Week-3

Algorithm design strategies: Brute force – Bubble sort, Selection Sort, Linear Search. Decrease and conquer - Insertion Sort

Algorithm design strategies:

There are six different techniques to design an algorithm, they are as follows:

1) Brute Force
2) Divide and Conquer
3) Decrease and Conquer
4) Transform and Conquer
5) Dynamic Programming
6) Greedy Approach

Brute Force:

- Brute force is a straight-forward approach fir solving the problem. It is also called as "Just do it" approach.
- It is a simplest way to explore the space of solutions. This will go through all possible solutions extensively.
- Brute force method solves the problem with acceptable speed and large class of input.
- Examples (Applications)
  - Bubble Sort
  - Selection Sort
  - Computing n! : The n! Can be computed as 1x2x3x......x n
  - Multiplication of two matrices

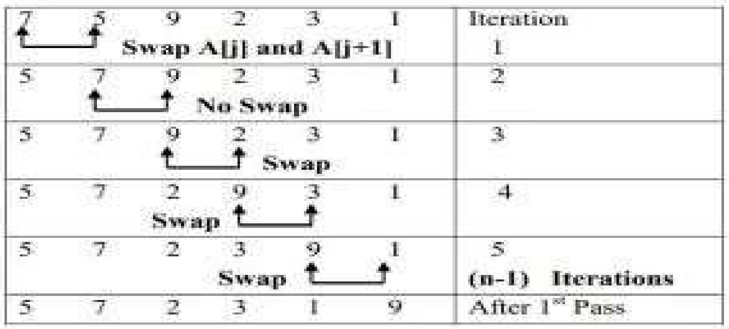
Bubble Sort:

- Bubble sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.
- The bubble sort algorithm is also known by the name sinking sort
- This algorithm works as follows
  - Step1: Compare each pair of adjacent elements from the beginning of the list
  - Step2: Swap those elements if the elements are not in proper order
  - Step3: Repeat the steps 1 and 2 until all the items are processed.

How Bubble sort Works??

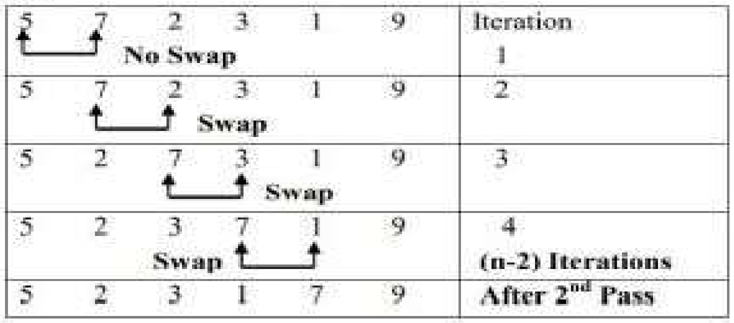
- The Bubble sort algorithm requires (n-1) number of steps/ passes to sort n items.
- In every pass/step the adjacent items are compared with each other and they are exchanged if they are in wrong order.
- Working of bubble sort algorithm to sort the given list of items in ascending order is as shown below.

- **Pass:1**



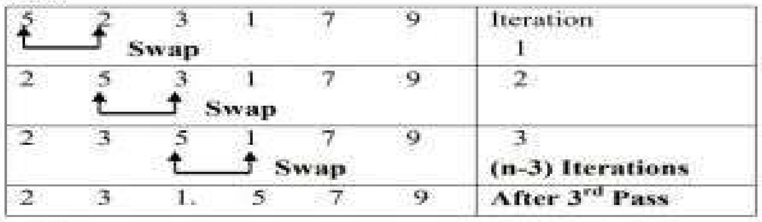
7	5	9	2	3	1	Swap A[j] and A[j+1]	Iteration 1
5	7	9	2	3	1	No Swap	2
5	7	9	2	3	1	Swap	3
5	7	2	9	3	1	Swap	4
5	7	2	3	9	1	Swap	5 (n-1) Iterations
5	7	2	3	1	9		After 1st Pass

- Pass:2



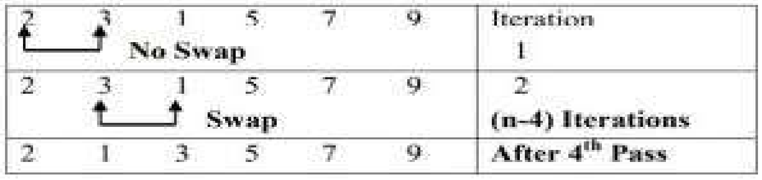
5	7	2	3	1	9	No Swap	Iteration 1
5	7	2	3	1	9	Swap	2
5	2	7	3	1	9	Swap	3
5	2	3	7	1	9	Swap	4 (n-2) Iterations
5	2	3	1	7	9		After 2nd Pass

- Pass:3



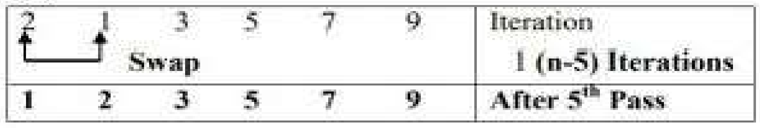
5	2	3	1	7	9	Swap	Iteration 1
2	5	3	1	7	9	Swap	2
2	3	5	1	7	9	Swap	3 (n-3) Iterations
2	3	1.	5	7	9		After 3rd Pass

- Pass:4



2	3	1	5	7	9	No Swap	Iteration 1
2	3	1	5	7	9	Swap	2 (n-4) Iterations
2	1	3	5	7	9		After 4th Pass

- Pass:5



2	1	3	5	7	9	Swap	Iteration 1 (n-5) Iterations
1	2	3	5	7	9		After 5th Pass

Implementation of Bubble sort in Python:

```
# function definition of Bubble sort
def Bubble_sort(Array):
    n=len(Array)
    for i in range(n-1):
        swapped=False
        for j in range(n-1-i):
            if Array[j]<Array[j+1]:
                 Array[j],Array[j+1]=Array[j+1], Array[j]
                 Swapped True
        if swapped== False:
             break
     return Array
# Code to invoke the function Bubble sort
Array=[ ]   # Create an empty list
n=int(input("How many elements???")) # n stores the number of items to be stored in the list
# Now populate the list with user inputs
for i in range(n):
          Array.append(int(input("Enter %d item:" %i)))
# Display the list before sorting it
print("Before sorting the list items are")
print(Array)
# Now call function Bubble_sort
Array=Bubble_sort(Array)
# Display the list items after sorting
print("Before sorting the list items are")
print(Array)
```

Time Complexities: (Running time of Bubble sort)

 The time complexity of the bubble sort is O(n^2)

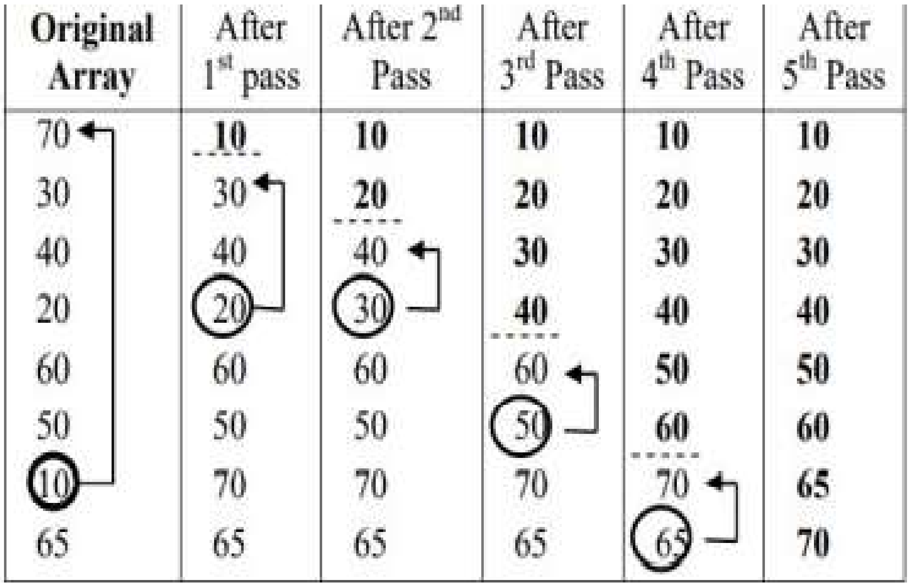
 Worst Case Complexity: O(n^2) : If we want to sort in ascending order and the array is in descending order then the worst case occurs.

 Best Case Complexity: Ω(n) : If the array is already sorted, then there is no need for sorting. But alteast one pass/step is required to check whether the array is sorted

 Average Case Complexity: ϕ(n^2) : It occurs when the elements of the array are in random order.

Selection Sort:

- Selection sort algorithm sorts array elements by repeatedly finding the smallest element from the unsorted array and putting at the beginning. This process will continue until the entire array is sorted.
- Selection sort means selecting the smallest element in case of ascending order and largest element in case of descending order
- How does the selection sort works:
  Let us take an example array consist of the following element
  70,30,40,20,60,50,10,65



Original Array	After 1st pass	After 2nd Pass	After 3rd Pass	After 4th Pass	After 5th Pass
70	10	10	10	10	10
30	30	20	20	20	20
40	40	40	30	30	30
20	20	30	40	40	40
60	60	60	60	50	50
50	50	50	50	60	60
10	70	70	70	70	65
65	65	65	65	65	70

Implementation of Selection sort in PYTHON:

```
def selectionsort(array):
    n=len(array)
    for i in range(n-1):
        min=i
        for j in range(i+1,n):
            if array[j]<array[min]:
                min=j
            array[i],array[min]=array[min],array[i]
        return array
array=[]
n=int(input("How many elements??"))
for i in range(n):
    array.append(int(input("Enter %d number:" %i)))
print(f"before swapping:{array}")
array=selectionsort(array)
print(f'After swapping:{array}')
```

Time Complexities:

The time complexity of the selection sort is O(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: Ω(n^2) :It occurs when the array is already sorted. Though the array is already sorted it requires the same time.
- Average Case Complexity: ϕ(n^2) :It occurs when the elements of the array are in random order.

Searching:

- **Searching is a process to find whether the particular data is present within the list of data or not.**
- **The data to be searched is known as the key element**
- **During searching, if the key element is found then the search is successful**
- **Otherwise search is said to be unsuccessful**
- **The following are the different searching algorithms available**
  1) **Linear search**
  2) **Binary Search**
  3) **Interpolation Search**

Sequential Search or Linear Search:

- Sequential search is the most natural searching method and it is simplest method.
- In this method, the searching process starts from the beginning of an array and continues until the given element is found or until the end of the array
- In this technique the key element to be searched is compared with the each data element of the list in sequential manner or in linear fashion.

Implementation of linear search in Python:

```
#Function Definition
def linearsearch(a, key):
    n = len(a)
    for i in range(n):
        if a[i] == key:
            return i;
    return -1
# code to call the function linearsearch()
a = [13,24,35,46,57,68,79]
print(f "the array elements are:{a}”)
key = int(input("enter the key element to search:"))
result = linearsearch(a,key)
if result == -1:
    print("Search UnSuccessful")
else:
    print("Search Successful key found at  %d location:" %result)
```

Time Complexities:

Linear search algorithm time taken by the algorithm depends on the position of the key element to be searched.

Best Case Time complexity: Ω(1): The best case occurs when the key to be searched is present at the very first position.

Worst case : O(n): The Worst case occurs when the key to be searched might be present at the last position or might not be present.

Average Case: ϕ(n): Here the random or typical inputs are used to compute the average case time

Decrease-and-Conquer:

- The decrease and conquer is a method of solving a problem by changing the problem size from n to smaller size of n-1, n/2 etc.
- In other words, change the problem from larger instance into smaller instance.
- Conquer (or solve) the problem of smaller size.
- Convert the solution of smaller size problem into a solution for larger size problem.
- Using decrease and conquer technique, we can solve a given problem using top-down technique (using recursion) or bottom up technique (using iterative procedure).
- There are three major variations of decrease-and-conquer:
    1) Decrease by a constant
    2) Decrease by a constant factor
    3) Variable size decrease

Decrease by a Constant: In this variation, the size of an instance is reduced by the same constant on each iteration of the algorithm. Typically, this constant is equal to one , although other constant size reductions do happen occasionally.

Below are example problems :

- Insertion sort
- Graph search algorithms:DFS,BFS
- Topological sorting

Decrease by a Constant factor: This technique suggests reducing a problem instance by the same constant factor on each iteration of the algorithm. In most applications, this constant factor is equal to two.

Variable-Size-Decrease: In this variation, the size-reduction pattern varies from one iteration

of an algorithm to another.

Below are example problems :

- Computing median and selection problem.
- Interpolation Search
- Euclid's algorithm to find the GCD of two positive integers

Insertion Sort:

- It is a simple Sorting algorithm which sorts the array by shifting elements one by one.
- Insertion sort works as follows
  - Set the marker for the sorted section after the first element in an array.
  - Select the first element from unsorted section
  - Compare the sorted section element with unsorted section element and swap those elements if the condition is true
  - Move the marker to the right position of sorted section.
  - Repeat the steps 3 and 4 until the unsorted section is empty

Implementation:

```
def insertionsort(array):
    n=len(array)
    for step in range(1,n):
        item=array[step]
        j=step-1
        while j>=0 and item<array[j]:
            array[j+1]=array[j]
            j=j-1
        array[j+1]=item
    return array

array=[]
n=int(input("How many elements??"))
for i in range(n):
    array.append(int(input("Enter %d number:" %i)))
```

```
print(f"before swapping:{array}")
array=insertionsort(array)
print(f'After swapping:{array}')
```

Time Complexities:

The time complexity of the Insertion sort is O($n^{2)}$

• 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) : If the array is already sorted, then there is no need for sorting.

• Average Case Complexity: O(n^2) : It occurs when the elements of the array are in random order.