16 SORTING (NCERT CLASS 12)

INTRODUCTION

Sorting is the process of ordering or arranging a given collection of elements in some particular order.

If the collection is of strings, we can sort it in an alphabetical order (a-z or z-a) or according to the length of the string.

BUBBLE SORT

It sorts a given list of elements by repeatedly comparing the adjacent elements and swapping them if they are unordered.

In algorithm, every iteration through each element of a list is called a pass. For a list with n elements, the bubble sort makes a total of n – 1 passes to sort the list.

In each pass, the required pairs of adjacent elements of the list will be compared. In order to arrange elements in ascending order, the largest element is identified after each pass and placed at the correct position in the list.

This can be considered as the largest element being 'bubbled up'. Hence the name Bubble sort. This sorted element is not considered in the remaining passes and thus the list of elements gets reduced in successive passes.

Algorithm : Bubble Sort

BUBBLESORT(numList, n)
Step 1: SET i = 0
Step 2: WHILE i< n REPEAT STEPS 3 to 8
Step 3: SET j = 0
Step 4: WHILE j< n-i-1,REPEAT STEPS 5 to 7
Step 5: IF numList[j] > numList[j+1] THEN
Step 6: swap(numList[j],numList[j+1])
Step 7: SET j=j+1
Step 8: SET i=i+1

Program : Implementation of bubble sort using Python.

```
def bubble_Sort(list1):
n = len(list1)
for i in range(n): # Number of passes
for j in range(0, n-i-1):
# size -i-1 because last i elements are already sorted
#in previous passes
if list1[j] > list1[j+1] :
# Swap element at jth position with (j+1)th position
list1[j], list1[j+1] = list1[j+1], list1[j]
numList = [8, 7, 13, 1, -9, 4]
bubble_Sort(numList)
print ("The sorted list is :")
for i in range(len(numList)):
print (numList[i], end=" ")
```

Output:
The sorted list is :
-9 1 4 7 8 13

SELECTION SORT

To sort a list having n elements, the selection sort makes (n-1) number of passes through the list. The list is considered to be divided into two lists — the left list containing the sorted elements, and the right list containing the unsorted elements. Initially, the left list is empty, and the right list contains all the elements.

For arranging elements in ascending order, in the first pass, all the elements in the unsorted list are traversed to find the smallest element.

The smallest element is then swapped with the leftmost element of the unsorted list. This element occupies the first position in the sorted list, and it is not considered in further passes.

In the second pass, the next smallest element is selected from the remaining elements in the unsorted list and swapped with the leftmost element of the unsorted list.

This element occupies the second position in the sorted list, and the unsorted list reduces by one element for the third pass.

This process continues until n-1 smallest elements are found and moved to their respective places. The nth element is the last, and it is already in place.

Algorithm 5.2: Selection Sort

SELECTIONSORT(numList, n)
Step 1: SET i=0
Step 2: WHILE i< n REPEAT STEPS 3 to 11
Step 3: SET min = i, flag = 0
Step 4: SET j= i+1
Step 5: WHILE j<n, REPEAT STEPS 6 to 10
Step 6: IF numList[j] < numList[min] THEN
Step 7: min = j
Step 8: flag = 1
Step 9: IF flag = 1 THEN
Step 10: swap(numList[i],numList[min])
Step 11: SET i=i+1

Program : Implementation of selection sort using Python :

```
def selection_Sort(list2):
flag = 0 #to decide when to swap
n=len(list2)
for i in range(n): # Traverse through all list elements
min = i
for j in range(i + 1, len(list2)): #the left elements
#are already sorted in previous passes
if list2[j] < list2[min]: # element at j is smaller
#than the current min element
min = j
flag = 1
if flag == 1 : # next smallest element is found
list2[min], list2[i] = list2[i], list2[min]
numList = [8, 7, 13, 1, -9, 4]
selection_Sort(numList)
print ("The sorted list is :")
for i in range(len(numList)):
print (numList[i], end=" ")
```

Output:

The sorted list is :

-9 1 4 7 8 13

INSERTIONSORT

Insertion sort is another sorting algorithm that can arrange elements of a given list in ascending or descending order. Like Selection sort, in Insertion sort also, the list is divided into two parts - one of sorted elements and another of unsorted elements. Each element in the unsorted list is considered one by one and is inserted into the sorted list at its appropriate position. In each pass, the sorted list is traversed from the backward direction to find the position where the unsorted element could be inserted. Hence the sorting method is called insertion sort.

In pass 1, the unsorted list has n-1 elements and the sorted list has a single element (say element s). The first element of the unsorted list (say element e) is compared with the element s of sorted list. If element e is smaller than element s, then element s is shifted to the right making space for inserting element e. This shifting will now make sorted list of size 2 and unsorted list of size n-2.

In pass 2, the first element (say element e) of unsorted list will be compared with each element of sorted list starting from the backward direction till the appropriate position for insertion is found. The elements of sorted list will be shifted towards right making space for the element e where it could be inserted.

This continues till all the elements in unsorted lists are inserted at appropriate positions in the sorted list. This results into a sorted list in which elements are arranged in ascending order.

Algorithm 5.3: Insertion Sort

INSERTIONSORT(numList, n)

Step 1: SET i=1

Step 2: WHILE i< n REPEAT STEPS 3 to 9

Step 3: temp = numList[i]

Step 4:SET j = i-1

Step 5: WHILE j> = 0 and numList[j]>temp,REPEAT STEPS 6 to 7

Step 6: numList[j+1] = numList[j]

Step 7: SET j=j-1

Step 8: numList[j+1] = temp #insert temp at position j

Step 9: set i=i+1

Program Implementation of insertion sort using Python :

```
def insertion_Sort(list3):
n= len(list3)
for i in range(n): # Traverse through all elements
temp = list3[i]
j = i-1
while j >=0 and temp< list3[j] :
list3[j+1] = list3[j]
j = j-1 list3[j+1] = temp
numList = [8, 7, 13, 1, -9, 4]
insertion_Sort(numList)
print ("The sorted list is :")
for i in range(len(numList)):
print (numList[i], end=" ")
```

Output:

The sorted list is :

-9 1 4 7 8 13

TIME COMPLEXITY OF ALGORITHMS

The following tips will guide us in estimating the time complexity of an algorithm:-

(I) Any algorithm that does not have any loop will have time complexity as 1 since the number of instructions to be executed will be constant, irrespective of the data size. Such algorithms are known as Constant time algorithms.

(II) Any algorithm that has a loop (usually 1 to n) will have the time complexity as n because the loop will execute the statement inside its body n number of times. Such algorithms are known as Linear time algorithms.

(III) A loop within a loop (nested loop) will have the time complexity as n2. Such algorithms are known as Quadratic time algorithms.

(IV) If there is a nested loop and also a single loop, the time complexity will be estimated on the basis of the nested loop only.

According to the above rules, all the sorting algorithms namely, bubble sort, selection sort and insertion sort have a time complexity of n^2. □□□

17 SEARCHING
(NCERT CLASS 12)

INTRODUCTION

Searching means locating a particular element in a collection of elements. Search result determines whether that particular element is present in the collection or not. If it is present, we can also find out the position of that element in the given collection. Searching is an important technique in computer science. In order to design algorithms, programmers need to understand the different ways in which a collection of data can be searched for retrieval.

LINEAR SEARCH

It is an exhaustive searching technique where every element of a given list is compared with the item to be searched (usually referred to as 'key'). So, each element in the list is compared one by one with the key. This process continues until an element matching the key is found and we declare that the search is successful. If no element matches the key and we have traversed the entire list, we declare the search is unsuccessful i.e., the key is not present in the list. This item by item comparison is done in the order, in which the elements are present in the list, beginning at the first element of the list and moving towards the last. Thus, it is also called sequential search or serial search. This technique is useful for collection of items that are small in size and are unordered.

Algorithm 6.1 : Linear Search

LinearSearch(numList, key, n)
Step 1: SET index = 0
Step 2: WHILE index < n, REPEAT Step 3
Step 3: IF numlist[index]= key THEN
PRINT "Element found at position", index+1
STOP
ELSE
index = index+1
Step 4: PRINT "Search unsuccessful"

The algorithm had to make only 1 comparison to display 'Element found at position 1'. Thus, if the key to be searched is the first element in the list, the linear search algorithm will always have to make only 1 comparison. This is the minimum amount of work that the linear search algorithm would have to do.

Program : Linear Search

```
def linear Search (list, key) :#function to perform the search
        for index in range(0, len(list)):
              in list [index] == key :    #key is present
                  return index+1   #position of key in list
    return None     #key is not in list
#end of function
list1 = []        #create an empty lsit
maximum = int(input ("How many elements in your list?"))
print ('Enter each element and press enter:')
for i in range (0, maximum):
    n = int (input ())
    list1.append(n)                  #append element to the list
print ("The List contents are :".list)

key = int (input ("Enter the number to be searched:")
position = linear Search (list1.key)
if position is non:
    print ('Number'.key. 'is not present in the list')
else
    print ('Number'.key, 'is present at position'.position)
```

Output
How many elements in your list? 4
Enter each element and press enter:
12
23
3
-45
The List contents are: [12, 23, 3, -45]
Enter the number to be searched:23
Number 23 is present at position 2

BINARY SEARCH

The binary search is a search technique that makes use of the ordering of elements in the list to quickly search a key. For numeric values, the elements in the list may be arranged either in ascending or descending order of their key values. For textual data, it may be arranged alphabetically starting from a to z or from z to a.

In binary search, the key to be searched is compared with the element in the middle of a sorted list. This could result in either of the three possibilities:

(I) the element at the middle position itself matches the key
or
(II) the element at the middle position is greater than the key
or
(III) the element at the middle position is smaller than the key.

If the element at the middle position matches the key, we declare the search successful and the searching process ends.

Algorithm 6.2: Binary Search

BinarySearch(numList, key)
Step 1: SET first = 0, last = n-1
Step 2: Calculate mid = (first+last)//2
Step 3: WHILE first <= last REPEAT Step 4
Step 4: IF numList[mid] = key
PRINT "Element found at position", " mid+1
STOP
ELSE
IF numList[mid] > key, THEN last = mid-1
ELSE first = mid + 1
Step 5: PRINT "Search unsuccessful"

Note : The binary search algorithm does not change the list. Rather, after every pass of the algorithm, the search area gets reduced by half. That is, only the index of the element to be compared with the key changes in each iteration.

Program:- Binary search:

```
def binary Search (list, key):
        first = 0
        last = len (list) - 1
        while (first <=last):
                mid = (first + last)//2
                    if list[mid] == key:
                     return mid
                elif key > list[mid]:
                     first = mid + 1
                elif key < list (mid) :
                     last = mid - 1
            return - 1
    list = []          #create an empty list
    print ("Create a lsit by entering elements in ascening order")
    print ("press enter after each element, press - 999 to stop")
    num = int (input ())
    while num! = -999:
          list.append (num)
          num = int (input ())
    n = int (input ("Enter the key to be search:")
    pos = binary Search (list 1, n)
    if (pos ! = -1):
      print (n, "is found at position",post+ 1)
    else:
      print (n, "is not found in the list")
```

Output:

Create a list by entering elements in ascending order
press enter after each element, press -999 to stop
1
3
4
5
-999
Enter the number to be searched: 4
4 is found at position 3
Second run of the program with different data:
Create a list by entering elements in ascending order
press enter after each element, press -999 to stop
12
8
3
-999
Enter the number to be searched: 4
4 is not found in the list

- **Applications of Binary Search**

(I) Binary search has numerous applications including – searching a dictionary or a telephone directory, finding the element with minimum value or maximum value in a sorted list, etc.

(II) Modified binary search techniques have far reaching applications such as indexing in databases, implementing routing tables in routers, data compression code, etc.

SEARCH BY HASHING

Hashing is a technique which can be used to know the presence of a key in a list in just one step.

A hash function takes elements of a list one by one and generates an index value for every element. This will generate a new list called the hash table. Each index of the hash table can hold only one item and the positions are indexed by integer values starting from 0. Note that the size of the hash table can be larger than the size of the list.

A simple hash function that works with numeric values is known as the remainder method. It takes an element from a list and divides it by the size of the hash table. The remainder so generated is called the hash value.

h(element) = element % size(hash table)

COLLISION

The hashing technique works fine if each element of the list maps to a unique location in the hash table. Consider a list [34, 16, 2, 26, 80]. While applying the hash function say, list [i]%10, two elements (16 and 26) would have a hash value 6. This is a problematic situation, because according to our definition, two or more elements cannot be in the same position in the list. This situation is called collision in hashing.
We must have a mechanism for placing the other items with the same hash value in the hash table. This process is called collision resolution.

If every item of the list maps to a unique index in the hash table, the hash function is called a perfect hash function. If a hash function is perfect, collision will never occur.

Apart from modulo division method, hash functions may be based on several other techniques like integer division, shift folding, boundary folding, mid-square function, extraction, radix transformation, etc.

The time taken by different hash functions may be different, but it remains constant for a particular hash function. The advantage of hashing is that the time required to compute the index value is independent of the number of items in the search list. It is to remember that the cost of computing a hash function must be small enough to make a hashing-based searching more efficient than other search methods. ❐❐❐