Unit III

3 Searching and Sorting

3.1 : Introduction to Searching and Sorting

Q.1 What are the applications of sorting ?

Ans. : Sorting is useful for arranging the data in desired order. After sorting the required element can be located easily.

1. The sorting is useful in database applications for arranging the data in desired order.
2. In the dictionary like applications the data is arranged in sorted order.
3. For searching the element from the list of elements, the sorting is required.
4. For checking the uniqueness of the element the sorting is required.
5. For finding the closest pair from the list of elements the sorting is required.

3.2 : Searching Techniques

Q.2 What is searching ?

Ans. : When we want to find out particular record efficiently from the given list of elements then there are various methods of searching that element. These methods are called **searching methods.** Various algorithms based on these searching methods are known as **searching algorithms.**

Most commonly used searching algorithms are -

i) Sequential or linear search

ii) Indexed sequential search

iii) Binary search

Q.3 Explain sequential search technique.

Ans. : • Sequential search is technique in which the given list of elements is scanned from the beginning. The key element is compared with every element of the list. If the match is found the searching is stopped otherwise it will be continued to the end of the list.

- Although this is a **simple method,** there are some **unnecessary comparisons** involved in this method.
- The time complexity of this algorithm is **O(n).** The time complexity will increase linearly with the value of n.
- For **higher value of n** sequential search is **not satisfactory solution.**
- **Example**

Array

	Roll no	Name	Marks
0	15	Parth	96
1	2	Anand	40
2	13	Lalita	81
3	1	Madhav	50
4	12	Arun	78
5	3	Jaya	94

Fig. Q.3.1 Represents students database for sequential search

From the above Fig. Q.3.1 the array is maintained to store the students record. The record is not sorted at all. If we want to search the student's record whose roll number is 12 then with the key-roll number we will see the every record whether it is of roll number = 12. We can obtain such a record at Array [4] location.

C++ Function

```
int search(int a[size],int key)
{
 for(i=0;i<n;i++)
   {
      if(a[i]==key)
         return 1;
   }
return 0;
}
```

Q.4 Explain Binary search algorithm with suitable example.

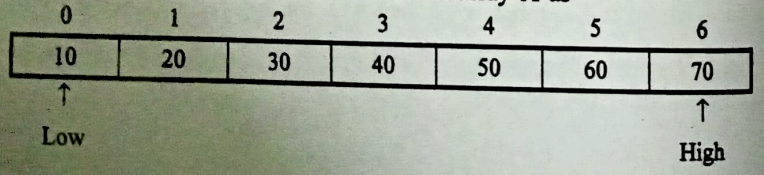
Ans. : • Concept : Binary search is a searching algorithm in which the list of elements is divided into two sublists and the key element is compared with the middle element. If the match is found then, the location of middle element is returned otherwise, we search into either of the halves(sublists) depending upon the result produced through the match. This algorithm is considered as an efficient searching algorithm.

Algorithm for binary search

1. if(low>high)
2. return;
3. mid =(low+high)/2;
4. if(x==a[mid])
5. return (mid);
6. if(x<a[mid])
7. search for x in a[low] to a[mid-1];
8. else
9. search for x in a[mid+1] to a[high];

Q.5 Apply binary search method to search 60 from the list 10, 20, 30, 40, 50, 60, 70.

Ans. : Consider a list of elements stored in array A as



The KEY element (i.e. the element to be searched) is 60.

Now to obtain middle element we will apply a formula :

$$m = (low + high)/2$$

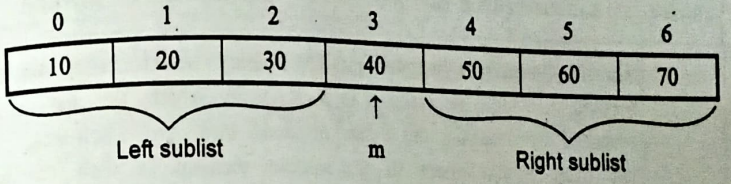
$$m = (0 + 6)/2$$

$$m = 3$$

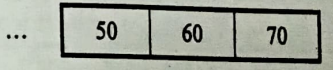
Then Check $A[m] \stackrel{?}{=} KEY$

i.e. $A[3] \stackrel{?}{=} 60$ NO $A[3] = 40$ and $40 < 60$

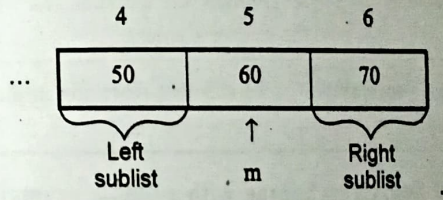
∴ Search the right sublist.



The right sublist is



Now we will again divide this list and check the mid element.



$m = (low + high)/2$

$m = (4 + 6)/2$

$\therefore m = 5$

is $A[m] \stackrel{?}{=} KEY$

i.e. $A[5] \stackrel{?}{=} 60$ Yes, i.e. the number is present in the list.

Thus we can search the desired number from the list of elements.

Advantages and disadvantages of binary search

Advantage

(1) It is efficient technique.

Disadvantages

(1) It requires specific ordering before applying the method.

(2) It is complex to implement.

Q.6 What is the difference between linear search and binary search ?

Ans. : Comparison between Linear search and Binary search

Sr. No.	Linear search method	Binary search method
1.	The linear search is a searching method in which the element is searched by scanning the entire list from first element to the last.	The binary search is a searching method in which the list is sub divided into two sub-lists. The middle element is then compared with the key element and then accordingly either left or right sub-list is searched.
2.	Many times entire list is searched.	Only sub-list is searched for searching the key element.
3.	It is simple to implement.	It involves computation for finding the middle element.
4.	It is less efficient searching method.	It is an efficient searching method.

Q.7 Explain the Fibonacci Search technique with suitable example.

☞ **[SPPU : June-22, Marks 9]**

Ans. : In binary search method we divide the number at mid and based on mid element i.e. by comparing mid element with key element we search either the left sublist or right sublist. Thus we go on dividing the corresponding sublist each time and comparing mid element with key element. If the key element is really present in the list, we can reach to the location of key in the list and thus we get the message that the "element is present in the list" otherwise get the message. "element is not present in the list."

In Fibonacci search rather than considering the mid element, we consider the indices as the numbers from fibonacci series. As we know, the Fibonacci series is -

0 1 1 2 3 5 8 13 21 ...

To understand how Fibonacci search works, we will consider one example, suppose, following is the list of elements.

arr [ ]

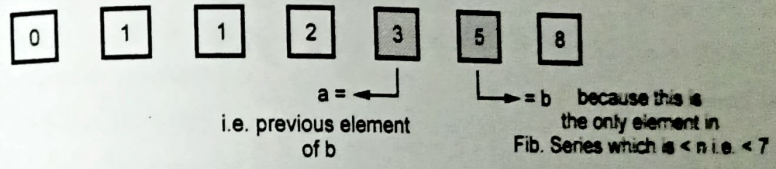
10	20	30	40	50	60	70
1	2	3	4	5	6	7

Here n = Total number of elements = 7

We will always compute 3 variables i.e. a, b and f.

Initially f = n = 7.

For setting a and b variables we will consider elements from Fibonacci series.



Now we have

f = 7

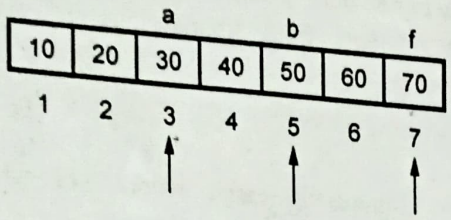
b = 5

a = 3

With these initial values we will start searching the key element from the list. Each time we will compare key element with arr [f]. That means

If (Key < arr [f])	If (Key > arr [f])
f = f – a	f = f + a
b = a	b = b – a
a = b – a	a = a – b

Suppose we have f = 7, b = 5, a = 3



If Key = 20

i.e. Key < arr[f]

i.e. 20 < 70

$\therefore f = f - a = 7 - 3 = 4$

b = a = 3

a = b – a = 2

Again we compare

if (Key < arr [f])

i.e. 20 < arr [4]

i.e. if (20 < 40) → Yes

At Present f = 4, b = 3, a = 2

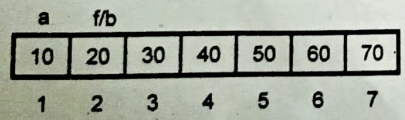
$\therefore f = f - a = 4 - 2 = 2$

b = a = 2

a = b – a = 3 – 2 = 1

Now we get f = 2, b = 2,

a = 1



If Key = 60

i.e. Key < arr [f]

60 < 70

$\therefore f = f - a = 7 - 3 = 4$

b = a = 3

a = b – a = 2

Again we compare

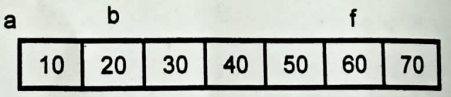
if (Key > arr [f])

$\therefore$ i.e. 60 > arr [f] i.e. 40

$\therefore f = f + a = 4 + 2 = 6$

b = b – a = 3 – 2 = 1

a = a – b = 2 – 3 = – 1



If (Key < arr [f])

i.e. if (60 < 60) → No

If (Key < arr [f]) i.e.

if (20 < 20) → No

If Key > arr [f]) i.e.

if (20 > 20) → No

That means

"Element is present at f = 2 location"

If (key > arr [f])

i.e. if (60 > 60) → No

That means

"Element is present at f = 6 location."

Analysis : The time complexity of fibonacci search is **O(log n).**

Algorithm

Let the length of given array be **n [0...n-1]** and the element to be searched be **key**

Then we use the following steps to find the element with minimum steps :

1. Find the **smallest Fibonacci number greater than or equal to n.** Let this number be f(m^{th} element).
2. Let the two Fibonacci numbers preceding it be a(m-1^{th} element) and b(m-2^{th} element)

   While the array has elements to be checked :

   Compare key with the last element of the range covered by b

   (a) If **key** matches, return index value

   (b) Else if **key is less** than the element, move the third Fibonacci variable two Fibonacci down, indicating removal of approximately two-third of the unsearched array.

   (c) Else key is greater than the element, move the third Fibonacci variable one Fibonacci down. Reset offset to index. Together this results into removal of approximately front one-third of the unsearched array.
3. Since there might be a single element remaining for comparison, check if **a** is '1'. If Yes, compare key with that remaining element. If match, return index value.

For example -

According to the algorithm we have to sort the elements of the array prior to applying Fibonacci search. Consider sorted array of elements as -

0	1	2	3	4	5	6
10	20	30	40	50	60	70

∴ n = 7, we want to find key = 60

Now check Fibbonacci series.

As n < 8

set f = 8

a = 5

b = 3

			b		a		f
0	1	2	3	4	5	6	8
10	20	30	40	50	60	70	...

Set offset = – 1

∴ i = 2 ∵ 1 = min (offset + b, n – 1)

A[i] = A[2] < (key = 60)

Here key is greater than the element

We move f one fibbonacci down (step 2C of algorithm)

∴ f = 5

a = 3

b = 2

		b	a		f	
0	1	2	3	4	5	6
10	20	30	40	50	60	70

Set offset = i i.e. = 2

Now i = 4 ∵ i = min (offset + b, n – 1)

Here key is again greater than the element. So we move f, one fibbonacci down

∴ f = 3

a = 2

b = 1

	b	a	f				
0	1	2	3	4	5	6	
10	20	30	40	50	60	70	

Set offset = i i.e. 4

Now new i = 5 ∵ i = min (offset + b, n – 1)

Here a[i] = key. Hence return value of i as the position of key element.

Note that due to fibonacci numbering the search portion is restricted and we need to compare very less number of elements.

Q.8 Write a Python program for index sequential search.

Ans. :

```
def IndexSeq(arr,key,n):
    Elements = [0]*10
    Index = [0]*10
    flag = 0
    ind = 0
    start=end=0
    for i in range(0,n,2) :
        # Storing element
        Elements[ind] = arr[i]
        # Storing the index
        Index[ind] = i
        ind += 1

    if (key < Elements[0]):
        print("Element is not present")
        exit(0)
```

```
    else:
    for i in range(1, ind + 1) :
        if(key < Elements[i]) :
            start = Index[i - 1]
            end = Index[i]
            break

for i in range(start, end + 1) :
    if (key == arr[i] ):
        flag = 1
        break

if flag == 1 :
    print("Element is Found at index", i)
else :
    print("Element is not present")

print("\n Program For Index Sequential Search")
array = [11,15,22,24,26,32,34,38]
n = len(array)
key=32
IndexSeq(array,key,n)
```

Output

```
Program For Index Sequential Search
Element is Found at index 5
```

3.3 : Types of Sorting

Q.9 Explain - internal and external sorting techniques.

☞ **[SPPU : June-22, Marks 9]**

Ans. : Internal sorting :

- The internal sorting is a sorting in which the data resides in the main memory of the computer.

- Various methods that make use of internal sorting are -
- 1. Bubble Sort 2. Insertion Sort 3. Selection Sort 4. Quick Sort 5. Radix Sort and so on.

- **External sorting :**
  - For many applications it is not possible to store the entire data on the main memory for two reasons, i) amount of **main memory available is smaller** than amount of data. ii)Secondly the **main memory is a volatile device** and thus will lost the data when the power is shut down. To overcome these problems the data is stored on the secondary storage devices. The technique which is used to sort the data which resides on the secondary storage devices are called **external sorting.**

3.4 : General Sort Concepts

Q.10 Explain the terms - ascending order and descending order.

Ans. : Ascending order : It is the sorting order in which the elements are arranged from low value to high value. In other words elements are in increasing order.

For example : 10, 50, 40, 20, 30

can be arranged in ascending order after applying some sorting technique as

10, 20, 30, 40, 50

Descending order : It is the sorting order in which the elements are arranged from high value to low value. In other words elements are in decreasing order. It is reverse of the ascending order.

For example : 10, 50, 40, 20, 30

can be arranged in descending order after applying some sorting technique as

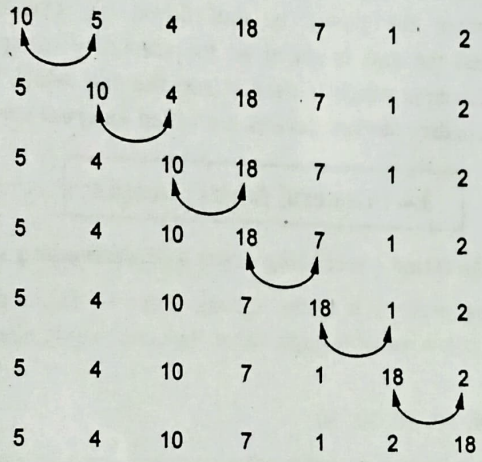
50, 40, 30, 20, 10

3.5 : Comparison based Sorting Methods

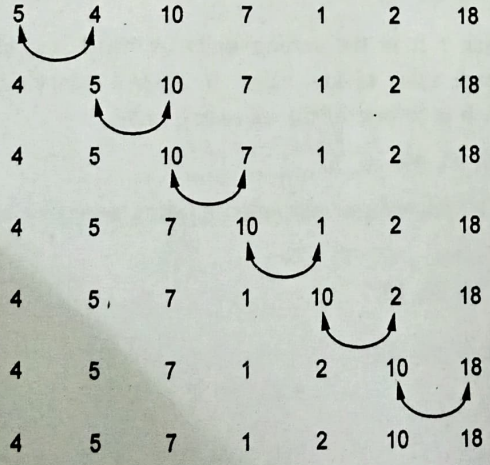
Q.11 Show the output of each pass for the following list 10, 5, 4, 18, 17, 1, 2.

Ans. : Let, 10, 5, 4, 18, 17, 1, 2 be the given list of elements. We will compare adjacent elements say A[i] and A[j]. If A[i] > A[j] then swap the elements.

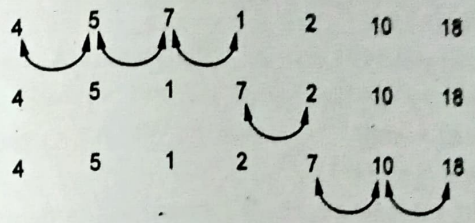
Pass I



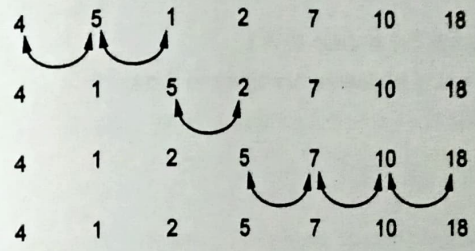
Pass II



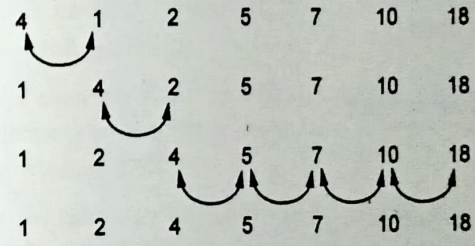
Pass III



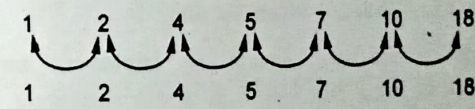
Pass IV



Pass V



Pass VI



This is the sorted list of elements.

Q.12 Write a Python program for sorting the elements using Bubble sort.

Ans. :

```
def Bubble(arr,n):
    i = 0
```

```
    for i in range(n-1):
        for j in range(0,n-i-1):
            if(arr[j] > arr[j+1]):
                temp = arr[j]
                arr[j] = arr[j+1]
                arr[j+1] = temp
        print("\nPass#",(i+1))
        print(arr)

print("\n Program For Bubble Sort")
print("\nHow many elements are there in Array?")
n = int(input())
array = []
i=0
for i in range(n):
    print("\n Enter element in Array")
    item = int(input())
    array.append(item)

print("Original array is\n")
print(array)

print("\n Sorted Array is")
Bubble(array,n)
```

Q.13 Write an algorithm for insertion sort.

Ans. : Although it is very natural to implement insertion using recursive(top down) algorithm but it is very efficient to implement it using bottom up(iterative) approach.

Algorithm Insert_sort(A[0...n-1])
//Problem Description: This algorithm is for sorting the
//elements using insertion sort
//Input: An array of n elements
//Output: Sorted array A[0...n-1] in ascending order


```
for i ← 1 to n-1 do
{
    temp ← A[i]//mark A[i]th element
    j ← i-1//set j at previous element of A[i]
    while(j>=0)AND(A[j]>temp)do
    {
  //comparing all the previous elements of A[i] with
  //A[i].If any greater element is found then insert
  //it at proper position
    A[j+1] ← A[j]
    j ← j-1
    }
    A[j+1] ← temp //copy A[i] element at A[j+1]
}
```

Analysis

When an array of elements is almost sorted then it is **best case** complexity. The best case time complexity of insertion sort is **O(n).**

If an array is randomly distributed then it results in **average case** time complexity which is $O(n^2)$.

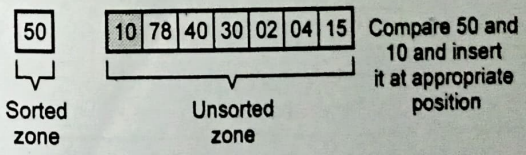
If the list of elements is arranged in descending order and if we want to sort the elements in ascending order then it results in **worst case** time complexity which is $O(n^2)$.

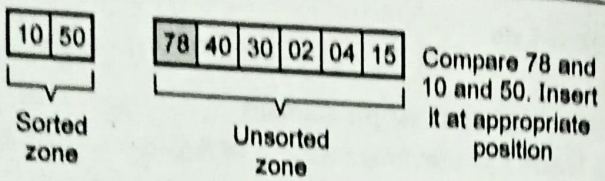
Q.14 Sort the following numbers using insertion sort. Show all passes 50, 10, 78, 40, 30, 02, 04, 15.

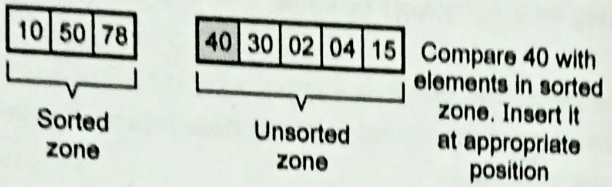
Ans. : Consider the list of elements as -

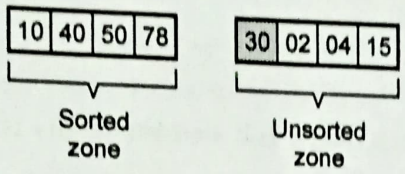
0	1	2	3	4	5	6	7
50	10	78	40	30	02	04	15

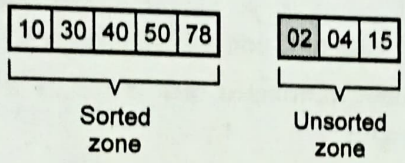
The process starts with first element.

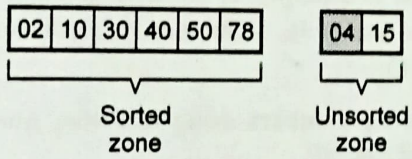


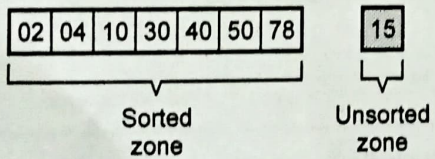


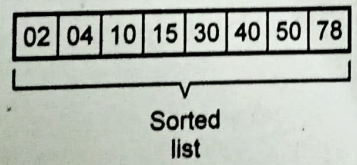






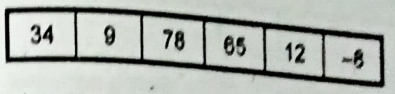






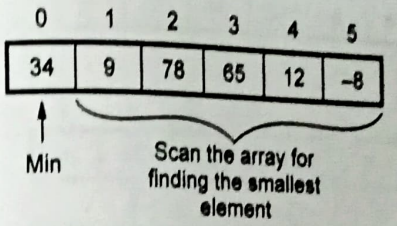
Q.15 Sort the following and show the status after every pass using selection sort : 34, 9, 78, 65, 12, – 8.

Ans. : Let

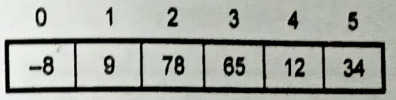


be the given elements.

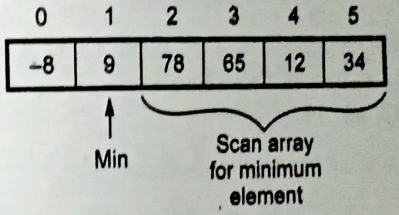
Pass 1 : Consider the elements A[0] as the first element. Assume this as the minimum element.



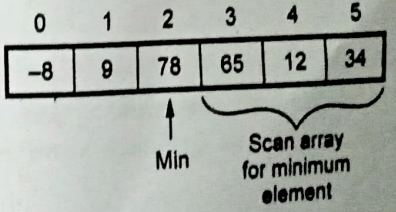
If smallest element is found, swap it with A[0]



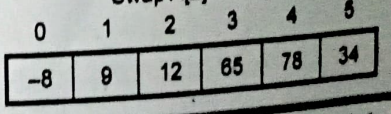
Pass 2 :



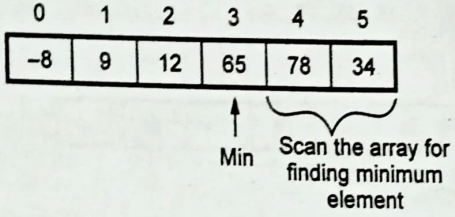
Pass 3 :

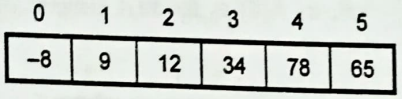


Swap A[2] and A[4]

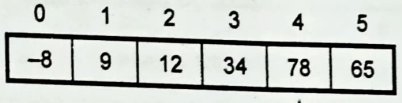


Pass 4 :

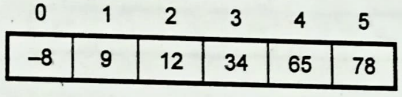




Pass 5 :



Swap A[4] and A[5]



This is a sorted list

Q.16 Consider following numbers, sort them using quick sort. Show all passes to sort the values in ascending order
25, 57, 48, 37, 12, 92, 86, 33.

Ans. : Consider the first element as a pivot element.

25	57	48	37	12	92	86	33
↑	↑						↑
Pivot	i						j

Now, if A[i] < Pivot element then increment i. And if A[j] > Pivot element then decrement j. When we get these above conditions to be false, Swap A[i] and A[j]

25	33	48	37	12	92	86	57
↑	i						j
Pivot							

25	33	48	37	12	92	86	57
↑	i			j			
Pivot							

Swap A[i] and A[j]

Low							High
25	12	48	37	33	92	86	57
	j	i					

As j > i Swap A[Low] and A[j]

After pass 1 :

[12]	25	[48	37	33	92	86	57]
		↑	↑				↑
		Pivot	i				j

12	25	[48	37	33	92	86	57]
					i		j
		Low					

12	25	[48	37	33	92	86	57]
				j	i		

As j > i, we will swap A[j] and A[Low]

After pass 2 :

12	25	[33	37]	48	[92	86	57]

After pass 3 :

12	25	33	37	48	[92	86	57]

Assume 92 to be pivot element

12	25	33	37	48	[92	86	57]
					↑	i	j
					Pivot		

12	25	33	37	48	92	86	57	
							j	i

As j > i swap 92 with 57.

After pass 4 :

12 25 33 37 48 57 [86 92]

After pass 5 :

12 25 33 37 48 57 86 92

is a sorted list.

Python Program

```
def Quick(arr,low,high):
    if(low<high):
        m=Partition(arr,low,high)
        Quick(arr,low,m-1)
        Quick(arr,m+1,high)

def Partition(arr,low,high):
    pivot = arr[low]
    i=low+1
    j=high
    flag = False
    while(not flag):
        while(i<=j and arr[i]<=pivot):
            i = i + 1
        while(i<=j and arr[j]>=pivot):
            j = j - 1

        if(j<i):
            flag = True
        else:
            temp = arr[i]
            arr[i] = arr[j]
            arr[j] = temp

    temp = arr[low]
    arr[low] = arr[j]
    arr[j] = temp
    return j
```



```
print("\n Program For Quick Sort")
print("\nHow many elements are there in Array?")
n = int(input())
array = []
i=0
for i in range(n):
    print("\n Enter element in Array")
    item = int(input())
    array.append(item)

print("Original array is\n")
print(array)

Quick(array,0,n-1)
print("\n Sorted Array is")
print(array)
```

Output

```
Program For Quick Sort
How many elements are there in Array?
8
 Enter element in Array
50
 Enter element in Array
30
 Enter element in Array
10
 Enter element in Array
90
 Enter element in Array
80
 Enter element in Array
20
 Enter element in Array
40
 Enter element in Array
```

70
Original array is
[50, 30, 10, 90, 80, 20, 40, 70]
Sorted Array is
[10, 20, 30, 40, 50, 70, 80, 90]
>>>

Q.17 Explain Shell sort method with suitable example.

Ans. : This method is a improvement over the simple insertion sort. In this method the elements at fixed distance are compared. The distance will then be decremented by some fixed amount and again the comparison will be made. Finally, individual elements will be compared.

Example : If the original file is

	0	1	2	3	4	5	6	7
X array	25	57	48	37	12	92	86	33

Step 1 : Let us take the distance k = 5

So in the first iteration compare

(x [0], x[5])

(x [1], x [(6])

(x [2], x[7])

(x[3])

(x[4])

i.e. first iteration

After first iteration,

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
25	57	33	37	12	92	86	48

Step 2 : Initially k was 5. Take some d and decrement k by d. Let us take d = 2

$\therefore$ k = k – d i.e. k = 5 – 2 = 3

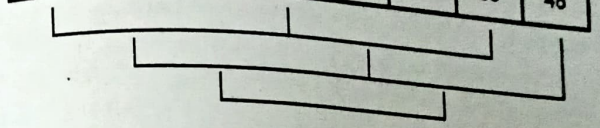
So now compare

(x[0], x[3], x[6]), (x[1], x[4], x[7])

(x[2], x[5])

Second iteration

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
25	57	33	37	12	92	86	48



After second iteration

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
25	12	33	37	48	92	86	57

Step 3 : Now k = k – d $\therefore$ k = 3 – 2 = 1

So now compare

(x[0], x[1], x[2], x[3], x[4], x[5], x[6], x[7])

This sorting is then done by simple insertion sort. Because simple insertion sort is highly efficient on sorted file. So we get

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
12	25	33	37	48	57	86	92

Python Program

```
def ShellSort(arr,n):
    d = n//2
    while d > 0:
        for i in range(d,n):
            temp = arr[i]
            j = i
            while(j >= d and arr[j-d] >temp):
                arr[j] = arr[j-d]
                j -= d
```

```
            arr[j] = temp
        d = d//2
print("\n Program For Shell Sort")
print("\nHow many elements are there in Array?")
n = int(input())
array = []
i=0
for i in range(n):
    print("\n Enter element in Array")
    item = int(input())
    array.append(item)

print("Original array is\n")
print(array)

ShellSort(array,n)
print("\n Sorted Array is")
print(array)
```

Output

```
Program For Shell Sort
How many elements are there in Array?
5
 Enter element in Array
30
 Enter element in Array
20
 Enter element in Array
10
 Enter element in Array
40
 Enter element in Array
50
Original array is
[30, 20, 10, 40, 50]
 Sorted Array is
[10, 20, 30, 40, 50]
>>>
```


Analysis :

Best Case : The best case in the shell sort is when the array is already sorted in the right order. The number of comparisons is less. In that case the inner loop does not need to any work and a simple comparison will be sufficient to skip the inner sort loop. the other loops gives O(nlogn). The best case of O(n) is reached by using a constant number of increments. Hence the best case time complexity of shell sort is **O(nlogn)**

Worst Case and Average Case : The running time of Shellsort depends on the choice of increment sequence. The problem with Shell's increments is that pairs of increments are not necessarily relatively prime and smaller increments can have little effect. The wort case and average case time complexity is **O(n)**.

3.6 : Non-Comparison based Sorting Methods

Q.18 Sort the following data in ascending order using Radix Sort : 25, 06, 45, 60, 140, 50. ☞ **[SPPU : June-22, Marks 9]**

Ans. :

Step 1 :

Sort the elements according to last digit and sort them.

Last digit	Element
0	50, 60, 140
1	
2	
3	
4	
5	25, 45
6	06
7	
8	
9	

Step 2 :

Sort the elements according to second last digit and sort them.

Second last digit	Element
0	06
1	
2	25
3	
4	45, 140
5	50
6	60
7	
8	

Step 3 :

Sort the elements according to 100^{th} position of the element and so them.

100^{th} position	Element
0	06, 25, 45, 50, 60
1	140
2	
3	
4	
5	
6	
7	
8	
9	

Thus the sorted list of elements is

06, 25, 45, 50, 60, 140

Algorithm :

1. Read the total number of elements in the array.
2. Store the unsorted elements in the array.
3. Now the simple procedure is to sort the elements by digit by digit.
4. Sort the elements according to the last digit then second last digit and so on.
5. Thus the elements should be sorted for up to the most significant bit.
6. Store the sorted element in the array and print them.
7. Stop.

Python Program

```
def RadixSort(arr):
    MaxElement = max(arr)
    place = 1
    while MaxElement      //place > 0:
        countingSort(arr,place)
        place = place * 10

def countingSort(arr,place):
    n = len(arr)
    result = [0]*n
    count = [0] *10
    #calculating the count of elements based on digits place
    i = 0
    for i in range(n):
        index = arr[i]          // place
        count[index % 10] += 1

    for i in range(1, 10):
        count[i] = count[i]+count[i - 1]
    #placing the elements in sorted order
    i = n - 1
```

```
while i >=0:
    index = arr[i]          // place
    result[count[index % 10] - 1] = arr[i]
    count[index % 10] -= 1
    i = i-1
#placing back the sorted elements in original
for i in range(0, n):
    array[i] = result[i]

print("\n Program For Radix Sort")
print("\nHow many elements are there in Array?")
n = int(input())
array = []
i=0
for i in range(n):
    print("\n Enter element in Array")
    item = int(input())
    array.append(item)

print("Original array is\n")
print(array)

RadixSort(array)
print("\n Sorted Array is")
print(array)
```

Output

```
Program For Radix Sort
How many elements are there in Array?
6
 Enter element in Array
121
 Enter element in Array
235
 Enter element in Array
55
 Enter element in Array
973
```



```
Enter element in Array
327
Enter element in Array
179
Original array is
[121, 235, 55, 973, 327, 179]
Sorted Array is
[55, 121, 179, 235, 327, 973]
>>>
```

Q.19 Write an algorithm for radix sort.

Ans. :

1. Read the total number of elements in the array.
2. Store the unsorted elements in the array.
3. Now the simple procedure is to sort the elements by digit by digit.
4. Sort the elements according to the last digit then second last digit and so on.
5. Thus the elements should be sorted for up to the most significant bit.
6. Store the sorted element in the array and print them.
7. Stop.

Q.20 What is counting sort ? Explain it with suitable example.

Ans. : Concept : Counting sort is a sorting algorithm that sorts the elements of an array by counting the number of occurrences of each unique element in the array. The count is stored in an auxiliary array and the sorting is done by mapping the count as an index of the auxiliary array.

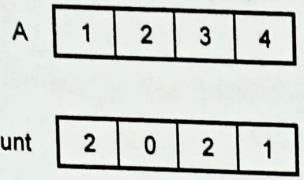
Q.21 Apply counting sort for the following numbers to sort in ascending order. 4, 1, 3, 1, 3.

Ans. : Step 1 : We will find the min and max values from given array. The min = 1 and max = 4. Hence create an array A from 1 to 4.

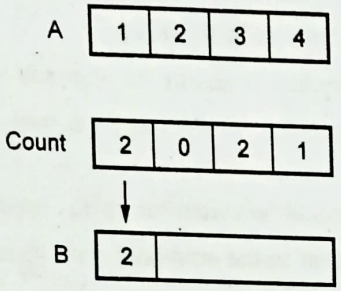
A	1	2	3	4

Now create another array named **count.** Just count the number of occurrences of each element and store that count in count array at

corresponding location of element i.e. element 1 appeared twice, element 2 is not present, element 3 appeared twice and element 4 appeared once.

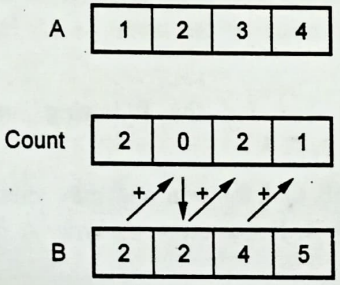


Step 2 : Now we will create another array B in which we will store sum of counts for given index.

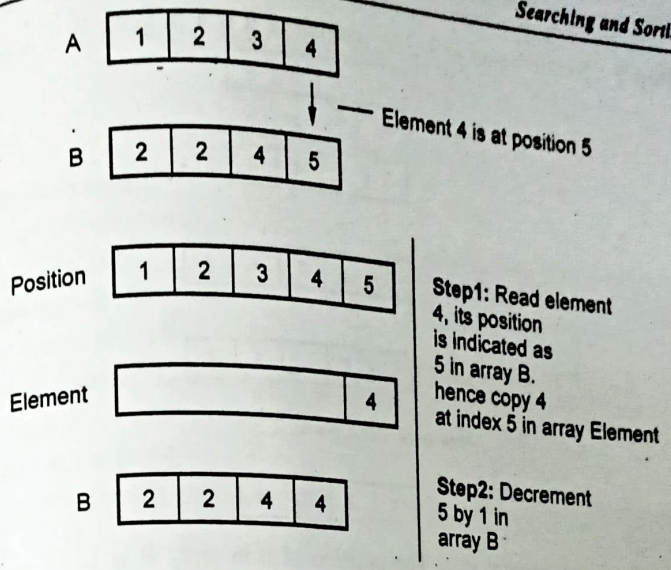


Simply copy first index value of count array to B.

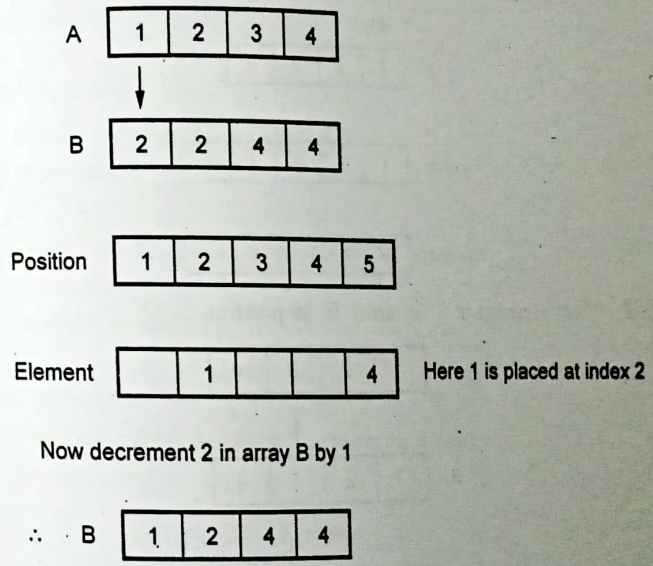
For filling up rest of the elements to B array copy the sum of previous index value of B with current index value of count array.



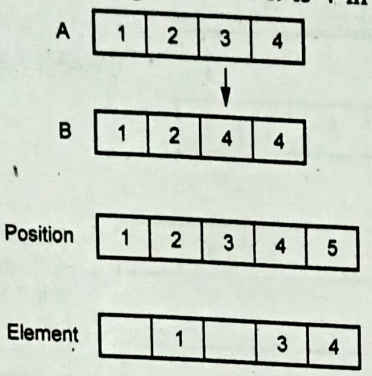
Step 3 : Now consider array A and B for creating two more arrays namely **Position** and **Element.**



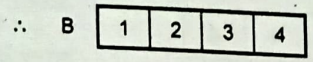
Step 4 : Next element is 1. The position of it is 2.



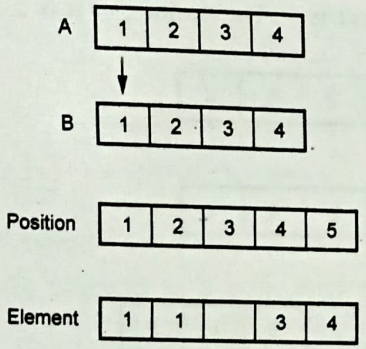
Step 5 : Next element is 3. The position of it is 4 in array B.



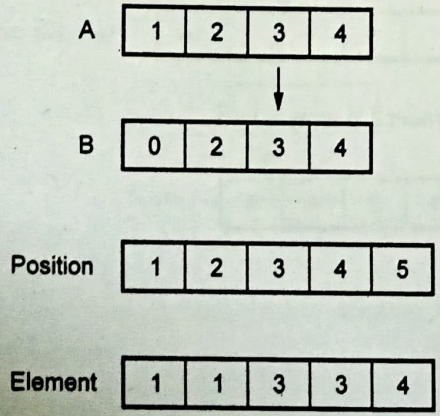
Now decrement 4 in array B by 1



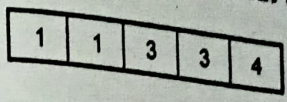
Step 6 : Next element is 1. The position of it in array B is 1.



Step 7 : Next element is 3. In array B its position is 3.



Step 8 : Thus we get sorted list in array Element as



Python Program

```
def countingSort(arr):
    n = len(arr)
    result = [0]*n
    count = [0] *10
    #calculating the count of elements
    i = 0
    for i in range(n):
        count[arr[i]] += 1

    for i in range(1, 10):
        count[i] = count[i]+count[i - 1]
    #placing the elements in sorted order
    i = n - 1
    while i >= 0:
        result[count[arr[i]] - 1] = arr[i]
        count[arr[i]] -= 1
        i = i-1
    #placing back the sorted elements in original
    for i in range(0, n):
        array[i] = result[i]

print("\n Program For Counting Sort")
print("\nHow many elements are there in Array?")
n = int(input())
array = []
i=0
for i in range(n):
    print("\n Enter element in Array")
    item = int(input())
    array.append(item)
```

```
print("Original array is\n")
print(array)

countingSort(array)
print("\n Sorted Array is")
print(array)
```

Output

```
Program For Counting Sort
How many elements are there in Array?
7
 Enter element in Array
5
 Enter element in Array
3
 Enter element in Array
5
 Enter element in Array
1
 Enter element in Array
3
 Enter element in Array
5
 Enter element in Array
4
Original array is
[5, 3, 5, 1, 3, 5, 4]

 Sorted Array is
[1, 3, 3, 4, 5, 5, 5]
>>>
```

Q.22 What is Bucket sort algorithm ?

Ans. : Bucket sort is a sorting technique in which array is partitioned into buckets. Each bucket is then sorted individually, using some other sorting algorithm such as insertion sort.

Algorithm

1. Set up an array of initially empty buckets.
2. Put each element in corresponding bucket.
3. Sort each non empty bucket.
4. Visit the buckets in order and put all the elements into a sequence and print them.

Q.23 Sort the elements using bucket sort. 56, 12, 84, 56, 28, 0, – 13, 47, 94, 31, 12, – 2.

Ans. : We will set up an array as follows

	0	1	2	3	4	5	6	7	8	9	10
Range	–20 to –1	0 to 10	10 to 20	20 to 30	30 to 40	40 to 50	50 to 60	60 to 70	70 to 80	80 to 90	90 to 100

Now we will fill up each bucket by corresponding elements

0	1	2	3	4	5	6	7	8	9	10
–13	0	12	28	31	47	56			84	94
–2		12				56				

Now sort each bucket

0	1	2	3	4	5	6	7	8	9	10
–13	0	12	28	31	47	56			84	94
–2		12				56				

Print the array by visiting each bucket sequentially.

– 13, – 2, 0, 12, 12, 28, 31, 47, 56, 56, 84, 94.

This is the sorted list.

3.7 : Comparison of all Sorting Methods and their Complexities

Q.24 Compare the best case, worst case and average case time complexitites of various sorting algorithms.

Ans. :

Sorting technique	Best case	Average case	Worst case
Insertion sort	O(n)	$O(n^2)$	$O(n^2)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Radix sort	O(nlogn)	O(nlogn)	O(nlogn)
Quick sort	O(nlogn)	O(nlogn)	$O(n^2)$
Heap sort	O(nlogn)	O(nlogn)	O(nlogn)
Shell sort	O(nlogn)	O(n)	O(n)

END...