

Week-3

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

Algorithm design strategies:**Brute Force:**

- Brute force is a straight-forward approach for 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 $1 \times 2 \times 3 \times \dots \times n$
 - ✓ 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.
- It works as follows
 - ✓ Compare each pair of adjacent elements from the beginning of an array.
 - ✓ Swap those elements if the elements are in reverse order
 - ✓ Repeat the steps 1 and 2 until the array is sorted
- The algorithm for bubble sort is given below

```
Algorithm Bubble_Sort(A[0,1,...n-1],n)  
//Input : An Array elements A[0,1,...n-1]  
//Output : Sorted Array  
{  
    for i ← 0 to n-2 do  
        for j ← 0 to n-2-i do  
            if(A[j] > A[j+1])  
            {  
                temp ← A[j]  
                A[j] ← A[j+1]  
                A[j+1] ← temp  
            }  
        }  
    }  
}
```

➤ **How Bubble Sort works :** Sort the following elements using Bubble Sort 7,5,9,2,3,1

➤ **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 |

➤ **Time Complexities:**

- ✓ 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:** $O(n)$: 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.

➤ **Space Complexity:** Space complexity is $O(1)$ because an extra variable (temp) is used for swapping.

➤ The python code implementation of bubble sort is given below

```
def bubblesort(a):
```

```
    n = len(a)
```

```
    for i in range(n-2):
```

```
        for j in range(n-2-i):
```

```
            if a[j]>a[j+1]:
```

```
                temp = a[j]
```

```
                a[j] = a[j+1]
```

```
                a[j+1] = temp
```

```
x = [34,46,43,27,57,41,45,21,70]
```

```
print("Before sorting:",x)
```

```
bubblesort(x)
```

```
print("After sorting:",x)
```

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.
- Algorithm for selection sort is given below

```
Algorithm Selection_Sort(A[0,1,...n-1],n)
```

```
//Input : An Array elements A[0,1,...n-1]
```

```
//Output : Sorted Array
```

```
{
    for i ← 0 to n-2 do
    {
        min ← i
        for j ← i+1 to n-1 do
        {
            if(A[j] < A[min])
                min ← j
        }
        temp ← A[i]
        A[i] ← A[min]
        A[min] ← temp
    }
}
```

- **How Selection Sort works:** Sort the following elements using Selection Sort 70,30,40,20,50,60,10,65

| Original Array | After 1 st pass | After 2 nd Pass | After 3 rd Pass | After 4 th Pass | After 5 th 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 |

- **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: $O(n^2)$:** It occurs when the array is already sorted
- ✓ **Average Case Complexity: $O(n^2)$:** It occurs when the elements of the array are in random order.

- The time complexity of the selection sort is the same in all cases.

- **Space Complexity:** Space complexity is $O(1)$ because an extra variable temp is used for swapping.

- The python code implementation of selection sort is given below

```
def selectionsort(a):
    n = len(a)
    for i in range(n-2):
        min = i
        for j in range(i+1,n-1):
            if a[j]<a[min]:
                min=j
        temp = a[i]
        a[i] = a[min]
        a[min] = temp
x = [34,46,43,27,57,41,45,21,70]
print("Before sorting:",x)
selectionsort(x)
print("After sorting:",x)
```

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.

- The algorithm for linear search is given below

```

Algorithm Sequential_Search(A[0,1,...n-1],n,key)
//Input : An Array elements A[0,1,...n-1] and search key
//Output : Returns of the index of element if found otherwise
returns -1
{
    for i ← 0 to n-1 do
    {
        if(A[i]=key)
        return i
    }
    return -1
}

```

- The **time complexity** of linear search is $O(n)$ and **space complexity** is $O(1)$
- The python code implementation of linear search is given below

```

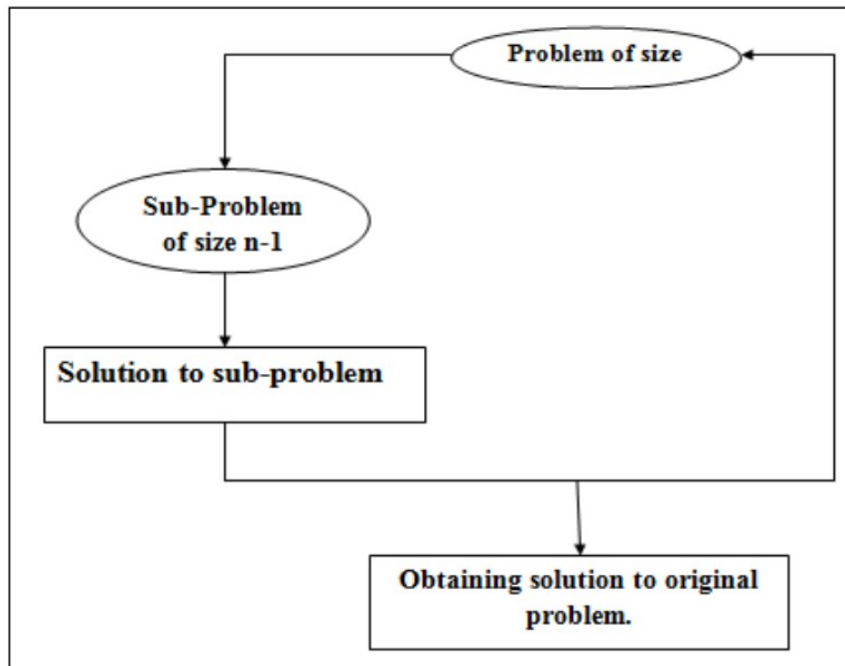
def linearsearch(a, key):
    n = len(a)
    for i in range(n):
        if a[i] == key:
            return i;
    return -1
a = [13,24,35,46,57,68,79]
print("the array elements are:",a)
k = int(input("enter the key element to search:"))
i = linearsearch(a,k)
if i == -1:
    print("Search UnSuccessful")
else:
    print("Search Successful key found at location:",i+1)

```

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:
 - ✓ decrease by a constant
 - ✓ decrease by a constant factor

- ✓ variable size decrease



Insertion Sort:

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

Algorithm Insertion_sort (A[0...n-1], n)

//Input : Array A[0..n-1] with n elements

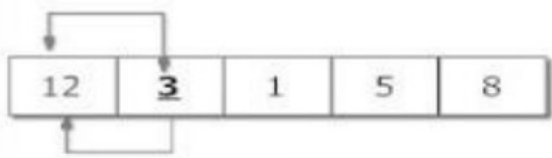
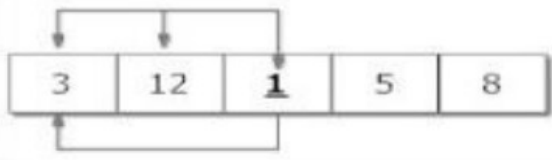
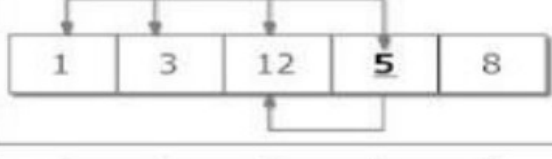
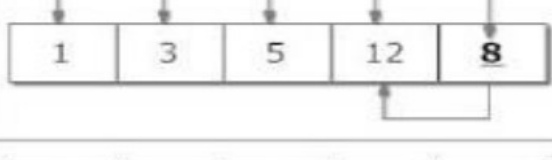

//Output : Sorted Array

```

{
  for i ← 1 to n-1 do
  {
    k ← A[i]
    j ← i-1
    while j ≥ 0 AND A[j] > k do
    {
      A[j+1] ← A[j]
      j ← j-1
    }
    A[j+1] ← k
  }
}

```


- The working of insertion sort is given below

| | | |
|---------------|---|--|
| Step 1 |  | Checking second element of array with element before it and inserting it in proper position. In this case, 3 is inserted in position of 12. |
| Step 2 |  | Checking third element of array with elements before it and inserting it in proper position. In this case, 1 is inserted in position of 3. |
| Step 3 |  | Checking fourth element of array with elements before it and inserting it in proper position. In this case, 5 is inserted in position of 12. |
| Step 4 |  | Checking fifth element of array with elements before it and inserting it in proper position. In this case, 8 is inserted in position of 12. |
| |  | Sorted Array in Ascending Order |

➤ **Time Complexities:**

- The time complexity of the insertion sort is $O(n^2)$

✓ **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 at all. So, there are only n numbers of comparisons. Thus, complexity is linear.

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

➤ **Space Complexity:** Space complexity is $O(1)$ because an extra variable k is used.

- The python code implementation of insertion sort is given below

```
def insertionsort(a):
    n = len(a)
    for i in range(1,n-1):
        k = a[i]
        j = i-1
        while j>=0 and a[j]>k:
            a[j+1] = a[j]
            j=j-1
        a[j+1] = k
x = [34,46,43,27,57,41,45,21,70]
print("Before sorting:",x)
insertionsort(x)
print("After sorting:",x)
```