Code :

```
print("**************************")
print("Name : Ritik Vishwakarma \nRoll no : 1773 \nBatch : B")
print("**************************")
found=False
A=[4,19,26,13,28,1,33,27]
search=int(input("Enter the number to be searched : "))
for i in range(len(A)):
  if (search==A[i]):
    found=True
    print("Number found at index ",i)
    break
if (found==False):
  print("Number not found in the list")
```

Output:

Case 1 (When Number is there in the list) :

```
>>>
**************************
Name : Ritik Vishwakarma
Roll no : 1773
Batch : B
**************************
Enter the number to be searched : 26
Number found at index  2
```

Case 2 (When Number is not there in the list) :

```
>>> ========== RESTART ==========
>>>
**************************
Name : Ritik Vishwakarma
Roll no : 1773
Batch : B
**************************
Enter the number to be searched : 34
Number not found in the list
>>>
```


PRACTICAL NO. 1

Aim :- To search a number from the unsorted list using linear search.

Theory :- The process of identifying or finding a particular record is called searching.

There are two types of search :-

(a) Linear Search

(b) Binary Search

The linear search is further classified as :-

(i) Sorted (ii) Unsorted

Here, we will look on the Unsorted linear search.

Linear search, also known as sequential search, is a process that checks every element in the list sequentially until the desired element is found.

When the elements to be searched are not specifically arranged in ascending or descending order. They are arranged in random manner, that is what it calls unsorted linear search.

Unsorted linear Search :

(1) The data is entered in random manner.

(2) User needs to specify the element to be searched in the list.

(3) The condition is checked that whether entered number is equal to the elements otherwise it proceeds to next elements. If element is not found then appropriate message is displayed.

Date 1-12-19

PRACTICAL NO. 2

Aim :- To search a number from the sorted list using linear search.

Theory :- Searching and Sorting are two different modes or types of data structures.

Sorting - To basically arrange the inputed data in ascending or descending order.

Searching - To search elements and to display the same.

In searching that too in linear sorted search, the data is arranged in ascending or descending order. That is all what it meant by searching through 'sorted' that is well arranged data.

Sorted Linear Search :-

1. The user is supposed to enter the data in sorted manner.
2. User has to give an element for searching through sorted list.
3. The elements are checked whether they are there in the list.
4. In sorted list we can check that element to be searched lies in the between the starting and the ending point of list.

Code :

```
print("*************************")
print("Name : Ritik Vishwakarma \nRoll no : 1773 \nBatch : B")
print("*************************")
found=False
A=[1,4,13,19,26,27,28,33]
search=int(input("Enter the number to be searched : "))
if (search<A[0] or search>A[len(A)-1]):
  print("Number does not exist in the list")
else:
  for i in range(len(A)):
    if (search==A[i]):
      found=True
      print("Number found at index ",i)
      break
  if (found==False):
    print("Number not found in the list")
```

Output :

Case 1 (When number is there in the list) :

```
>>> ========== RESTART ==========
>>>
*************************
Name : Ritik Vishwakarma
Roll no : 1773
Batch : B
*************************
Enter the number to be searched : 27
Number found at index  5
```

Case 2 (When number is out of range of the list) :

```
>>> ========== RESTART ==========
>>>
*************************
Name : Ritik Vishwakarma
Roll no : 1773
Batch : B
*************************
Enter the number to be searched : 55
Number does not exist in the list
>>>
```

```
print("**********")
print("Name : Ritik Vishwakarma \nRoll no : 1773 \nBatch : B")
print("**********")
found=False
A=[1,4,13,19,26,27,28,33]
search=int(input("Enter the number to be searched : "))
l=0
r=len(A)-1
m=int((l+r)/2)
if (search<A[l] or search>A[r]):
  print("Number does not exist in the given range of list")
elif (search==A[l]):
  print("Number found at index ",l)
elif (search==A[r]):
  print("Number found at index ",r)
else:
  while True:
    if (search==A[m]):
      print("Number found at index ",m)
      break
    else:
      if (search<A[m]):
        l=0
        r=m-1
        m=int((l+r)/2)
      else:
        l=m+1
        r=len(A)-1
        m=int((l+r)/2)
if (search!=A[m]):
  print("Number not found in the list")
```

21-12-19

PRACTICAL NO. 3

Aim:- To search a number from the given sorted list using binary search.

Theory:- A binary search also known as a half-interval search, is an algorithm used in computer science to locate a specified value (key) within an array. For the search to be binary, the array must be sorted in either ascending or descending order. At each step of the algorithm a comparison is made and the procedure branches into one of two directions.
Specifically, the key value is compared to the middle element of the array.
If the key value is less than or greater than this middle element, the algorithm knows which half of the array to continue searching in because the array is sorted.
This process is repeated on progressively smaller segments of the array until the value is located.
Because each step in the algorithm divides the array size in half, a binary search will complete successfully in logarithmic time.

Case 1 (When number is there in the list) :

```
>>> ========== RESTART ==========
>>>
**************************
Name : Ritik Vishwakarma
Roll no : 1773
Batch : B
**************************
Enter the number to be searched : 33
Number found at index  7
```

Case 2 (When number is not there in the list) :

```
>>> ========== RESTART ==========
>>>
**************************
Name : Ritik Vishwakarma
Roll no : 1773
Batch : B
**************************
Enter the number to be searched : 34
Number does not exist in the given range of list
>>>
```

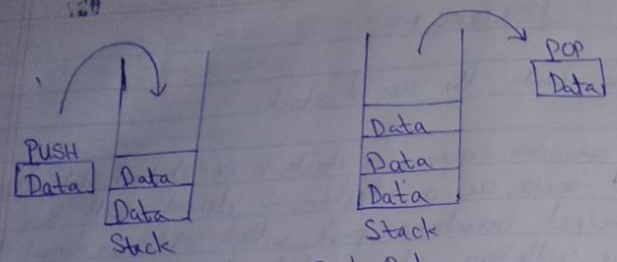
```
class stack:
  global tos
  def __init__(self):
    self.l=[0,0,0,0,0,0,0,0]
    self.tos=-1
  def push(self,data):
    n=len(self.l)
    if self.tos==n-1:
      print("Stack is full")
    else:
      self.tos=self.tos+1
      self.l[self.tos]=data
  def pop(self):
    if self.tos<0:
      print("Stack is empty")
    else:
      k=self.l[self.tos]
      print("Data = ",k)
      self.tos=self.tos-1
s=stack()
s.push(10)
s.push(20)
s.push(30)
s.push(40)
s.push(50)
s.push(60)
s.push(70)
s.push(80)
s.pop()
s.pop()
s.pop()
```


PRACTICAL NO. 5

Aim :- To demonstrate the use of stack

Theory :- In computer science, a stack is an abstract data type that serves as a collection of elements with two principal operations push, which adds an element to the collection and pop, which removes the most recently added element that was not yet removed.
The order may be LIFO (Last In First Out) or FILO (First In Last Out).
Three Basic Operations are performed in the stack

- **Push:** Adds an item in the stack. If the stack is full then it is said to be the overflow condition.
- **Pop :-** Removes an item from the stack, the items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an underflow condition.
- **Peek or Top :-** Returns top element of stack.
- **isempty :-** Returns true if stack is empty else false.



Last In First Out.

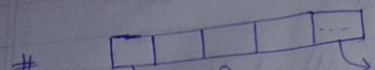
```
s.pop()
s.pop()
s.pop()
s.pop()
s.pop()
>>> ========== RESTART ==========
>>>
************************
Name : Ritik Vishwakarma
Div : A
Roll no : 1773
Batch : B
************************
Stack is full
Data =  70
Data =  60
Data =  50
Data =  40
Data =  30
Data =  20
Data =  10
Stack is empty
>>>
```

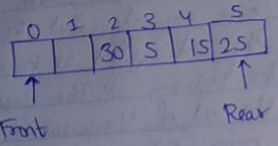
```
class queue:
  global r
  global f
  def __init__(self):
    self.r=0
    self.f=0
    self.l=[0,0,0,0,0,0]
  def add(self,data):
    n=len(self.l)
    if self.r<n:
      self.l[self.r]=data
      self.r=self.r+1
    else:
      print("Queue is full")
  def remove(self):
    n=len(self.l)
    if self.f<n:
      print(self.l[self.f])
      self.f=self.f+1
    else:
      print("Queue is empty")
q=queue()
q.add(30)
q.add(40)
q.add(50)
q.add(60)
q.add(70)
q.add(80)
q.remove()
q.remove()
```


PRACTICAL NO. 6

Aim:- To demonstrate queue add and delete.

Theory:- Queue is a linear data structure where the first element is inserted from one end called REAR and deleted from the other end called as FRONT.
FRONT points to the begining of the queue and REAR points to the end of the queue.
Queue follows the FIFO (First In First Out) structure.
According to its FIFO structure, element inserted first will also be removed first.
In a queue, one end is always used to insert data (enqueue) and the other is used to delete data (dequeue) because queue is open at both of its ends.
enqueue() can be termed as add() in queue i.e. adding an element in queue.
dequeue() can be termed as delete() or remove i.e. deleting or removing of element.
FRONT is used to get the front data item from a queue.
REAR is used to get the last item from a queue.





Front = 2
Rear = 5


```
q.remove()
q.remove()
q.remove()
q.remove()
>>> ========== RESTART ==========
>>>
**********
Name : Ritik Vishwakarma
Div : A
Roll no : 1773
Batch : B
**********
30
40
50
60
70
80
Queue is empty
>>>
```

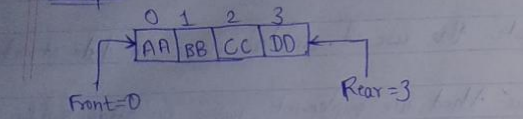
```
class queue:
  global r
  global f
  def __init__(self):
    self.r=0
    self.f=0
    self.l=[0,0,0,0,0,0]
  def add(self,data):
    n=len(self.l)
    if self.r<n:
      self.l[self.r]=data
      print("Data added : ",data)
      self.r=self.r+1
    else:
      s=self.r
      self.r=0
      if self.r<self.f:
        self.l[self.r]=data
        self.r=self.r+1
      else:
        self.r=s
        print("Queue is full")
  def remove(self):
    n=len(self.l)
    if self.f<=n:
      print("Data removed : ",self.l[self.f])
      self.f=self.f+1
    else:
      s=self.f
      self.f=0
      if self.f<self.r:
```

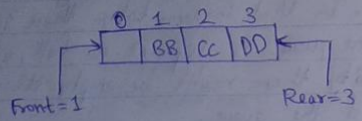

PRACTICAL NO. 7

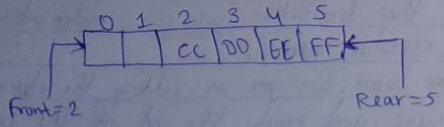
Aim:- To demonstrate the use of circular queue in data structure

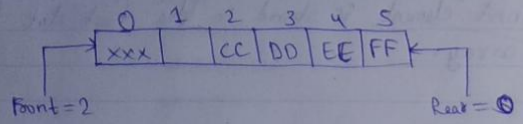
Theory:- The queue that we implement using an array suffer from one limitation. In that implementation there is a possibility that the queue is reported as full, even though in actuality there might be empty slots at the begining of the queue.
To overcome this limitation we can implement queue as circular queue.
In circular queue, we go on adding the elements to the queue and reach the end of the array. The next element is stored in the first slot of the array.

Example:










```
            print(self.l[self.f])
            self.f=self.f+1
        else:
            print("Queue is empty")
            self.f=s
q=queue()
q.add(15)
q.add(25)
q.add(35)
q.add(45)
q.add(55)
q.add(65)
q.remove()
q.add(75)
>>> ========== RESTART ==========
>>>
**************************
Name : Ritik Vishwakarma
Div : A
Roll no : 1775
Batch : B
**************************
Data added :  15
Data added :  25
Data added :  35
Data added :  45
Data added :  55
Data added :  65
Data removed :  15
>>>
```

```
print("**********************")
print("Name : Ritik Vishwakarma \nDiv : A \nRoll no : 1773 \nBatch : B")
print("**********************")
def evaluate(s):
    k=s.split()
    n=len(k)
    stack=[]
    for i in range(n):
        if k[i].isdigit():
            stack.append(int(k[i]))
        elif k[i]=='+':
            a=stack.pop()
            b=stack.pop()
            stack.append(int(b)+int(a))
        elif k[i]=='-':
            a=stack.pop()
            b=stack.pop()
            stack.append(int(b)-int(a))
        elif k[i]=='*':
            a=stack.pop()
            b=stack.pop()
            stack.append(int(b)*int(a))
        else:
            a=stack.pop()
            b=stack.pop()
            stack.append(int(b)/int(a))
    return stack.pop()
s=input("Enter a postfix with spaces between each element : ")
r=evaluate(s)
print("The evaluated value is ",r)
```

```
>>> ========== RESTART ==========
>>>
**********
Name : Ritik Vishwakarma
Div : A
Roll no : 1773
Batch : B
**********
Enter a postfix with spaces between each element : 8 6 9 * +
The evaluated value is  62
>>>
```


PRACTICAL NO. 8

Aim:- To evaluate postfix expression using stack.

Theory:- Stack is an (ADT) and works on LIFO (Last In First Out) i.e. PUSH and POP operations.

A postfix expression is an collection of operators and operands in which the operator is placed after the operands.

Steps to be followed:

1. Read all the symbols one by one from left to right in the given postfix expression.
2. If the reading symbol is operand then push it on to the stack.
3. If the reading symbol is operator (+, −, *, /, etc.) then perform TWO pop operations and store the two popped operands in two different variables (operand 1 and operand 2). Then perform reading symbol operator using operand 1 and operand 2 and push result back on to the stack.
4. Finally, perform a pop operation and display the popped value as final result.

4/2

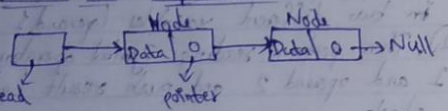
PRACTICAL NO. 9

Aim: To demonstrate the use of linked list in data structure.

Theory: A linked list is a sequence of data structures linked list is a sequence of links which contains items. Each link contains a connection to another link.

- **LINK:** Each link of a linked list can store a data called an element.
- **NEXT:** Each link of a linked list contains a link to the next link called NEXT.
- **LINKED LIST:** A linked list contains the connection link to the first link called First.

LINKED LIST Representation:-



Types of Linked List:-

- Simple.
- Doubly
- Circular.

Basic Operations

- Insertion.
- Deletion.
- Display

```
print("************************")
print("Name : Ritik Vishwakarma \nDiv : A \nRoll no : 1773 \nBatch : B")
print("************************")
class node:
    global data
    global next
    def __init__(self,item):
        self.data=item
        self.next=None
class linkedlist:
    global s
    def __init__(self):
        self.s=None
    def addL(self,item):
        newnode=node(item)
        if self.s==None:
            self.s=newnode
        else:
            head=self.s
            while head.next!=None:
                head=head.next
            head.next=newnode
    def addB(self,item):
        newnode=node(item)
        if self.s==None:
            self.s=newnode
        else:
            newnode.next=self.s
            self.s=newnode
    def display(self):
        head=self.s
        while head.next!=None:
            print(head.data)
            head=head.next
        print(head.data)
```

```
start=linkedlist()
start.addL(50)
start.addL(60)
start.addL(70)
start.addL(80)
start.addB(40)
start.addB(30)
start.addB(20)
start.display()
>>> ========== RESTART ==========
>>>
**************************
Name : Ritik Vishwakarma
Div : A
Roll no : 1773
Batch : B
**************************
20
30
40
50
60
70
80
>>>
```


- Search
- Delete.

PRACTICAL NO. 10

Aim:- To sort given random data by using bubblesort.

Theory:- Sorting is an operation in which any random data is sorted i.e. arranged in ascending or descending order.
Bubblesort sometimes refered to as a sinking sort.
It is a simple sorting algorithm that repeatedly traverses through the list, compares adjacent elements and swaps them if they are in wrong order.
The pass through the list is repeated until the list is sorted.
The algorithm which is a comparison sort is named for the way smaller or larger "bubble" to the top of the list. Although the algorithm is simple, it is too slow as it compares one element, checks if condition fails then only swaps otherwise goes on.


```
print("************************")
print("Name : Ritik Vishwakarma \nDiv : A \nRoll no : 1773 \nBatch : B")
print("************************")
a=[4,33,19,26,73,71]
print("The elements before sorting are : ",a)
for i in range(len(a)-1):
  for j in range(len(a)-1-i):
    if (a[j]>a[j+1]):
      temp=a[j]
      a[j]=a[j+1]
      a[j+1]=temp
print("The elements after sorting are : ",a)
```

```
>>> ========== RESTART ==========
>>>
************************
Name : Ritik Vishwakarma
Div : A
Roll no : 1773
Batch : B
************************
The elements before sorting are :  [4, 33, 19, 26, 73, 71]
The elements after sorting are :  [4, 19, 26, 33, 71, 73]
>>>
```

```
def quickSort(alist):
    quickSortHelper(alist,0,len(alist)-1)
def quickSortHelper(alist,first,last):
    if first<last:
        splitpoint=partition(alist,first,last)
        quickSortHelper(alist,first,splitpoint-1)
        quickSortHelper(alist,splitpoint+1,last)
def partition(alist,first,last):
    pivotvalue=alist[first]
    leftmark=first+1
    rightmark=last
    done=False
    while not done:
        while leftmark<=rightmark and alist[leftmark]<=pivotvalue:
            leftmark=leftmark+1
        while alist[rightmark]>=pivotvalue and rightmark>=leftmark:
            rightmark=rightmark-1
        if rightmark<leftmark:
            done=True
        else:
            temp=alist[leftmark]
            alist[leftmark]=alist[rightmark]
            alist[rightmark]=temp
    temp=alist[first]
    alist[first]=alist[rightmark]
    alist[rightmark]=temp
    return rightmark
print("************************")
print("Name : Ritik Vishwakarma \nDiv : A \nRoll no : 1773 \nBatch : B")
print("************************")
alist=[]
```


PRACTICAL NO. 11

Aim :- To sort the given data in Quick Sort.

Theory: QuickSort is an efficient sorting algorithm.
It is a type of a Divide and Conquer algorithm.
It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quick sort that pick pivot in different ways.
(1) Always pick first element as pivot.
(2) Always pick last element as pivot.
(3) Pick a random element as pivot.
(4) Pick median as pivot.
The key process in quicksort is partition().
Target of partition is, given an array and an element x of array as pivot, put x at it's correct position in sorted array and put all smaller elements (smaller than x) before x and put all greater elements (greater than x) after x. All this should be done in linear time.

ME

PRACTICAL NO. 12

Aim :- To sort the numbers by using selection sort.

Theory: The selection sort algorithm sorts an array by repeatedly find the minimum element (considering ascending

```
n=int(input("Enter how many number you want to input for sorting :"))
print("Enter the elements in unsorted manner --->")
for i in range(0,n):
  alist.append(int(input("Enter element :")))
print("Elements before sorting are : ")
print(alist)
quickSort(alist)
print("Elements after sorting are : ")
print(alist)
```

```
>>> ================================ RESTART ================================
>>>
*************************
Name : Ritik Vishwakarma
Div : A
Roll no : 1773
Batch : B
*************************
Enter how many number you want to input for sorting :6
Enter the elements in unsorted manner --->
Enter element :10
Enter element :13
Enter element :11
Enter element :19
Enter element :18
Enter element :21
Elements before sorting are
[10, 13, 11, 19, 18, 21]
Elements after sorting are :
[10, 11, 13, 18, 19, 21]
>>>
```

```
print("**********")
print("Name : Ritik Vishwakarma \nDiv : A \nRoll no : 1773 \nBatch : B")
print("**********")
a=[4,33,19,26,73,71]
print("The elements before sorting are : ",a)
for i in range(len(a)):
  for j in range(len(a)):
    if (a[i]<a[j]):
      min=a[i]
      a[i]=a[j]
      a[j]=min
print("The elements after sorting are : ",a)
```

```
>>> ========== RESTART ==========
>>>
**********
Name : Ritik Vishwakarma
Div : A
Roll no : 1773
Batch : B
**********
The elements before sorting are :  [4, 33, 19, 26, 73, 71]
The elements after sorting are :  [4, 19, 26, 33, 71, 73]
>>>
```

PRACTICAL NO: 12

Aim: To sort the numbers by using selection sort.

Theory: The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the begining.
The algorithm maintains two subarrays in a given array :-
(1) The subarray which is already sorted.
(2) Remaining subarray which is unsorted.
In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

PRACTICAL NO. 13

Aim:- Write a program based on binary tree.

Theory: Binary tree is a tree which supports maximum of two child for any node within the tree, thus any particular node can have either 0 or 1 or 2 child.
Leaf Node: Nodes which do not have any child.
Internal Node: Nodes which are non-leaf nodes.
Traversing can be defined as a process of visiting every node of the tree exactly once.

Preorder: (1) Visit the root node.
(2) Traverse the left subtree. The subtree in turn might have left and right subtrees.
(3) Traverse the right subtree. The right subtree inturn might have left and right subtrees.

Inorder: (1) Traverse the left subtree. The left subtree inturn might have left and right subtree.
(2) Visit the root node.
(3) Traverse the right subtree. The right subtree inturn might have left and right subtrees.


```
class node:
  global r
  global l
  global data
  def __init__(self,l):
    self.l=None
    self.data=l
    self.r=None
class tree:
  global root
  def __init__(self):
    self.root=None
  def add(self,val):
    if self.root==None:
      self.root=node(val)
    else:
      newnode=node(val)
      h=self.root
      while True:
        if newnode.data<h.data:
          if h.l!=None:
            h=h.l
          else:
            h.l=newnode
            print(newnode.data," added on left of ",h.data)
            break
        else:
          if h.r!=None:
            h=h.r
```

```
                else:
                    h.r=newnode
                    print(newnode.data," added on right of ",h.data)
                    break
        def preorder(self,start):
            if start!=None:
                print(start.data)
                self.preorder(start.l)
                self.preorder(start.r)
        def inorder(self,start):
            if start!=None:
                self.inorder(start.l)
                print(start.data)
                self.inorder(start.r)
        def postorder(self,start):
            if start!=None:
                self.postorder(start.l)
                self.postorder(start.r)
                print(start.data)
print("Name : Ritik Vishwakarma")
t=tree()
t.add(120)
t.add(98)
t.add(130)
t.add(85)
t.add(30)
t.add(45)
t.add(73)
t.add(88)
```


Postorder: (1) Traverse the left subtree. The left subtree inturn might have left and right subtrees.
(2) Traverse the right subtree. The right subtree inturn might have left and right subtrees.
(3) Visit the root node.

```
t.add(15)

t.add(12)

print("Preorder of given tree is :")

t.preorder(t.root)

print("Inorder of given tree is :")

t.inorder(t.root)

print("Postorder of given tree is :")

t.postorder(t.root)
```

```
Name : Ritik Vishwakarma
98  added on left of  120
130  added on right of  120
85  added on left of  98
30  added on left of  85
45  added on right of  30
73  added on right of  45
88  added on right of  85
15  added on left of  30
12  added on left of  15
Preorder of given tree is :
120
98
85
30
15
12
45
73
88
130
Inorder of given tree is :
12
15
30
45
73
85
88
98
120
130
Postorder of given tree is :
12
15
73
45
30
88
85
98
130
120
>>>
```

```
print("*************************")
print("Name : Ritik Vishwakarma\nRoll no : 1773\nDiv : A\nBatch : B")
print("*************************")
def merge(list,l,m,r):
  n1=m-l+1
  n2=r-m
  L=[0]*(n1)
  R=[0]*(n2)
  for i in range(0,n1):
    L[i]=list[l+i]
  for j in range(0,n2):
    R[j]=list[m+1+j]
  i=0
  j=0
  k=l
  while(i<n1 and j<n2):
    if(L[i]<=R[j]):
      list[k]=L[i]
      i+=1
    else:
      list[k]=R[j]
      j+=1
    k+=1
  while(i<n1):
    list[k]=L[i]
    i+=1
    k+=1
  while(j<n2):
    list[k]=R[j]
    j+=1
    k+=1
```


Practical NO. 14

Aim: To sort the numbers using mergesort.

Theory: Merge sort uses the divide and conquer technique. In merge sort we divide the list into nearly equal sublist each of which are then again divided into two sublists.

- We continue this procedure until there is only one element in the individual sublist.
- Once we have individual elements in the sublists, we consider these sublist as sub problems which can be solved. Since there is only one element in the sublist, the sublist is already sorted. So now we merge the subproblems.
- Now while merging the sub problems, we compare the two sublist and create the combined list by comparing the element of the sublist. After comparison we place the element in their correct position in the original sublists.

```
def mergesort(list,l,r):
    if(l<r):
        m=int((l+(r-1))/2)
        mergesort(list,l,m)
        mergesort(list,m+1,r)
        merge(list,l,m,r)
list=[12,11,13,5,6,7]
n=len(list)
print("Given unsorted numbers are : ")
for i in range(n):
    print(list[i],"\t")
mergesort(list,0,n-1)
print("Sorted numbers are : ")
for i in range(n):
    print(list[i],"\t")
```

```
>>> ========== RESTART ==========
>>>
*************************
Name : Ritik Vishwakarma
Roll no : 1773
Div : A
Batch : B
*************************
Given unsorted numbers are :
12
11
13
5
6
7
Sorted numbers are :
5
6
7
11
12
13
>>>
```