Unit-2: Iterative Algorithms

Iterative algorithms use reputation of steps /operations. They use looping statements like for loop, while loop or do- while loop to repeat the same steps.

An iterative algorithm executes steps in iterations. It aims to find successive approximation in sequence to reach a solution. They are most commonly used in linear programs where large numbers of variables are involved.

Iterative algorithm to find GCD of two numbers

⇒ Greatest Common Divisor *(GCD)* of two positive numbers *a* and *b* is the largest integer that divides both *a* and *b*. It can be denoted as *gcd(a,b)*

- ***gcd(0, 0) = 0***.

Euclidian Algorithm to find GCD

⇒ The Euclidean algorithm can be based on the following theorem:

⇒ *For any integers a, b, with $a \geq b \geq 0$, gcd(a, b) = gcd(b, a mod b)*

Algorithm

Input: Two positive integers, a and b.

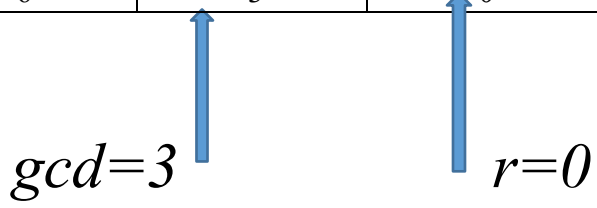
Output: The greatest common divisor, g, of a and b.

Internal computation

- If b =0, print a as gcd
- If a = 0 print b as gcd
- Divide a by b and get the remainder, r.
- If r=0, print b as the GCD of a and b
- Otherwise replace a by b and replace b by r then then repeat previous steps.

Example : To compute gcd(27,21)

Iteration	a	b	r=a%b
1	27	21	6
2	21	6	3
3	6	3	0



Pseudocode

```
PrintGcd(m, n){
        if(m==0)
                print (n) if(n==0)
                print (m)
        while(n!=0){
                r = m%n
                m= n
                n= r
        }
  print(n)
}
```

Analysis

According to pseudocode, the while loop executes at most n times. The Bih Oh notation for the time complexity of this algorithm is O(n).

The algorithm needs three variable m,n and r so the Space complexity is constant i.e. O(1)

Iterative algorithm for Fibonacci numbers

- Each number in the Fibonacci sequence is the sum of the two numbers that precede it.
- Fibonacci numbers are generated by setting $F_0 = 0$, $F_1 = 1$, and using the formula $F_n = F_{n-1} + F_{n-2}$ we get the rest.
- Thus the Sequence is : 0, 1, 1, 2, 3, 5, 8, 13, 21, and so on

Algorithm to find nth term of Fibonacci sequence

1. Start
2. Set F1 = 0 and F2 = 1
3. Read the term of Fibonacci sequence say n
4. Set i = 3
5. while (i<=n)
6. Fn = F1+F2
7. F1= F2
8. F2 = tmp
9. i ++
10. Print Fn as nth term of Fibonacci sequence
11. Stop

Example: To compute 10^{th} term of Fibonacci sequence

Iteration	F1	F2	Fn	i
1	0	1	0+1 =1	3
2	1	1	1+1 =2	4
3	1	2	1+2 =3	5
4	2	3	2+3 =5	6
5	3	5	3+5 =8	7
6	5	8	5+8 =13	8
7	8	13	8+13 =21	9
8	13	21	13+21 =34	10

10^{th} term of Fibonacci sequence = 34

Analysis

The while loop executes at most n-2 times. Hence the time complexity is T(n) = O(n)

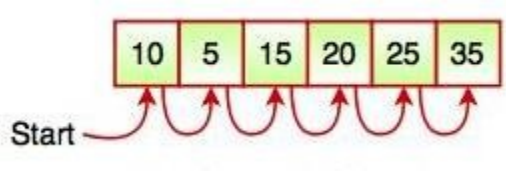
Space complexity = O(1)

Searching Algorithms

- Searching is the process of finding a given value in a list of values. It decides whether a search key is present in the data or not. It is the algorithmic process of finding a particular item in a collection of items.
- Any search is said to be successful or unsuccessful depending upon whether the element that is being searched is found or not.
- Some of the standard searching techniques that is being followed in the data structure are:
    - Linear Search or Sequential Search
    - Binary Search

Sequential Search

- Sequential search starts at the beginning of the list and checks every element of the list. It is a basic and simple searching algorithm.
- Sequential search compares the element with all the other elements given in the list. If the element is matched, it returns the value index, else it returns -1.



The above figure shows how sequential search works. It searches an element or value from an array till the desired element or value is not found. If we search the element 25, it will go step by step in a sequence order.

- It is used for unsorted and unordered small list of elements.

Pseudocode (If search is successful then index of the key is returned else -1 is returned)

```
LinearSearch(A,n, key)
{

       for (i = 0;  i<n; i++)
       {
         if (A[i] == key)

             return i;
       }
       return -1;
}
```

Analysis

Time Complexity:

For loop executes at most n times in worst case. So, time complexity is O(n)

Space Complexity:

n memory units are required to store n sized array. Similarly other variables i,n, and key takes 1 /1 memory units. So The space complexity is O(n).

Example: Consider the following array A of size 7 (n =7)

234	789	123	450	983	239	125

When key = 450

i	**A[i]== key A[i]==450**	**Returns**
0	A[0] is 234 234 == 350 => false	
1	A[1] is 789 789 ==350 => false	
2	A[2] is 123 234 ==350 => false	
3	A[3] is 450 450 == 450 => true	3
Search is successful and index of key (450) i.e. 3 is returned		

When key = 500

i	**A[i]== key** **A[i]==500**	**Returns**
0	A[0] is 234 234 == 500 => false	
1	A[1] is 789 789 ==500 => false	
2	A[2] is 123 123 ==500 => false	
3	A[3] is 450 450 == 500 => false	
4	A[4] is 983 983 == 500 => false	
5	A[5] is 239 239 == 500 => false	
6	A[5] is 125 125 == 500 => false	
7		-1
Search is not successful and -1 is returned		

Sorting Algorithms

Sorting

Sorting is the process of arranging data in particular order (ascending or descending).

Sorting is among the most basic problems in algorithm design. We are given a sequence of items, each associated with a given key value. The problem is to permute the items so that they are in increasing (or decreasing) order by key. Sorting is important because it is often the first step in more complex algorithms. Sorting algorithms are usually divided into two classes, internal sorting algorithms, which assume that data is stored in an array in main memory, and external sorting algorithm, which assume that data is stored on disk or some other device that is best accessed sequentially. We will only consider internal sorting. Sorting algorithms often have additional properties that are of interest, depending on the application. Here are two important properties.

In-place: The algorithm uses no additional array storage, and hence (other than perhaps the system's recursion stack) it is possible to sort very large lists without the need to allocate additional working storage.

Stable: A sorting algorithm is stable if two elements that are equal remain in the same relative position after sorting is completed. This is of interest, since in some sorting applications you sort first on one key and then on another. It is nice to know that two items that are equal on the second key, remain sorted on the first key.

Bubble Sort

The bubble sort algorithm Compare adjacent elements. If the first is greater than the second, swap them. This is done for every pair of adjacent elements starting from first two elements to last two elements. At the end of pass1 greatest element takes its proper place. The whole process is repeated except for the last one so that at each pass the comparisons become fewer

- The "bubble" sort is called so because the list elements with greater value than their surrounding elements "bubble" towards the end of the list. For example, after first pass, the largest element is bubbled towards the right most position. After second pass, the second largest element is bubbled towards the second last position in the list and so on.

  Algorithm

```
BubbleSort(A, n)
{
        for(i = 0; i <n-1; i++)
        {
        for(j = 0; j < n-i-1; j++)
        {
                if(A[j] > A[j+1])
                {
                        temp = A[j];
                        A[j] = A[j+1];
                        A[j+1] = temp;
                }
        }
        }
}
```

- The array with n elements is sorted by using n-1 pass of bubble sort algorithm.

Time Complexity:

Inner loop executes for (n-1) times when i=0, (n-2) times when i=1 and so on:

Time complexity = (n-1) + (n-2) + (n-3) + ...+2 +1

= $O(n^2)$

There is no best-case linear time complexity for this algorithm.

Space Complexity:

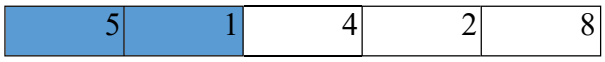
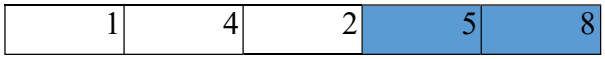
Since no extra space besides 3 variables is needed for sorting

Space complexity = O(1)

Example: Sorting the array A = {5,1,4,2,8} using Bubble sort.

5	1	4	2	8

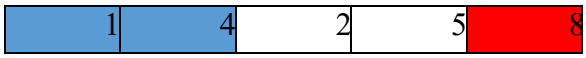
Pass 1 i = 0 , j = 0 to n-0-2 (since j < n-0-1) i.e. 3]

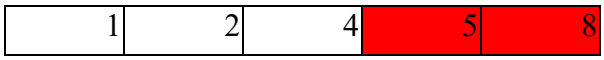
j	A[j]>A[j+1]	swap	Array
0	A[0]>A[1] 5>1	yes	
1	A[1]>A[2] 5>4	yes	
2	A[2]>A[3] 5>2	yes	
3	A[3]>A[4] 5>8	no	

After Pass 1:

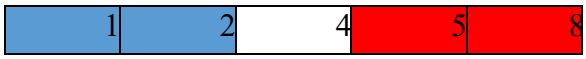
1	4	2	5	8

Pass 2: [i = 1, j = 0 to n-1-2 i.e. 2]

j	A[j]>A[j+1]	swap	Array
0	A[0]>A[1] 1>4	no	
1	A[1]>A[2] 4>2	yes	
2	A[2]>A[3] 4>5	no	

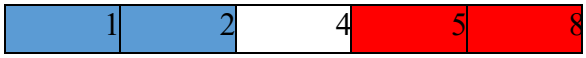
After Pass 2: 

Pass 3: [i = 2, j = 0 to n-2-2 i.e. 1]

j	A[j]>A[j+1]	swap	Array
0	A[0]>A[1] 1>2	no	
1	A[1]>A[2] 2>4	no	

After Pass 3: 

Pass 4: [i = 3, j = 0 to n-3-2 i.e. 0]

j	A[j]>A[j+1]	swap	Array
0	A[0]>A[1] 1>2	no	

After Pass 4: the fully sorted array is obtained.

Selection Sort

Idea: Find the least (or greatest) value in the array, swap it into the leftmost (or rightmost) component (where it belongs), and then forget the leftmost component. Do this repeatedly.

- This algorithm gets its name from the way it iterates through the array: it selects the current smallest element, and swaps it into place.
- In selection sort, the smallest value among the unsorted elements of the array is selected in every pass and inserted to its appropriate position into the array.
- First, find the smallest element of the array and place it on the first position. Then, find the second smallest element of the array and place it on the second position. The process continues until we get the sorted array.
- The array with n elements is sorted by using n-1 pass of selection sort algorithm.

Algorithm:

```
SelectionSort(A)
{
        for( i = 0;i < n-1 ;i++)
        {
                min = i;
                for ( j = i + 1;j < n ;j++)
                {
                        if (A[j] < A[min])
                          min = j;
                }
                if( min!=i)
                        swap(A[i],A[min]);
        }
}
```

Time Complexity:

Inner loop executes for (n-1) times when i=0, (n-2) times when i=1 and so on:

Time complexity = (n-1) + (n-2) + (n-3) + ...+2 +1

= $O(n^2)$

There is no best-case linear time complexity for this algorithm, but number of swap operations is reduced greatly.

Space Complexity:

Since no extra space besides 5 variables is needed for sorting

Space complexity = O(1)

Example:

A[ ] = [64 25 12 22 11]

64	25	12	22	11

Pass 1: For i = 0

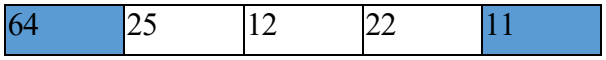
min = i = 0

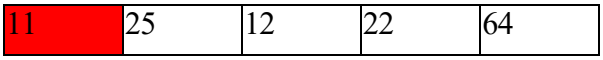
for j = i+1 i.e. 0+1 = 1 to 4 (j<n i.e 5)

64	25	12	22	11

j	**A[j]<A[min]**	**min = j**
1	A[1]<A[0] 25<64	min = 1
2	A[2]<A[1] 12<25	min = 2
3	A[3]<A[2] 22<12	no
4	A[4]<A[2] 11< 12	min = 4

Herer, min!= i, so swap (A[i], A[min]) i.e. swap (A[0], A[4])

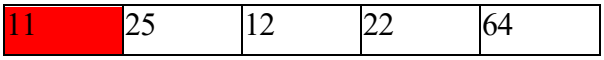




Pass 2: For i = 1

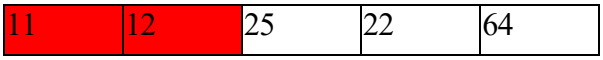
min = i =1

for j = i+1 i.e. 1+1 = 2 to 4 (j<n i.e 5)



j	**A[j]<A[min]**	**min=j**
2	A[2]<A[1] 12<25	min = 2
3	A[3]<A[2] 22<12	no
4	A[4]<A[2] 64<12	no

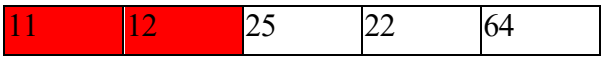
Here, min!= i , so swap (A[i], A[min]) i.e. swap (A[1], A[2])



Pass 3: For i = 2

min = i =2

for j = i+1 i.e. 2+1 = 3 to 4 (j<n i.e 5)



j	A[j]<A[min]	min=j
3	A[3]<A[2] 22<25	min=3
4	A[4]<A[3] 64<22	no

Here, min!= i , so swap (A[i], A[min]) i.e. swap (A[2], A[3])

Pass 4: For i = 3

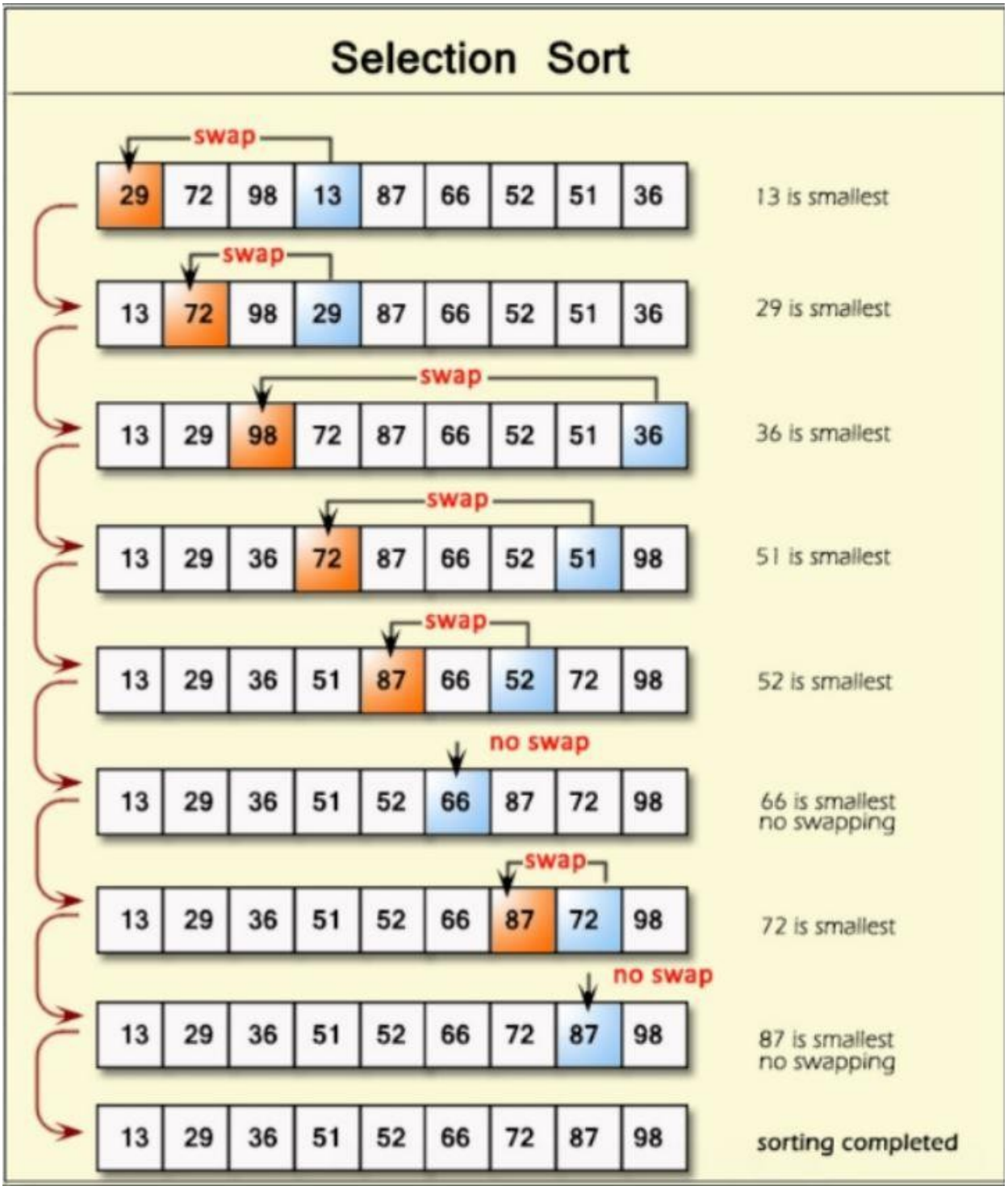
min = i =3

for j = i+1 i.e. 4+1 = 4 to 4 (j<n i.e 5)

j	A[j]<A[min]	min=j
4	A[4]<A[3] 64<25	no

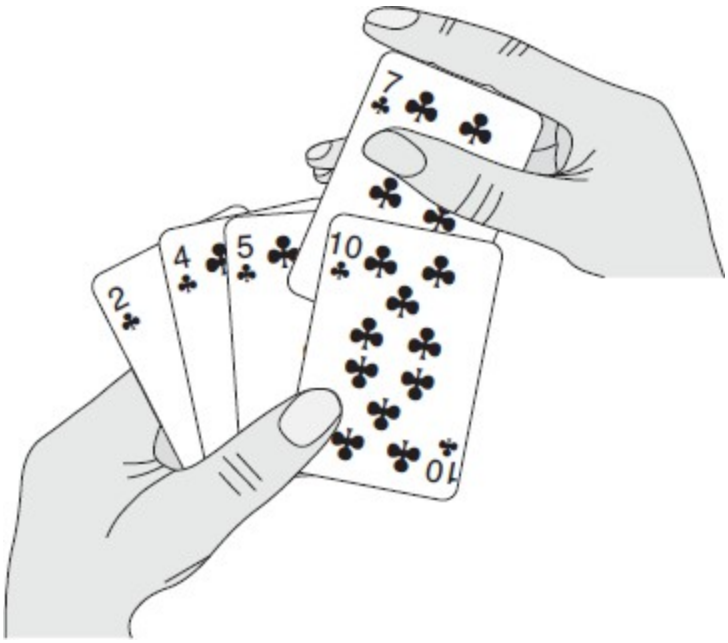
Here, min= i , so no swap and the array is now sorted

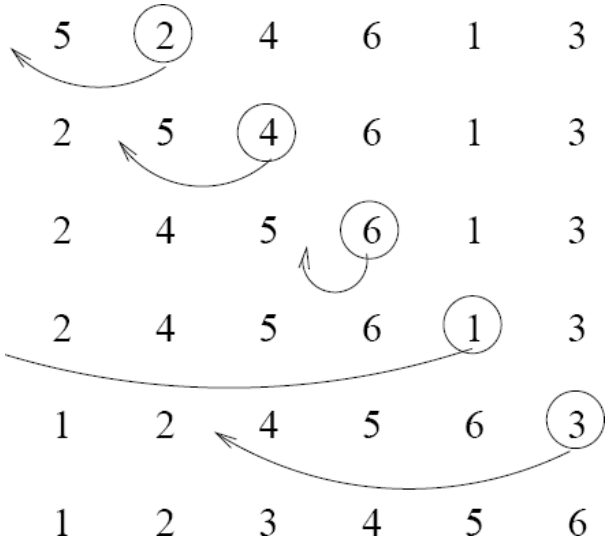
Another example:



Insertion Sort

Idea: like sorting a hand of playing cards start with an empty left hand and the cards facing down on the table. Remove one card at a time from the table, and insert it into the correct position in the left hand. Compare it with each of the cards already in the hand, from right to left. The cards held in the left hand are sorted





- It iterates the input elements by growing the sorted array at each iteration. It compares the current element with the largest value in the sorted array. If the current element is greater, then it leaves the element in its place and moves on to the next element else it finds its correct position in the sorted array and moves it to that position. This is done by shifting all the elements, which are larger than the current element, in the sorted array to one position ahead (this for ascending order)

Algorithm:

```
InsertionSort(A)
        {
                for (i=1;i<n;i++)
                {
                        temp= A[ i]
                        j = i-1;
                         while( j>=0 && A[j] >temp)
                        {
                                A[j + 1] = A[j]
                                 j- -;
                        }
                       A[j + 1] = temp
                }
        }
```

- 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.
- Insertion sort algorithm arranges a list of elements in a particular order. In insertion sort algorithm, every iteration moves an element from unsorted portion to sorted portion until all the elements are sorted in the list.

Example: A= {17,25,31,13,2}

17	25	31	13	2

Pass1: i=1, temp= A[i] = A[1] = 25

17	25	31	13	2

j = 1-1 i.e. 0 to 0 (j>=0)

j	**A[j]>temp**	**A[j+1] = A[j]**
0	A[0]>25 17>25	no

A[0+1]=temp

A[1] = 25

17	25	31	13	2

Pass2: i=2, temp = A[2] = 31

j = 2-1 i.e. 1 to 0 (j>=0)

j	**A[j]>temp**	**A[j+1] = A[j]**
1	A[1]>31 25>31	no

A[1+1]=temp

A[2] = 31

17	25	31	13	2

Pass3: i=3, key = A[3] = 13

j = 3-1 i.e . 2 to 0 (j>=0)

j	**A[j]>key**	**A[j+1] = A[j]**
2	A[2]>13 31>13	A[3]= A[2] A[3] = 31
1	A[1]>13 25>13	A[2]=A[1] A[2]=25
0	A[0]>13 17>13	A[1]=A[0] A[1]=17
-1	-	-

j= -1

A[-1+1]=temp

A[0] = 13

13	17	25	31	2

Pass4: i=4, key = A[4] = 2

j = 4-1 i.e . 3 to 0 (j>=0)

j	**A[j]>key**	**A[j+1] = A[j]**
3	A[3]>2 31>2	A[4]= A[3] A[4] = 31
2	A[2]>2 25>2	A[3]=A[2] A[3]=25
1	A[1]>2 17>2	A[2]=A[1] A[2]=17
0	A[0]>2 13>2	A[1]=A[0] A[1]=13
-1	-	-

j= -1

A[-1+1]=temp

A[0] = 2

2	13	17	25	31

Now the array is sorted.

Time Complexity:

Worst Case Analysis:

Array elements are in reverse sorted order

Inner loop executes for 1 times when i=1, 2 times when i=2... and n-1 times when i=n-1:

Time complexity = 1 + 2 + 3 + ... + (n-2) + (n-1)

= $O(n^2)$

Best case Analysis:

Array elements are already sorted

Inner loop executes for 1 times when i=1, 1 times when i=2... and 1 times when i=n-1:

Time complexity = 1 + 1+ 1 + ... +1 +1

= O(n)

Space Complexity:

Since no extra space besides 4 variables is needed for sorting

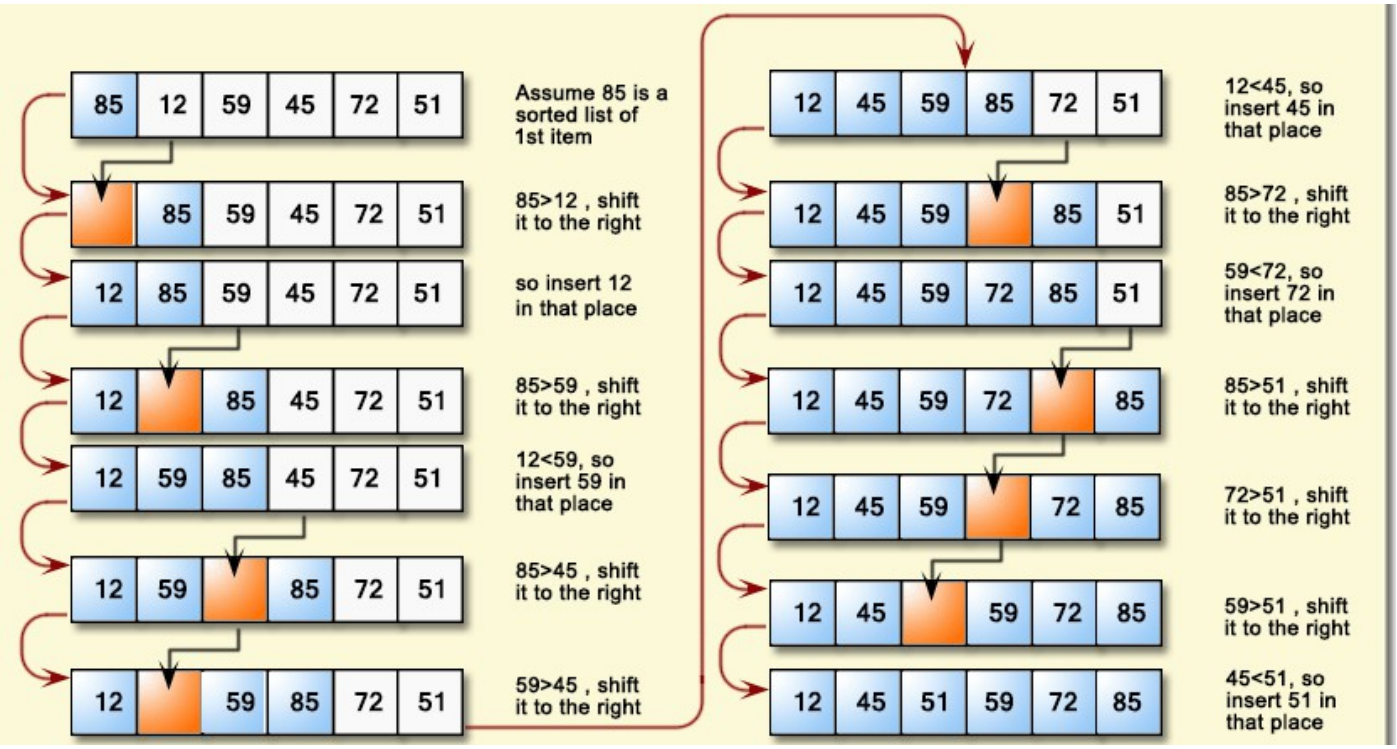
Space complexity = O(1)

Another Example:

Consider the following elements are to be sorted in ascending order

85, 12, 59, 45, 72, 51

The process of insertion sort is illustrated as follow:


85 12 59 45 72 51
Assume 85 is a sorted list of 1st item
85 59 45 72 51
85>12 , shift it to the right
12 85 59 45 72 51
so insert 12 in that place
12 85 45 72 51
85>59 , shift it to the right
12 59 85 45 72 51
12<59, so insert 59 in that place
12 59 85 72 51
85>45 , shift it to the right
12 59 85 72 51
59>45 , shift it to the right
12 45 59 85 72 51
12<45, so insert 45 in that place
12 45 59 85 51
85>72 , shift it to the right
12 45 59 72 85 51
59<72, so insert 72 in that place
12 45 59 72 85
85>51 , shift it to the right
12 45 59 72 85
72>51 , shift it to the right
12 45 59 72 85
59>51 , shift it to the right
12 45 51 59 72 85
45<51, so insert 51 in that place