g(n))** as **f(n) is Omega of g(n)** or f(n) is on the order of g(n)) if there exists constants c and no such that:
f(n)>=c.g(n) for all n≥no and c>0



Let us consider
f(n) = 3n+2, g(n) = n
Can we say that f(n) = **Ω** (g(n)) ?
Initially we need to find **c, n_0**
f(n) >= c.g(n) c>0 and n_0 >=1
3n+2 >= c.n
c=1 3n+2 >= (1).n => 3n+2>=1n
3n+2>=1n

$\boxed{c=1}$ $\boxed{n<=1}$

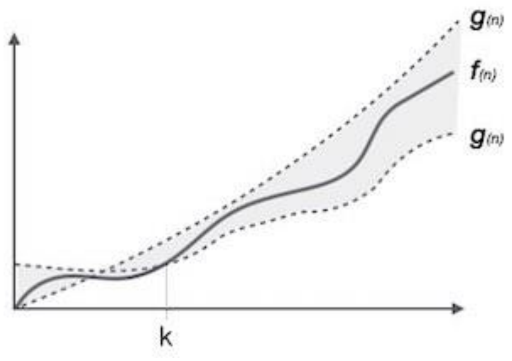
For every n<=1 at c=1 f(n)>=c.g(n) such that f(n) = **Ω** (c.g(n)) or f(n) = **Ω** (g(n))

Theta Notation, θ:

- The theta notation mainly describes the average case scenarios.
- It represents the realistic time complexity of an algorithm. Every time, an algorithm does not perform worst or best, in real-world problems, algorithms mainly fluctuate between the worst-case and best-case, and this gives us the average case of the algorithm.
- Big theta is mainly used when the value of worst-case and the best-case is same.
- It is the formal way to express both the upper bound and lower bound of an algorithm running time.

c1.g(n)<=f(n)<=c2.g(n) or f(n) = θ (g(n))
After some 'n'
$n >= n_0$ where c, n are real numbers c>0 , $n_0 >= 1$

Let's understand the big theta notation mathematically:
Let f(n) and g(n) be the functions of n where n is the steps required to execute the program then:
f(n)= θ(g(n))
The above condition is satisfied only if when
c1.g(n)<=f(n)<=c2.g(n)
where the function is bounded by two limits, i.e., upper and lower limit, and f(n) comes in between.
The condition **f(n)= θ(g(n))** will be true if and only if c1.g(n) is less than or equal to f(n) and c2.g(n) is greater than or equal to f(n).



Example:
f(n) = 3n+2, g(n) = n
Initially we need to find **c, n_0**
f(n) <= c1.g(n) c1>0 and $n_0 >= 1$
3n+2 <= c.n
c1=1 3n+2 <= (1).n => 3n+2<=1n
c1=2 3n+2 <= (2).n => 3n+2<=2n
c1=3 3n+2 <= (3).n => 3n+2<=3n
c1=4 3n+2 <= (4).n => 3n+2<=4n
2 <= 4n-3n

$\boxed{c1=4}$ $\boxed{n>=2}$

For every n>=2 at c1=4

$f(n) <= c1.g(n)$

$f(n) = 3n+2$, $g(n) = n$

Initially we need to find **c, n_0**

$f(n) >= c2.g(n)$ $c2>0$ and $n_0 >=1$

$3n+2 >= c2.n$

$c2=1$ $3n+2 >= (1).n$ => $3n+2>=1n$

$\boxed{C2=1}$ $\boxed{n<=1}$

For every $n<=1$ at $c2=1$

$f(n)>=c2.g(n)$

$$\boxed{\mathbf{c1.g(n)>=f(n)>=c2.g(n) \text{ or } c2.g(n)<=f(n)<=c1.g(n) \text{ or } f(n) = \theta (g(n))}}$$

Average, Best, Worst case complexities:

Usually, the time required by an algorithm falls under three types −

- Best Case − Minimum time required for program execution.
- Average Case − Average time required for program execution.
- Worst Case − Maximum time required for program execution.

Analyzing recursive programs:

The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function. Using recursive algorithm, certain problems can be solved quite easily. Examples of such problems are Towers of Hanoi (TOH), Tree Traversals, DFS of Graph, etc.

For every recursive algorithm, we can write recurrence relation to analyse the time complexity of the algorithm.

Searching

Searching techniques are used to retrieve a particular record from a list of records in an efficient manner so that least possible time is consumed. The list of records can have a key field, which can be used to identify a record. Therefore, the given value must be matched with the key value in order to find a particular record. If the searching technique identifies a particular record, then the search operation is said to be successful; otherwise, it is said to be unsuccessful.

The techniques which are used to search data from the memory of the computer are called internal searching techniques. The techniques that are used to search data from a storage device- such as a hard disk or a floppy disk-are called external searching techniques.

Searching techniques:

- Linear search
- Binary search
- Fibonacci search

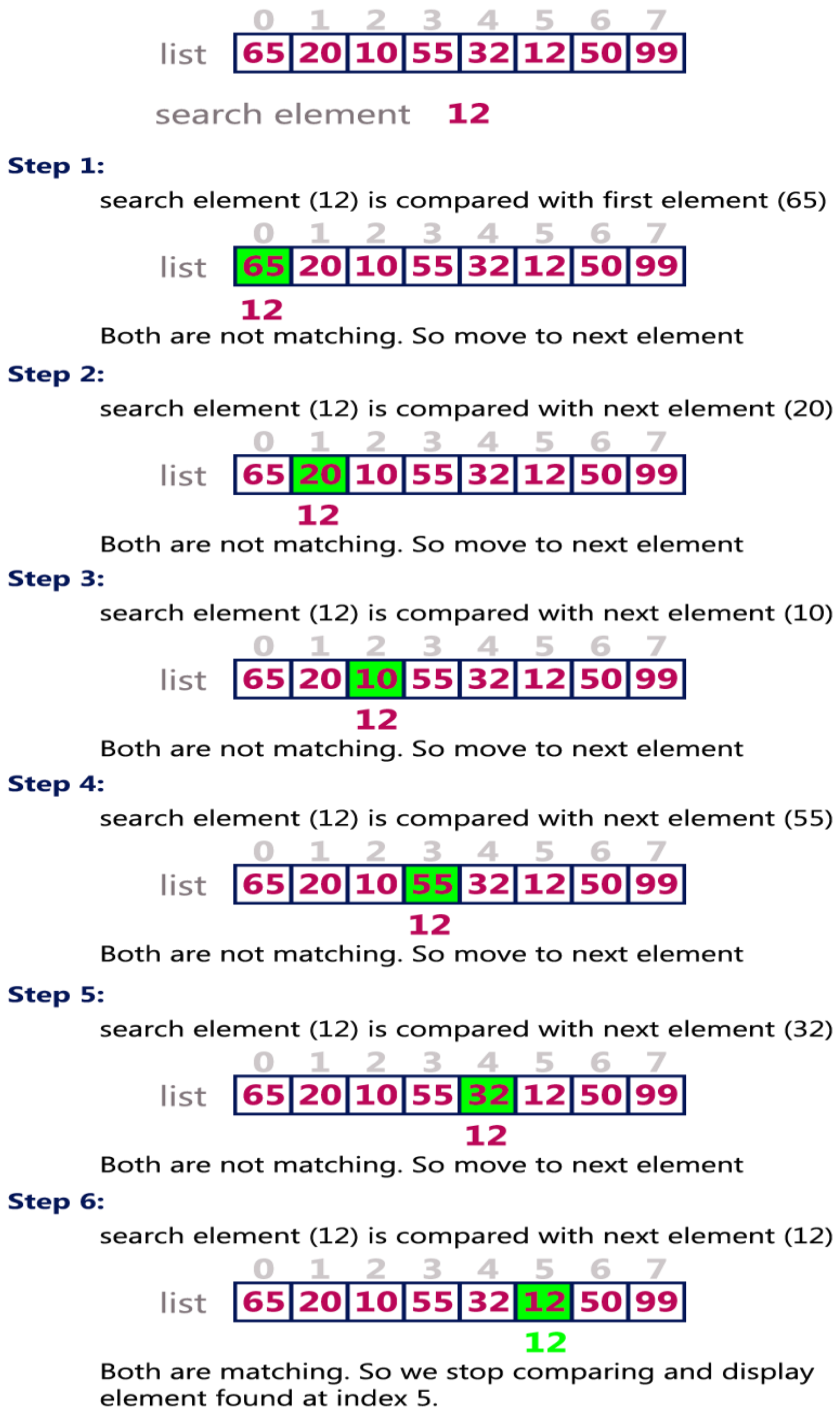
Linear Search:

Linear search, also known as sequential search involves searching a data item from a given set of data items. These data items are stored in structures, such as arrays. Each data item is checked one by one in the order in which it exists in the structure to determine if it matches the criteria of search.

Linear search is implemented using following steps...

- Step 1 - Read the search element from the user.
- Step 2 - Compare the search element with the first element in the list.
- Step 3 - If both are matched, then display "Given element is found!!!" and terminate the function
- Step 4 - If both are not matched, then compare search element with the next element in the list.
- Step 5 - Repeat steps 3 and 4 until search element is compared with last element in the list.
- Step 6 - If last element in the list also doesn't match, then display "Element is not found!!!" and terminate the function.

Example:



Step 1:
search element (12) is compared with first element (65)

Both are not matching. So move to next element

Step 2:
search element (12) is compared with next element (20)

Both are not matching. So move to next element

Step 3:
search element (12) is compared with next element (10)

Both are not matching. So move to next element

Step 4:
search element (12) is compared with next element (55)

Both are not matching. So move to next element

Step 5:
search element (12) is compared with next element (32)

Both are not matching. So move to next element

Step 6:
search element (12) is compared with next element (12)

Both are matching. So we stop comparing and display element found at index 5.

Advantages of Linear Searching

1. It is simple to implement
2. It does not require specific ordering before applying the method

Disadvantages of Linear Searching

1. It is less efficient.

```
/*Linear Search algorithm*/

    Linear Search ( Array A, key)
    {
       count =0;
       for(i = 0; i < n; i++)
       {
              if(A[i] = = key)
                    count= count+1;
                    break;
       }
      if(count == 1)
              print "element is found at position".
      else
              print "element not found".
    }
```

Time complexity of linear search:

Best case: The element being searched may be found at the first position. In this case, the search terminates in success with just one comparison. Thus in best case, linear search algorithm takes O(1) comparisons.

Worst case: The element being searched may be present at the n position or not present in the array at all. In the former case, the search terminates in success with n comparisons. In the later case, the search terminates in failure with n comparisons. Thus in worst case, linear search algorithm takes O(n) comparisons**.**

Average case: The element being searched may be present at the middle position. Thus in average case, linear search algorithm takes O(n) comparisons.

Binary search:

Binary search is the fastest searching algorithm. It works on the ordered list either in ascending order or descending order**.** The principle used in this search is to divide the list into two halves and compare the key with the middle element. If the comparison results in true then print its position. If it is false, then check key element is greater than the middle element or less than the middle element. If the key element is greater than the middle element then search the key element in second half of the list. If the key is less than the middle element then search the key in the first half of the list**.** Same process is repeated for both the sub lists depending upon the key present in half of the list or second half of the list.

Binary search is implemented using following steps...

- Step 1 - Read the search element from the user.
- Step 2 - Find the middle element in the sorted list.
- Step 3 - Compare the search element with the middle element in the sorted list.
- Step 4 - If both are matched, then display "Given element is found!!!" and terminate the function.
- Step 5 - If both are not matched, then check whether the search element is smaller or larger than the middle element.
- Step 6 - If the search element is smaller than middle element, repeat steps 2, 3, 4 and 5 for the left sublist of the middle element.
- Step 7 - If the search element is larger than middle element, repeat steps 2, 3, 4 and 5 for the right sublist of the middle element.

- Step 8 - Repeat the same process until we find the search element in the list or until sublist contains only one element.
- Step 9 - If that element also doesn't match with the search element, then display "Element is not found in the list!!!" and terminate the function**.**

Advantages of binary searching

1. It is an efficient technique.

Disadvantages of binary searching

1. It requires specific ordering before the applying the methods
2. It is complex to implement.

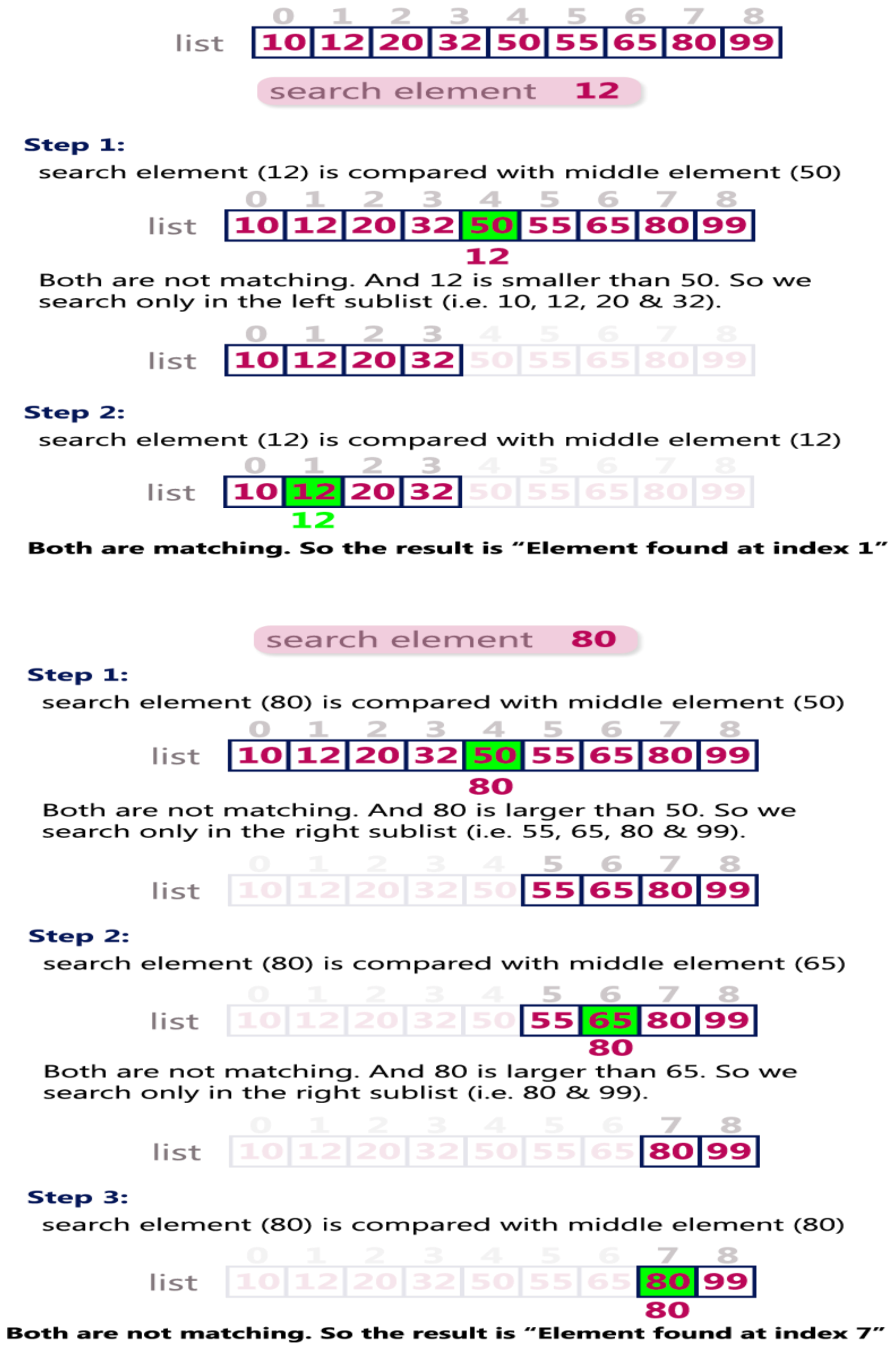
/*binary search algorithm*/

```
Binarysearch(A, n, key)
{
    low=0, high=n;
    while(low < =high)
    {
      mid=(low + high)/2;
      if(a[mid] > key)
                high=mid - 1;
      else if(a[mid] <key)
                low = mid + 1;
      else if(key==a[mid])
                print  "The element is found "
      else
                print "element is not found"
    }
 }
```

Time complexity:

- Best case -   O(1)
- Average case - O(log n)
- Worst case – O(log n)

Example:


0 1 2 3 4 5 6 7 8
list 10 12 20 32 50 55 65 80 99

search element 12

Step 1:
search element (12) is compared with middle element (50)

0 1 2 3 4 5 6 7 8
list 10 12 20 32 50 55 65 80 99
12

Both are not matching. And 12 is smaller than 50. So we search only in the left sublist (i.e. 10, 12, 20 & 32).

0 1 2 3 4 5 6 7 8
list 10 12 20 32 50 55 65 80 99

Step 2:
search element (12) is compared with middle element (12)

0 1 2 3 4 5 6 7 8
list 10 12 20 32 50 55 65 80 99
12

Both are matching. So the result is "Element found at index 1"

search element 80

Step 1:
search element (80) is compared with middle element (50)

Both are not matching. And 80 is larger than 50. So we search only in the right sublist (i.e. 55, 65, 80 & 99).

Step 2:
search element (80) is compared with middle element (65)

Both are not matching. And 80 is larger than 65. So we search only in the right sublist (i.e. 80 & 99).

Step 3:
search element (80) is compared with middle element (80)

Both are not matching. So the result is "Element found at index 7"

Fibonacci Search:

In binary search method we divide the number at mid and based on mid element i.e. by comparing mid element with key element we search either the left sublist or right sublist. Thus we go on dividing the corresponding sublist each time and comparing mid element with key element. If the key element is really present in the list, we can reach to the location of key in the list and this we get the message the "element is present in the list" otherwise get the message. "element is not present in the list".

Algorithm:

```
Step1: initially find Fibonacci series of a given number n, find F(k) -> (k is the Fibonacci number) which is greater than or equal to n
Step2: if F(k)=0 stop and print element not foud
Step3: set offset = -1
Step4: find    i=min(offset+F(k-2),n-1)
Step 5: if(key= =a[i])
        Return a[i] and print element found
Step6: if(key>a[i])
        k=k-1 , offset =i repeat 4,5
step7: if(key<a[i])
        k=k-2 repeat 4,5
```

In fiboacci search rather than considering the mid element, we consider the indices as the numbers from Fibonacci series. As we know, the Fibonacci series is…

0 1 1 2 3 5 8 13 21….

To understand how Fibonacci search works, we will consider one example, suppose following is the list of elements.

Arr[]

10	20	30	40	50	60	70
0	1	2	3	4	5	6

k =	0	1	2	3	4	5	6
F(k)=	0	1	1	2	3	5	8

k=6 , F(k)=8, offset= -1, key=40
i= min(offset+F(k-2), n-1)
 = min(-1+F(4),6)
 = min(-1+3,6)
 =min(2,6)
i=2 => a[i]=2 =>a[2]=30

if(40==30) ->false
if(40>30) -> true
  k=k-1 , offset=i
  k=5, offset=2
i=min(offset+F(k-2), n-1)

```
 = min(2+F(3),6)
 = min(2+2,6)
 i=min(4,6)
i=4    =>  a[i]=4    =>a[4]=50

if(40==50)   ->false
if(40>50) -> false
if(40<50) -> true
   k=k-2 , offset=2
   k=3

i=min(offset+F(k-2), n-1)
 = min(2+F(1),6)
 = min(2+1,6)
 i=min(3,6)
i=3    =>  a[i]=3    =>a[3]=40

if(40==40)   ->true
so print "element is found"
```

Time Complexity:

- Best case: O(1)
- Average case: O(log n)
- Worst case: O(log n)

Sorting

Sorting is an important concept that is extensively used in the fields of computer science. Sorting is nothing but arranging the elements in some logical order. For example, we want to obtain the telephone number of a person. If the telephone directory is not arranged in alphabetical order, one has to search from the very first page to till the last page. If the directory is sorted, we can easily search for the telephone number.

Bubble sort:

This is the simplest sorting technique when compare with all other sorting.
The bubble sort technique starts with comparing the first two data items of an array and swapping these two data items if they are not in a proper order, i.e. in descending or ascending order. This process continues until all the data items in the array are compared or the end of an array is reached.

How Bubble Sort will work?

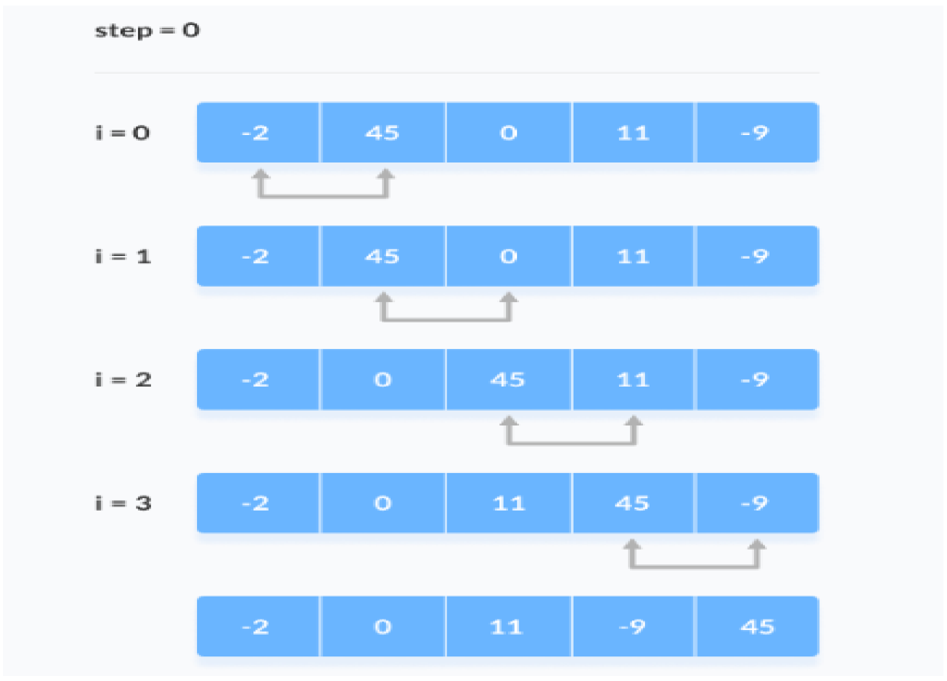
1. Starting from the first index, compare the first and the second elements.
2. If the first element is greater than the second element, they are swapped.

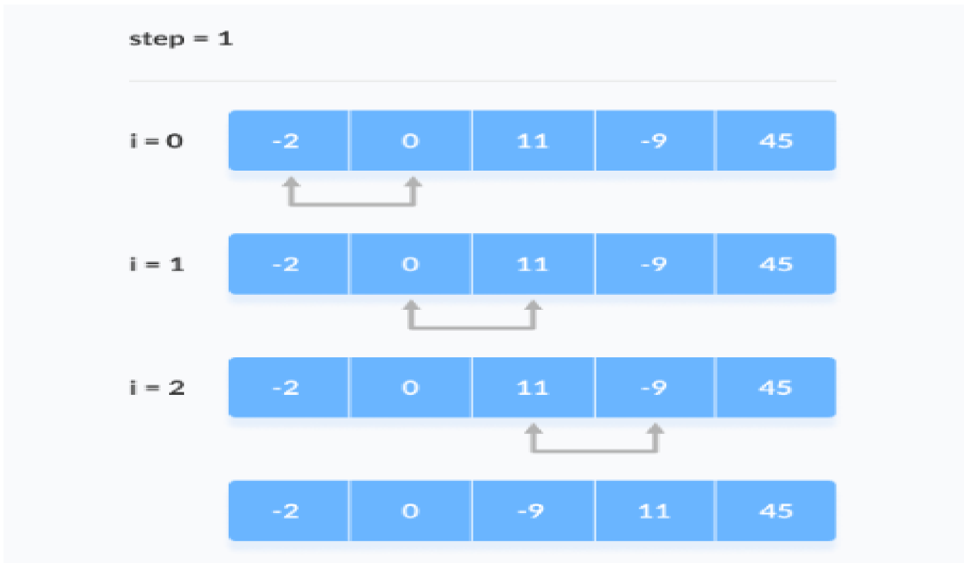
3. The same process goes on for the remaining iterations. After each iteration, the largest element among the unsorted elements is placed at the end.
4. The array is sorted when all the unsorted elements are placed at their correct positions.

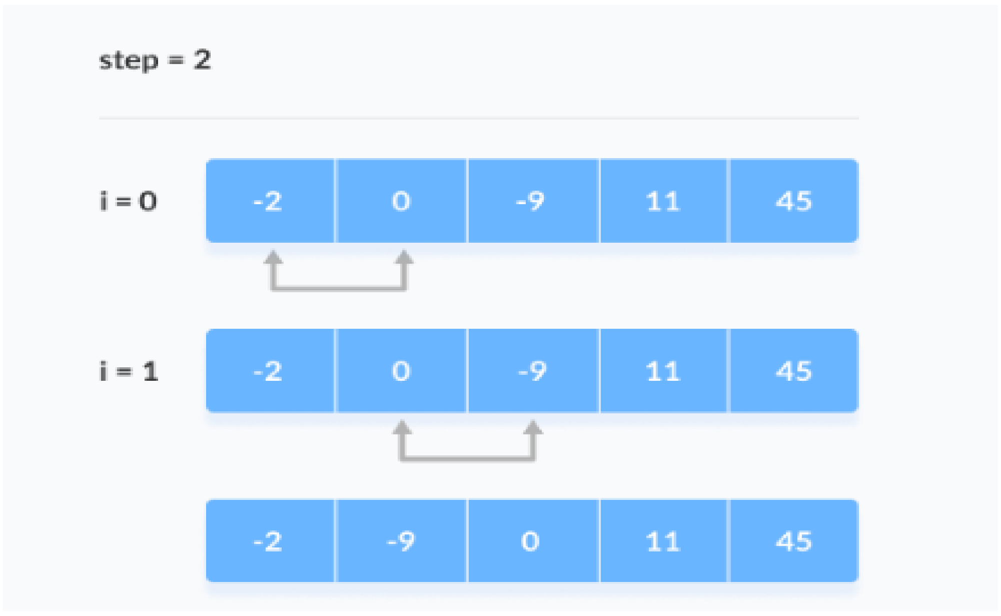
Algorithm:

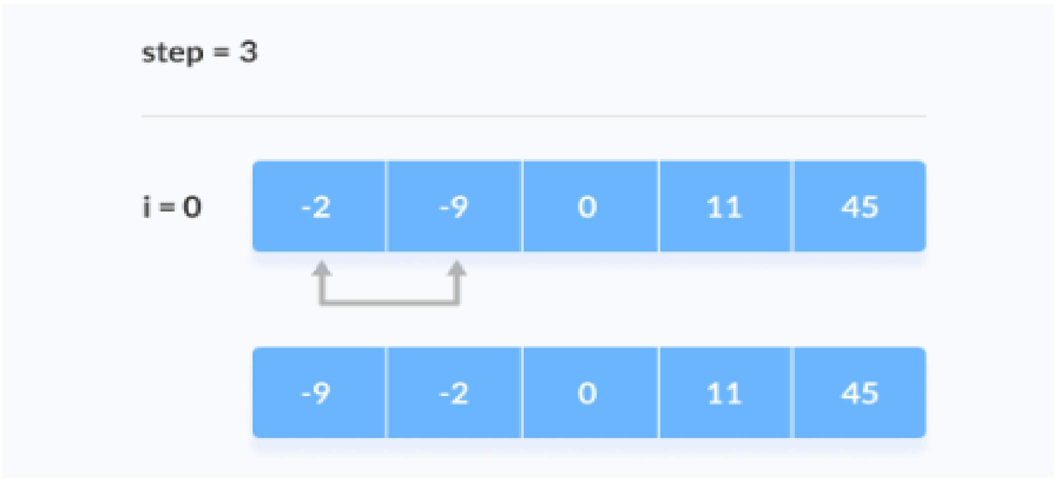
```
begin BubbleSort(array)
        for all elements of list
             if (array[i] > array[i+1])
                    swap(array[i], array[i+1])
             end if
        end for
        return list
end BubbleSort
```

Example









Time complexity

- Best case : O(n^2)
- Worst case: O(n^2)
- Average case: O(n^2)

Advantages

- The primary advantage of the bubble sort is that it is popular and easy to implement.
- In bubble sort, elements are swapped in place without using additional temporary storage, so the space requirement is at a minimum.

Disadvantage

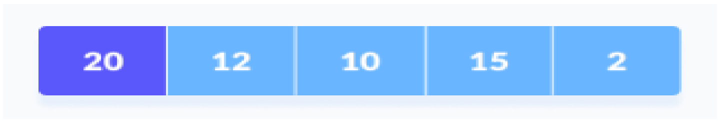
- The main disadvantage of the bubble sort is that it does not deal well with a list containing a huge number of items.

Selection sort:

Selection sort is an algorithm that selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list.

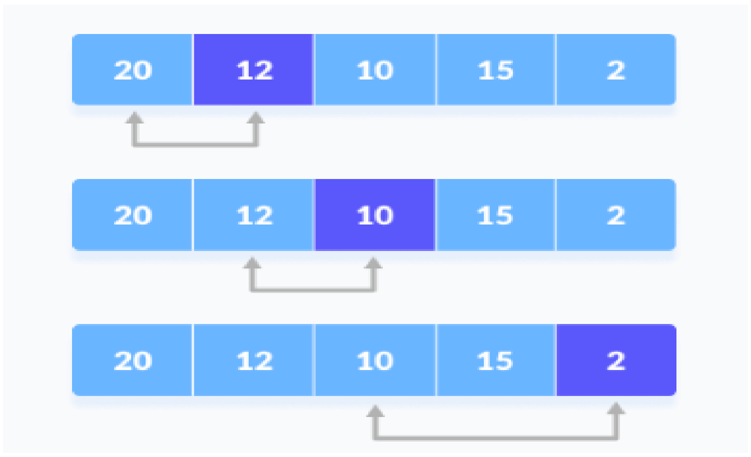
How Selection Sort Works?

1. Set the first element as ‘minimum’.

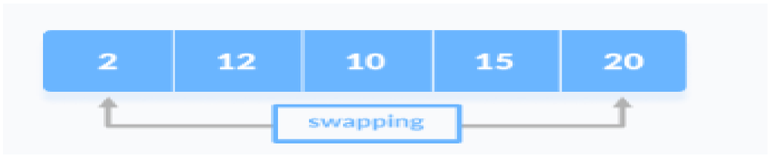


2. Compare ’minimum’ with the second element. If the second element is smaller than minimum, assign the second element as minimum.

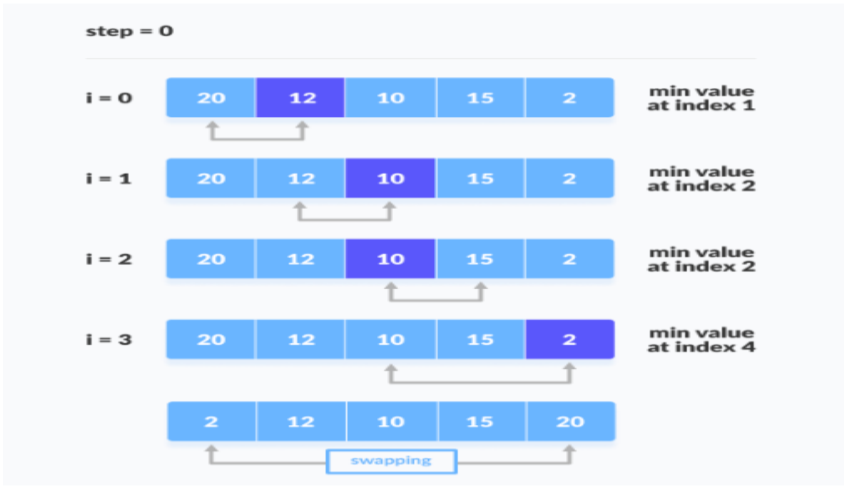
Again, if the third element is smaller, then assign minimum to the third element otherwise do nothing. The process goes on until the last element.

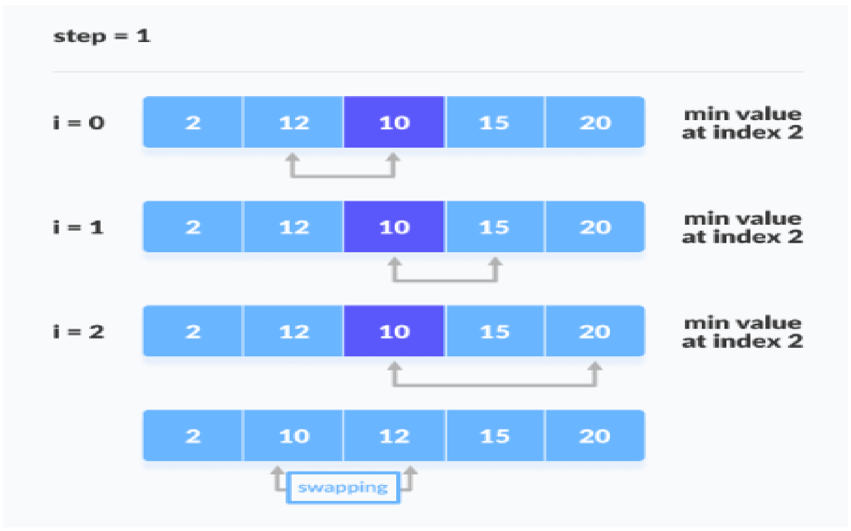


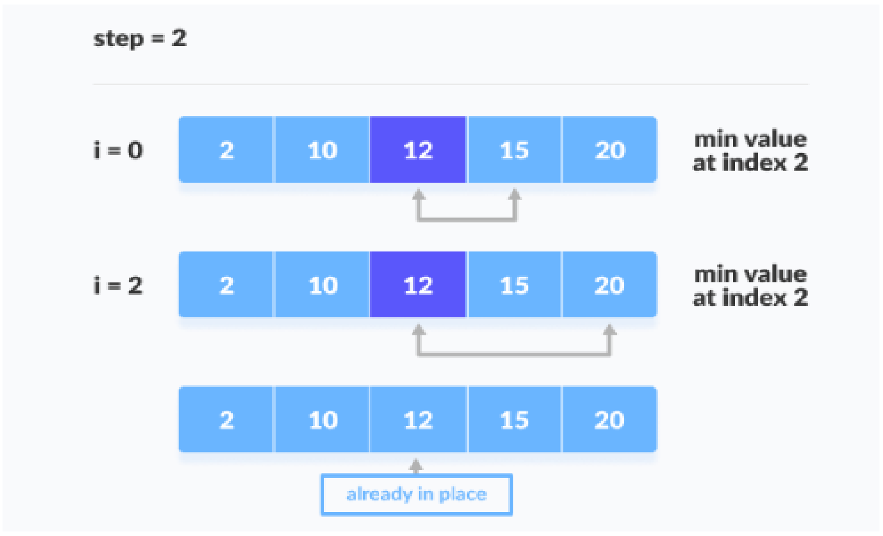
3. After each iteration, minimum is placed in the front of the unsorted list.

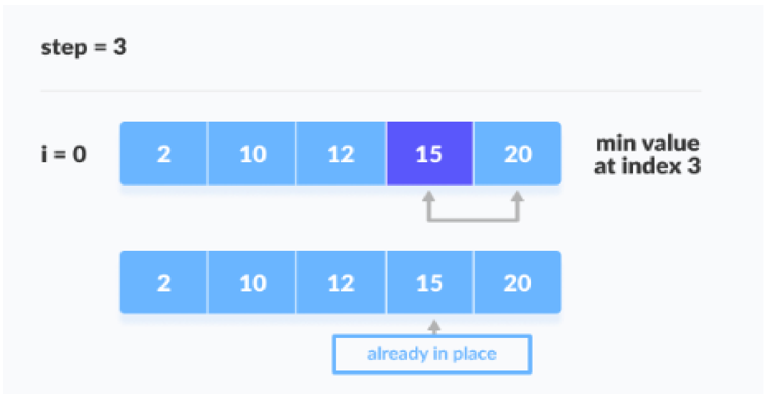


4. For each iteration, indexing starts from the first unsorted element. Step 1 to 3 are repeated until all the elements are placed at their correct positions.





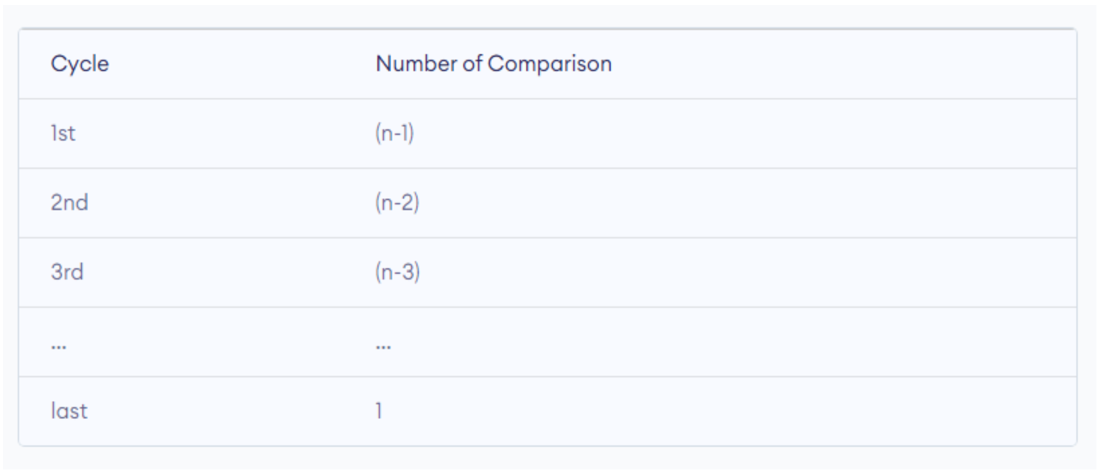




Selection sort algorithm:

Step 1 − Set MIN to location 0
Step 2 − Search the minimum element in the list
Step 3 − Swap with value at location MIN
Step 4 − Increment MIN to point to next element
Step 5 − Repeat until list is sorted

Time complexity



Cycle	Number of Comparison
1st	(n-1)
2nd	(n-2)
3rd	(n-3)
...	...
last	1

Number of comparisons: $(n - 1) + (n - 2) + (n - 3) + ... + 1 = n(n - 1) / 2$ nearly equals to n^2. i.e, $O(n^2)$
Also, we can analyze the complexity by simply observing the number of loops. There are 2 loops so the complexity is $n*n = n^2$.

Worst Case Complexity: $O(n^2)$: If we want to sort in ascending order and the array is in descending order then, the worst case occurs.

Best Case Complexity: $O(n^2)$ It occurs when the array is already sorted

Average Case Complexity: $O(n^2)$ It occurs when the elements of the array are in jumbled order (neither ascending nor descending).

Advantages:

- Selection sort algorithm is 60% more efficient than bubble sort algorithm.
- Selection sort algorithm is easy to implement.
- Selection sort algorithm can be used for small data sets, unfortunately Insertion sort algorithm best suitable for it.

Disadvantage

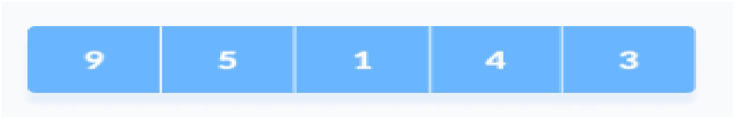
- Running time of Selection sort algorithm is very poor of $0 (n^2)$.
- Insertion sort algorithm is far better than selection sort algorithm.

Insertion sort:

The insertion sort works very well when the number of elements are very less. This technique is similar to the way a librarian keeps the books in the shelf. Initially all the books are placed in shelf according to their access number. When a student returns the book to the librarian, he compares the access number of this book with all other books of access numbers and inserts it into the correct position, so that all books are arranged in order with respect to their access numbers.

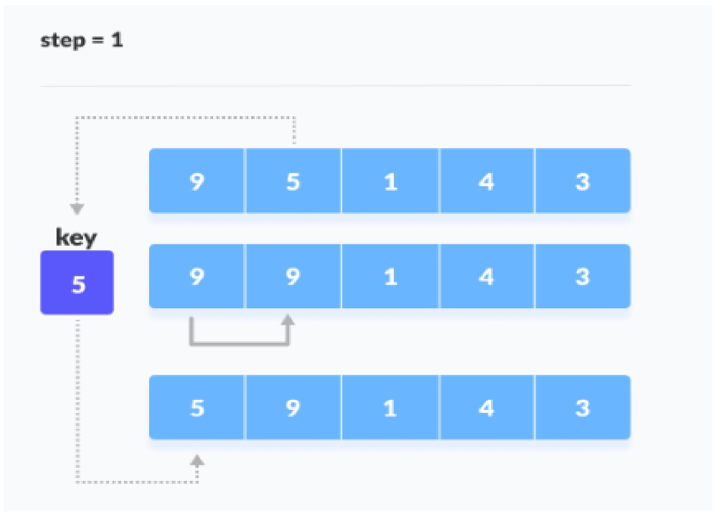
How Insertion Sort Works?

Suppose we need to sort the following array.



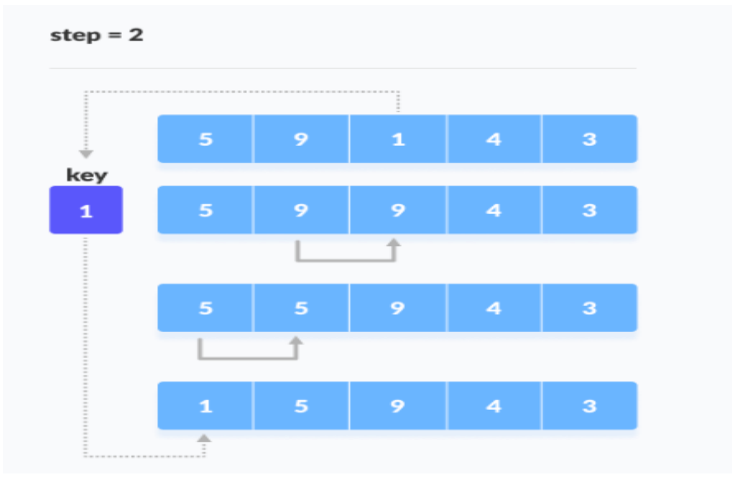
1. The first element in the array is assumed to be sorted. Take the second element and store it separately in key.

Compare key with the first element. If the first element is greater than key, then key is placed in front of the first element.

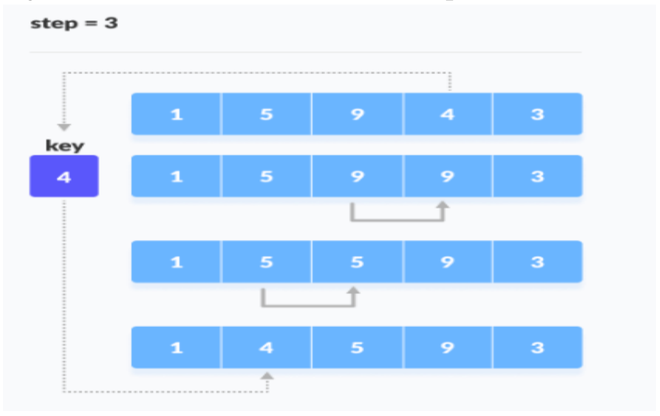


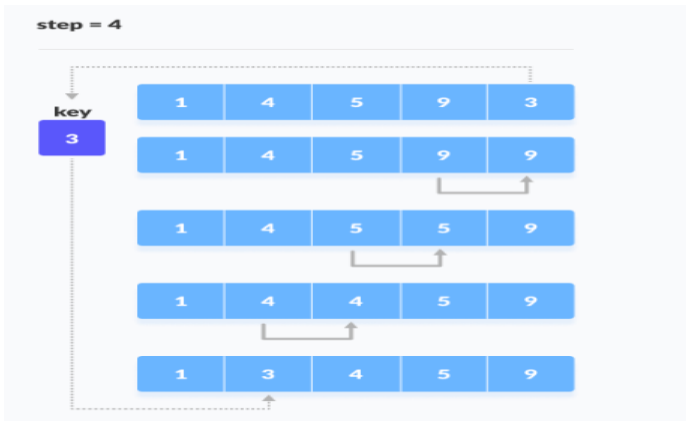
2. Now, the first two elements are sorted.

- Take the third element and compare it with the elements on the left of it. Placed it just behind the element smaller than it. If there is no element smaller than it, then place it at the beginning of the array.



3. Similarly, place every unsorted element at its correct position.





Insertion sort algorithm
Step 1 − If it is the first element, it is already sorted. return 1;
Step 2 − Pick next element
Step 3 − Compare with all elements in the sorted sub-list
Step 4 − Shift all the elements in the sorted sub-list that is greater than the value to be sorted .
Step 5 − Insert the value.
Step 6 − Repeat until list is sorted
Time complexity

- **Worst case complexity: O(n^2)** Suppose, an array is in ascending order, and you want to sort it in descending order. In this case, worst case complexity occurs.
- **Best Case Complexity:** O(n) when the array is already sorted, the outer loop runs for n number of times whereas the inner loop does not run all. So, there are only n number of comparisons. Thus, complexity is linear.
- **Average Case Complexity:** O(n^2) It occurs when the elements of an array are in jumbled order (neither ascending nor descending).

Advantages

- The insertion sort is an in-place sorting algorithm so the space requirement is minimal.
- It also exhibits a good performance when dealing with a small list.

Disadvantage

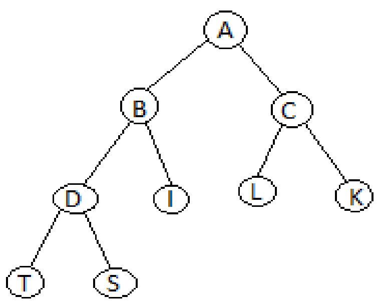
- With n-squared steps required for every n element to be sorted, the insertion sort does not deal well with a huge list.
- The disadvantage of the insertion sort is that it does not perform as well as other, better sorting algorithms

Heap Sort:

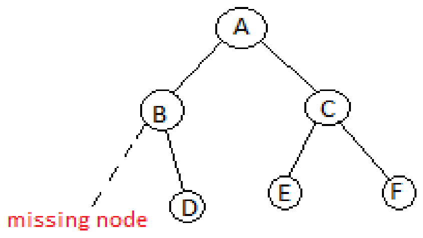
- Heap sort is one of the sorting algorithms used to arrange a list of elements in an order. Heap sort algorithm uses one of the tree concepts called **Heap Tree**.
- In this sorting algorithm, we use **Max Heap** to arrange list of elements in Ascending order and **Min Heap** to arrange list elements in Descending order.

What is a heap?
Heap is a special tree-based data structure, that satisfies the following special heap properties:

1. **Shape Property:** Heap data structure is always a Complete Binary Tree, which means all levels of the tree are fully filled.

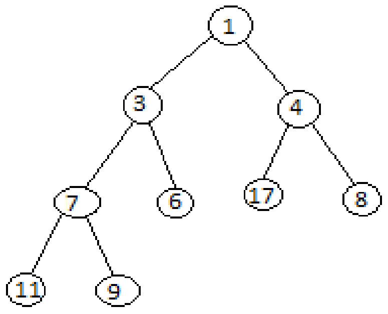


Complete Binary Tree

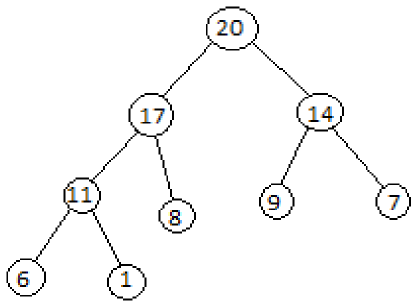


In-Complete Binary Tree

2. **Heap Property:** All nodes are either **greater than or equal to** or **less than or equal to** each of its children i.e, **Max-Heap**, **Min-Heap**.



Min-Heap



Max-Heap

How Heap Sort Works?

- Heap sort algorithm is divided into two basic parts:
  ---- Creating a Heap of the unsorted list/array.
  ---- Then a sorted array is created by repeatedly removing the largest/smallest element from the heap, and inserting it into the array. The heap is reconstructed after each removal.

Heap sorting Algorithm:

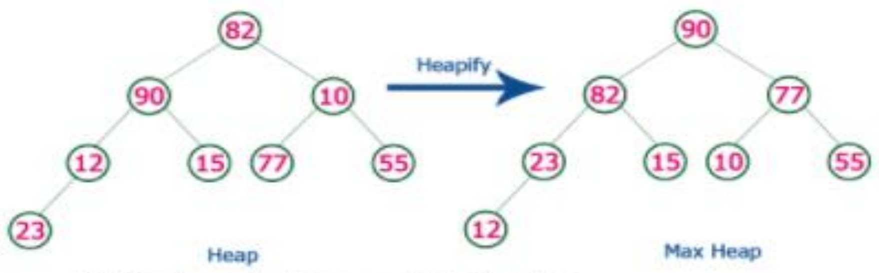
- The Heap sort algorithm to arrange a list of elements in ascending order is performed using following steps...

  Step 1 - Construct a **Binary Tree** with given list of Elements.
  Step 2 - Transform the Binary Tree into **Min/Max Heap.**
  Step 3 - Delete the root element from Min/Max Heap using **Heapify( )** method.

  Step 4 - Put the deleted element into the Sorted list.
  Step 5 - Repeat the same until Max/Min Heap becomes empty.
  Step 6 - Display the sorted list.

Heap Sort Example:

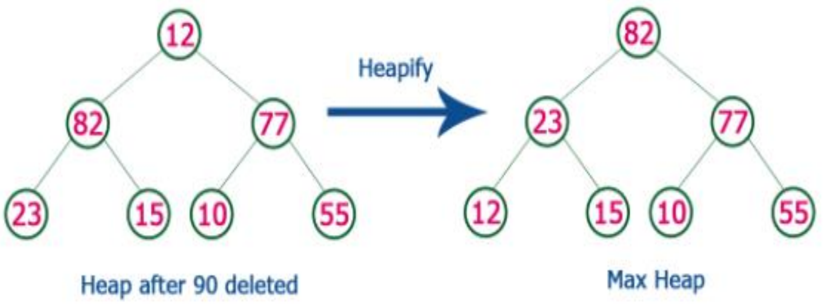
- Consider the following list of unsorted numbers which are to be sort using Heap Sort: 82, 90, 10, 12, 15, 77, 55, 23
  Step 1: Construct a Heap with given list of unsorted numbers and convert to Max Heap.



list of numbers after heap converted to Max Heap

90, 82, 77, 23, 15, 10, 55, 12

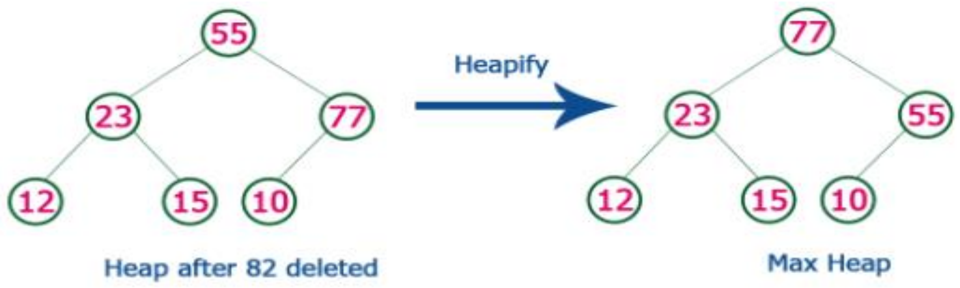
- Step 2: Delete root (90) from the Max heap. To delete root node it needs to be swapped with last node (12). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 90 with 12.

12, 82, 77, 23, 15, 10, 55, **90**

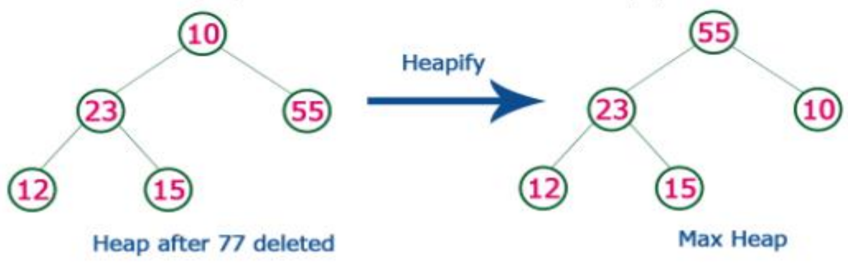
- Step 3: Delete root (82) from the Max heap. To delete root node it needs to be swapped with last node (55). After delete tree needs to be heapify to make it max heap.



list of numbers after swapping 82 with 55.

12, 55, 77, 23, 15, 10, **82, 90**

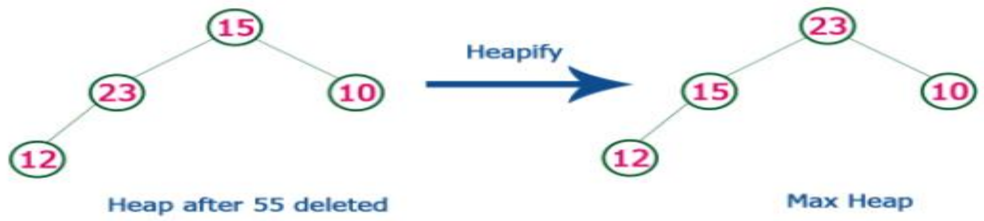
- Step4: Delete root (77) from the Max Heap. To delete root node it needs to be swapped with last node (10). After delete tree needs to be notify to make it Max Heap.



list of numbers after swapping 77 with 10.

12, 55, 10, 23, 15, **77, 82, 90**

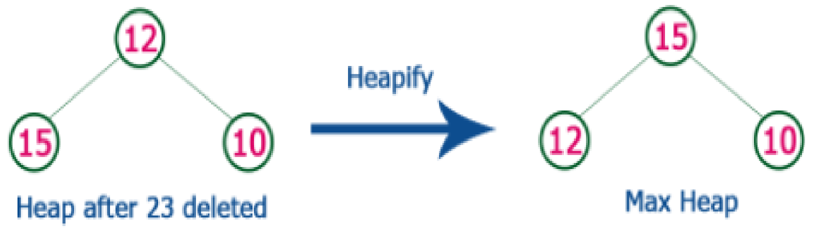
- Step 5: Delete root (55) from the Max Heap. To delete root node it needs to be swapped with last (15). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 55 with 15.

12, 15, 10, 23, 55, 77, 82, 90

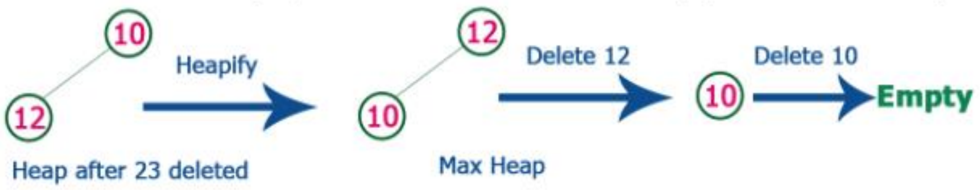
- Step 6: Delete root (23) from the Max Heap. To delete root node it needs to be swapped with last node(12). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 23 with 12.

12, 15, 10, 23, 55, 77, 82, 90

- Step 7: Delete root (15) from the Max Heap. To delete root node it need to be swapped with last node (10). After delete tree needs to be heapify to make it Max Heap.



list of numbers after Deleting 15, 12 & 10 from the Max Heap.

10, 12, 15, 23, 55, 77, 82, 90

Whenever Max Heap becomes Empty, the list get sorted in Ascending order

Complexity of the Heap Sort:

For best, worst and average case time complexity is O(n log n).

Advantages:

- The Heap sort algorithm is widely used because of its efficiency.
- its memory usage is minimal.

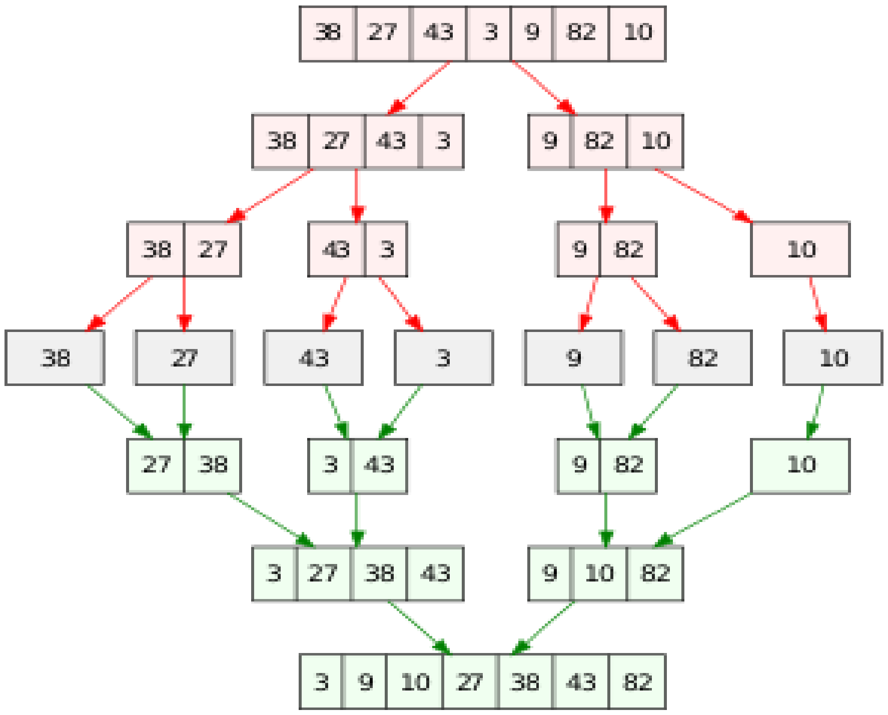
Disadvantages:

- Heap sort requires more space for sorting
- Quick sort is much more efficient than Heap in many cases

Merge sort:

- Merge sort is one of the most efficient sorting algorithms. It works on the principle of Divide and Conquer.

- Merge sort repeatedly breaks down a list into several sub-lists until each sub-list consists of a single element and merging those sub-lists in a manner that results into a sorted list.



Merge sort algorithm:

```
mergeSort(A,lb,ub)
        if (lb<ub)
          1. Find the middle point to divide the array into two halves:
                  middle mid = (lb+ub)/2
          2. Call mergeSort for first half:
                 Call mergeSort(A,lb, mid)
          3. Call mergeSort for second half:
                  Call mergeSort(A, mid+1, ub)
          4. Merge the two halves sorted in step 2 and 3:
                 Call merge(A, lb, mid, ub)
```

merge(A,lb,mid,ub):

```
{
        i=lb;
        j=mid+1;
        k=lb;
        while(i <= mid && j<= ub)
        {
                if(a[i]<=a[j])
                {
                        b[k]=a[i];
                        i++;   k++;
                }
                else{
```

```
                b[k]=a[j];
                j++;  k++;
                }
        }
        if(i>mid)
        {
                while(j<=ub)
                {
                        b[k]=a[j];
                        j++;
                        k++;
                }
        }
        else{
                while(i<=mid)
                {
                        b[k]=a[i];
                        i++;
                        k++;
                }
        }
}
```

Complexity of the Merge Sort:

For best, worst and average case time complexity is O(n log n).

Advantages

- It is quicker for larger lists because unlike insertion and bubble sort it does not go through the whole list several times.
- It has a consistent running time, carries out different bits with similar times in a stage**.**

Disadvantages

- Slower comparative to the other sort algorithms for smaller tasks
- Uses more memory space to store the sub elements of the initial split list.

Quick sort:

- Quick sort is a fast sorting algorithm used to sort a list of elements.
- The quick sort algorithm attempts to separate the list of elements into two parts and then sort each part recursively. That means it use divide and conquer strategy**.**
- In quick sort, the partition of the list is performed based on the element called *pivot*. Here pivot element is one of the elements in the list.
- The list is divided into two partitions such that "all elements to the left of pivot are smaller than the pivot and all elements to the right of pivot are greater than or equal to the pivot".

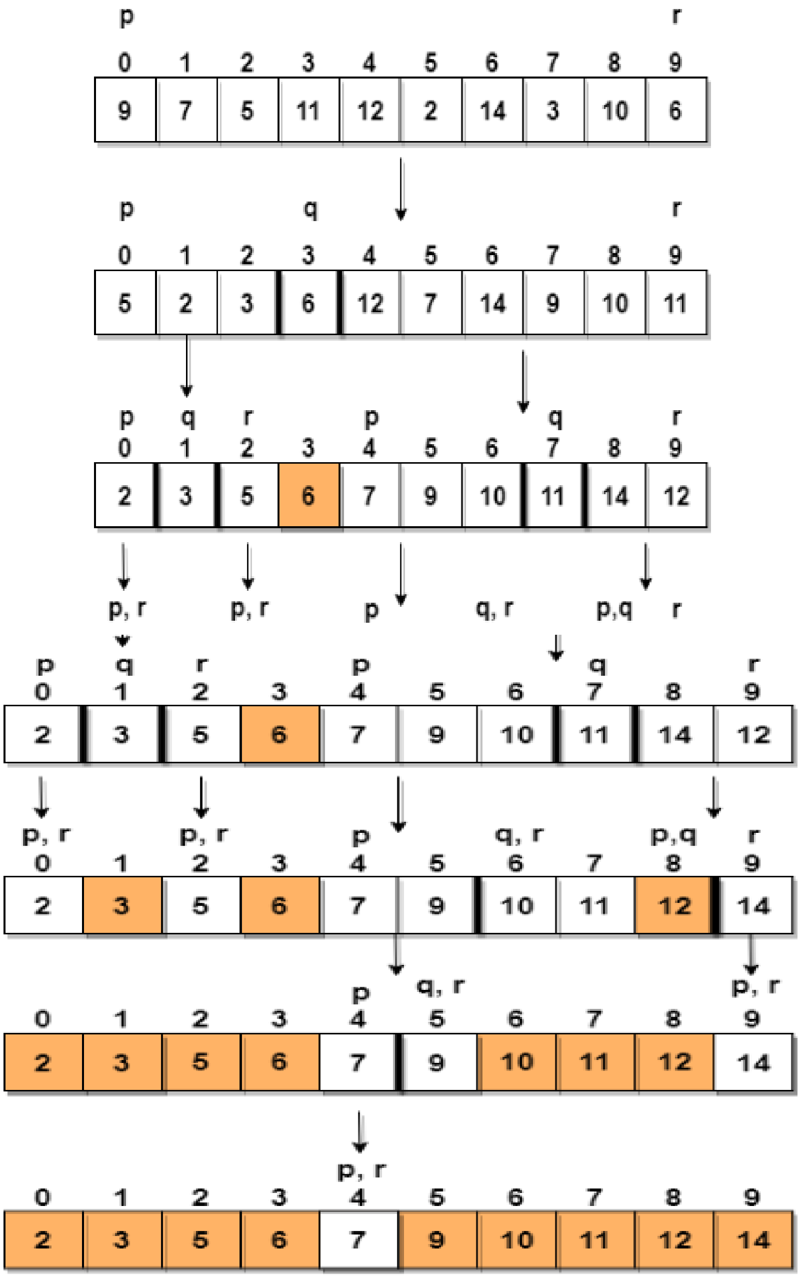
Quick sort algorithm:

- **In Quick sort algorithm, partitioning of the list is performed using following steps...**

Step 1 - Consider the first element of the list as pivot (i.e., Element at first position in the list).
Step 2 - Define two variables i and j. Set i and j to first and last elements of the list respectively.
Step 3 - Increment i until list[i] > pivot then stop.
Step 4 - Decrement j until list[j] < pivot then stop.
Step 5 - If i < j then exchange list[i] and list[j].
Step 6 - Repeat steps 3,4 & 5 until i >= j.
Step 7 - Exchange the pivot element with list[j] element.

Quick sort Example:

- Let us consider an array with 10 elements   9, 7, 5, 11, 12, 2, 14, 3, 10, 6



Time complexity:

- To sort an unsorted list with 'n' number of elements, we need to make ((n-1)+(n-2)+(n-3)+......+1) = (n (n-1))/2 number of comparisons in the worst case.

- If the list is already sorted, then it requires 'n' number of comparisons.
- Worst case: O(n^2)
- Best case: O (n log n)
- Average case: O (n log n)

Advantages

- Its average-case time complexity to sort an array of n elements is O(n log n).
- On the average it runs very fast, even faster than Merge Sort.
- It requires no additional memory.

Disadvantages

- Its running time can differ depending on the contents of the array.
- Quick sort's running time degrades if given an array that is almost sorted (or almost reverse sorted). Its worst-case running time, O(n 2) to sort an array of n elements, happens when given a sorted array.
- It is not stable.