UNIT-2
Iterative Algorithms

⊛. Basic Algorithms:-

1) Algorithm for GCD:

1. Start
2. Read any two numbers say m and n.
3. If n==0, return the value of m as the answer and stop.
4. If m==0, return the value of n as the answer and stop.
5. Divide m by n and assign the value of remainder to r.
6. Assign the value of n to m and value of r to n.
7. Go to step 3.

Pseudo code for GCD

```
GCD(m,n)
{ if(m==0)
        Print "n as GCD"
  else if(n==0)
        Print "m as GCD"
  else
    { while(n!=0)
        { r=m%n;
          m=n;
          n=r;
        }
      Print "m as GCD"
    }
}
```

Analysis:

Since while loop executes at most n times so, time complexity = O(n).
Since space complexity is constant (i.e, 3), so space complexity = O(1).

Example: Find the GCD of 24 and 9, by Euclid's algorithm.

Solution:

$m = 24$
$n = 9$
$m \neq 0$
$n \neq 0$

since atmost 3 variables used to store data i.e, constant for constant always O(1).

Iteration 1: $r = 24 \% 9 = 6$

$m = n = 9$

$n = r = 6$

Iteration 2:

$r = 9 \% 6 = 3$

$m = n = 6$

$n = r = 3$

Iteration 3:

$r = 6 \% 3 = 0$

$m = n = 3$

$n = r = 0$

⇒ Hence, GCD of 24 and 9 is 3.

2) Algorithm for Fibonacci Number:

1. Start
2. Set first = 0, second = 1
3. Read term of Fibonacci number say it be n.
4. Set i = 3
5. While (i <= n)
   - Set temp = first + second
   - Set first = second
   - Set second = temp
   - Increment i by 1 as, i++
6. Print temp as required Fibonacci number
7. Stop.

Pseudo code for Fibonacci number

```
Fibonacci(n)
 {
    first = 0;
    second = 1;
    i = 3;
      while(i<=n)
         { temp = first + second;
           first = second;
           second = temp;
           i++;
           Print " temp";
         }
 }
```

Analysis:

Since while loop executes at most n-2 times, so time complexity = O(n).
Since the space complexity is constant (i.e, 4), so, space complexity = O(1).

Example/Tracing: Find 5th Fibonacci number.

first = 0
second = 1
i = 3

Iteration 1:

temp = 0+1 = 1
first = 1
second = 1
i = 3+1 = 4

Iteration 2:

temp = 1+1 = 2
first = 1
second = 2
i = 4+1 = 5

Iteration 3:

temp = 1+2 = 3
first = 2
second = 3
i = 5+1 = 6

⇒ Hence, 3 is the 5th number of Fibonacci series.

⊛ Searching Algorithms:-

⊛ Sequential Search:-

(also called linear search)

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

Pseudo code

```
LinearSearch(A, n, key)
{  flag=0;
   for(i=0; i<n; i++)
     {  if(A[i]==key)
            flag=1;
     }
   if(flag==1)
        Print "Search successful"
   else
        Print "Search un-successful"
```

A is an already existing array in which we are going to search user input key. n is the size to array used during for loop.

Analysis:-

Since for loop executes at most n times so time complexity = O(n).
Since array A takes n memory refrences so, space complexity = O(n).

⊛. Sorting Algorithms:

1) Bubble Sort:

Bubble sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order. In bubble sort, each element of the array is compared with its adjacent element. The algorithm processes the list in passes. A list with n elements requires n-1 passes for sorting. Each pass consists of comparing each element in the array with its adjcent (successor) element (a[i] with a[i+1] and interchanging two elements if they are not in proper order.

Algorithm:

1. Start
2. For the first iteration, compare all the elements (n).
   For the subsequent runs, compare (n-1) (n-2) and so on.
3. Compare each element with its right side neighbour.
4. Swap the smaller element to the left.
5. Keep repeating steps 1 to 3 until whole list is covered.
6. Stop.

Pseudo code

```
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;
      }
    }
  }
}
```

Analysis:

→ The best time complexity for Bubble Sort is $O(n)$. The average and worst time complexity is $O(n^2)$.

→ The space complexity of Bubble Sort is $O(n)$, because array A takes n memory refrences.

Example/Tracing:- Sort the following data items using Bubble sort.

A[] = {25, 57, 48, 37, 12, 92, 86, 33}.

Solution:

Array Position	0	1	2	3	4	5	6	7
Initial state	25	57	48	37	12	92	86	33
Pass 1	25	48	37	12	57	86	33	92
Pass 2	25	37	12	48	57	33	86	92
Pass 3	25	12	37	48	33	57	86	92
Pass 4	12	25	37	33	48	57	86	92
Pass 5	12	25	33	37	48	57	86	92
Pass 6	12	25	33	37	48	57	86	92
Pass 7	12	25	33	37	48	57	86	92
Pass 8	12	25	33	37	48	57	86	92

Even if the list is already sorted we write upto n passes, since pass start from 1.

But if we start pass from 0 then upto n-1

Hence, sorted array is A[] = {12, 25, 33, 37, 48, 57, 86, 92}.

2) Selection Sort:

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. This method includes finding 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 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:

1. Start.
2. Consider the first element to be sorted and the rest to be unsorted.
3. Assume the first element to be the smallest element.
4. Check if the first element is smaller than each of the other elements:
   - → If yes, do nothing
   - → If no, choose the other smaller element as minimum and repeat step 3.
5. After completion of one iteration through the list, swap the smallest element with the first element of the list.
6. Now consider the second element in the list to be smallest and so on till all the elements in the list are covered.
7. Stop.

Pseudo code:-

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

Analysis:-

→ If felt hard simply remember time complexity as $O(n^2)$

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

Time complexity $= (n-1)+(n-2)+(n-3)+\ldots+2+1$

$= O(n^2)$.

→ Since Array A takes n memory refrences so,

Space complexity $= O(n)$.

Example/Tracing:- Sort the following data items by using selection sort.

A[ ] = [25, 57, 48, 37, 12, 92, 86, 33].

Solution:

Array Position	0	1	2	3	4	5	6	7
Initial state	25	57	48	37	12	92	86	33
Pass 1	12	57	48	37	25	92	86	33
Pass 2	12	25	48	37	57	92	86	33
Pass 3	12	25	33	37	57	92	86	48
Pass 4	12	25	33	37	57	92	86	48
Pass 5	12	25	33	37	48	92	86	57
Pass 6	12	25	33	37	48	57	86	92
Pass 7	12	25	33	37	48	57	86	92
Pass 8	12	25	33	37	48	57	86	92

3) Insertion Sort:

In this method initially A[0] is the only element on the sorted set. In pass 1, A[1] is placed at it's proper index in the array. In pass 2, A[2] at it's proper index in the array. Likewise, in pass n-1, A[n-1] is placed at its proper index into the array.

To insert an element A[k] to its proper index, we must compare it with all other elements i.e, A[k-1], A[k-2], and so on until we find an element A[j] such that, $A[j] <= A[k]$. All the elements from A[k-1] to A[j] need to be shifted and A[k] will be moved to A[j+1].

Algorithm:

1. Start
2. Consider the first element to be sorted and rest to be unsorted.
3. Compare with the second element:
   → If the second element < the first element, insert the element in the correct position of the sorted portion.

→ else, leave it as it is.

4. Repeat steps 2 and 3 until all elements are sorted.
5. Stop.

Pseudo code:

```
Insertion(A,n)
{ for i=1 to n
  { temp=A[i]
    j=i-1
    while (j>=0 && A[j]>temp)
    { A[j+1]=A[j]
      j=j-1
    }
  }
  A[j+1]=temp
}
```

Analysis:

→ Time complexity of insertion sort is $O(n^2)$ similar to bubble sort but it is considered better than bubble sort.

→ Space complexity is $O(n)$ since array A takes n memory refrences.

Example/Tracing: Sort the following data items by using insertion sort.

A[] = {25, 57, 48, 37, 12, 92, 86, 33}

Solution:

Array Position	0	1	2	3	4	5	6	7
Initial state	25	57	48	37	12	92	86	33
Pass1	25	57	48	37	12	92	86	33
Pass2	25	57	48	37	12	92	86	33
Pass3	25	48	57	37	12	92	86	33
Pass4	25	37	48	57	12	92	86	33
Pass5	12	25	37	48	57	92	86	33
Pass6	12	25	37	48	57	92	86	33
Pass7	12	25	37	48	57	86	92	33
Pass8	12	25	33	37	48	57	86	92

n pass मा n elements sorted हुनु पर्यो तर बाहिरको element को मतलब गर्दा for.eg. for pass 4 first 4 elements must be in sorted form, elements behind first 4 elements remain as they were.