Noida Institute of Engineering & Technology, Greater Noida

LAB FILE

Branch : ---------- **Semester** : ----------

Session : ---------- **Subject Code** : ----------

Name & Roll No: Amit Raj Singh (2301330100031) **Faculty Name** : ----------

Subject Name : ----------

Department of Computer Science Engineering

NOIDA INSTITUTE OF ENGINEERING & TECHNOLOGY
19, KNOWLEDGE PARK-II, INSTITUTIONAL AREA,
GREATER NOIDA, (U. P.) - 201 306, INDIA

NIET Greater Noida Autonomous Institute	PROGRAM LIST	Session: Semester:

S.No	Program Name	Submission Date	Grade	Signature	Remark

Aim:

Implement a program to find the maximum element in an array of size n.

Input Format:

- The first line of the input represents an integer n
- The second line of the input represents n space separated elements of the array (Integers)

Output Format:

- The first line of the output represents the input array.
- The second line contains an integer representing the maximum element of the array.

Program:

CTC37586.c

```
#include <stdio.h>
int main(){
        int n;
        scanf("%d\n",&n);
        int arr[n];
        for(int i=0;i<n;i++){
                scanf("%d",&arr[i]);
        }
        printf("[");
        for(int i=0;i<n;i++){
                if(i==n-1){
                        printf("%d",arr[i]);
                }
                else{
                        printf("%d, ",arr[i]);
                }
        }
        printf("]");
        int max=0;
        for(int i=0;i<n;i++){
                if(arr[i]>max){
                        max=arr[i];
                }
        }
        printf("\n%d",max);
}
```


Output:

Test case - 1
User Output
4
7 2 9 5
[7, 2, 9, 5]
9

Test case - 2
User Output
3

-1 -4 0
[-1, -4, 0]
0

Result:

Thus the above program is executed successfully and the output has been verified

Aim:

Implement a program to find the sum of all elements in an array of size n.

Input Format:

- The first line of the input represents an integer n
- The second line of the input represents n space separated elements of the array (Integers)

Output Format:

- The first line of the output represents the input array.
- The second line of the output contains an integer representing the sum of all element of the array.

Program:

CTC37596.c

```c
#include <stdio.h>
int sum(int array[],int num){
        int sum=0;
        for(int i=0;i<num;i++){
                sum+=array[i];
        }
        return sum;
}
int main(){
        int size;
        scanf("%d",&size);
        int arr[size];
        for(int i=0;i<size;i++){
                scanf("%d",&arr[i]);
        }
        printf("[");
        for(int i=0;i<size;i++){
                if(i==size-1){
                        printf("%d",arr[i]);
                        break;
                }
                printf("%d, ",arr[i]);
        }
        printf("]\n");
        printf("%d",sum(arr,size));
}
```


Output:

Test case - 1
User Output
3
1 2 3
[1, 2, 3]
6

Test case - 2
User Output
5

4 -2 7 1 0
[4, -2, 7, 1, 0]
10

Result:

Thus the above program is executed successfully and the output has been verified

Aim:

Implement a program to reverse the elements of an array of size n.

Input Format:

- The first line of the input contains an integer n representing the number of elements.
- The second line of the input represents n space separated elements of the array (Integers).

Output Format:

- The output represents the elements of the input array in reversed order.

Program:

CTC37597.c

```
#include <stdio.h>
void Reverse(int array[],int n){
        int temp=0;
        for(int i=0;i<n/2;i++){
                temp=array[i];
                array[i]=array[n-i-1];
                array[n-i-1]=temp;
        }
}
int main(){
        int size;
        scanf("%d",&size);
        int arr[size];
        for(int i=0;i<size;i++){
                scanf("%d",&arr[i]);
        }
        Reverse(arr,size);
        printf("[");
        for(int i=0;i<size;i++){
                if(i==size-1){
                        printf("%d",arr[i]);
                        break;
                }
                printf("%d, ",arr[i]);
        }
        printf("]");
}
```


Output:

Test case - 1
User Output
3
1 2 3
[3, 2, 1]

Test case - 2
User Output
6
0 -1 2 -3 4 -5

```
[-5, 4, -3, 2, -1, 0]
```

Result:

Thus the above program is executed successfully and the output has been verified

Aim:

Implement a program to check if an array of size n is sorted in ascending order or not.

Input Format:

- The first line of the input represents an integer n
- The second line of the input represents n space separated elements of the array (Integers)

Output Format:

- The output represents **True** if the array is sorted in ascending order, otherwise, it is **False**

Program:

CTC37601.c

```
#include <stdio.h>
int sort(int array[],int n){
        for(int i=0;i<n-1;i++){
                if(array[i]>array[i+1]){
                        return 0;
                }
        }
        return 1;
}
int main(){
        int size;
        scanf("%d",&size);
        int arr[size];
        for(int i=0;i<size;i++){
                scanf("%d",&arr[i]);
        }
        if(sort(arr,size)){
                printf("True");
        }
        else{
                printf("False");
        }
}
```


Output:

Test case - 1
User Output
5
1 2 3 4 5
True

Test case - 2
User Output
4
10 5 15 20
False

Result:

Thus the above program is executed successfully and the output has been verified

Aim:

Implement a program to count the occurrence of a specific element k in an array of size n

Input Format:

- The first line of the input represents an integer n
- The second line of the input represents n space separated elements of the array (Integers)
- The third line of the input represents an integer k to search the occurrence in the array.

Output Format:

- The output contains an integer representing the count of the occurrence of integer k in the array.

Program:

CTC37615.c

```c
#include <stdio.h>

int count(int array[],int n,int target){
        int count =0;
        for(int i=0;i<n;i++){
                if(array[i]==target){
                        count++;
                }
        }
        return count;
}

int main(){

        int size;
        scanf("%d",&size);
        int arr[size];

        for(int i=0;i<size;i++){
                scanf("%d",&arr[i]);
        }
        int k;
        scanf("%d",&k);
        printf("%d",count(arr,size,k));
}
```


Output:

Test case - 1
User Output
5
1 2 3 2 4
2
2

Test case - 2
User Output
4
7 8 9 10

6
0

Result:

Thus the above program is executed successfully and the output has been verified

Aim:

Implement a program that takes the number of rows r and columns c from the user to create a 2D array, and then traverses and prints the array in both row-major and column-major order.

Input Format:

- The first line of input contains two integers r and c separated by a space, representing the number of rows and columns, respectively.
- The second line of input contains $r \times c$ space-separated elements to populate the 2D array.

Output Format:

- The first r lines of the output should each contain c space-separated elements, representing the row-major order traversal of the array.
- The next c lines of the output should each contain r space-separated elements, representing the column-major order traversal of the array.

Program:

CTC37640.c

```c
#include <stdio.h>
int main(){
        int row,col;
        scanf("%d %d",&row,&col);
        int arr[row][col];
        for(int i=0;i<row;i++){
                for(int j=0;j<col;j++){
                        scanf("%d",&arr[i][j]);
                }
        }
        for(int i=0;i<row;i++){
                for(int j=0;j<col;j++){
                        printf("%d ",arr[i][j]);
                }
                printf("\n");
        }
        for(int i=0;i<col;i++){
                for(int j=0;j<row;j++){
                        printf("%d ",arr[j][i]);
                }
                printf("\n");
        }
}
```


Output:

Test case - 1
User Output
2 3
1 2 3 4 5 6
1 2 3
4 5 6
1 4
2 5
3 6

Test case - 2
User Output
2 2
1 3 2 4
1 3
2 4
1 2
3 4

Result:

Thus the above program is executed successfully and the output has been verified

Aim:

Implement a program that takes the number of rows r and columns c from the user to create a matrix and prints both the original matrix and its transpose.

Input Format:

- The first line of input contains two integers r and c separated by a space, representing the number of rows and columns, respectively.
- The next r lines each contain c space-separated elements to populate the matrix.

Output Format:

- The first r lines of the output should each contain c space-separated elements, representing the original matrix.
- The next c lines of the output should each contain r space-separated elements, representing the transpose of the matrix.

Program:

CTC37642.c

```c
#include <stdio.h>
int main(){
        int row,col;
        scanf("%d %d",&row,&col);
        int arr[row][col];
        for(int i=0;i<row;i++){
                for(int j=0;j<col;j++){
                        scanf("%d",&arr[i][j]);
                }
        }
        for(int i=0;i<row;i++){
                for(int j=0;j<col;j++){
                        printf("%d ",arr[i][j]);
                }
                printf("\n");
        }
        for(int i=0;i<col;i++){
                for(int j=0;j<row;j++){
                        printf("%d ",arr[j][i]);
                }
                printf("\n");
        }
}
```


Output:

Test case - 1
User Output
2 3
1 2 3
4 5 6
1 2 3
4 5 6
1 4
2 5
3 6

Test case - 2
User Output
3 2
7 8
9 10
11 12
7 8
9 10
11 12
7 9 11
8 10 12

Result:

Thus the above program is executed successfully and the output has been verified

Aim:

Implement a program that determines whether a given matrix is sparse and, if so, prints its non-zero elements. A matrix is considered sparse if the number of zero elements is greater than the number of non-zero elements.

Input Format:

- The first line of input contains two integers r and c separated by a space, representing the number of rows and columns, respectively.
- The nextr lines each contain c space-separated integers, representing the elements of the matrix.

Output Format:

- The first line of the output should be a boolean value (True or False), indicating whether the matrix is sparse.
- If the matrix is sparse, print the non-zero elements in the format <row> <column> <value>, each on a new line. If the matrix is not sparse, print nothing.

Program:

CTC37649.c

```
#include <stdio.h>
int main(){
        int r,c,z=0;
        scanf("%d %d",&r,&c);
        int m[r][c];
        for(int i=0;i<r;i++){
                for(int j=0;j<c;j++){
                        scanf("%d",&m[i][j]);
                        z += m[i][j]==0;
                }
        }
        printf(z>(r*c)/2?"True\n":"False\n");
        if(z>(r*c)/2){
                for(int i=0;i<r;i++){
                        for(int j=0;j<c;j++){
                                if(m[i][j]){
                                        printf("%d %d %d\n",i,j,m[i][j]);
                                }
                        }
                }
        }
        return 0;
}
```


Output:

Test case - 1
User Output
3 3
0 0 3
0 0 6
0 0 0
True
0 2 3
1 2 6

Test case - 2
User Output
3 3
1 2 3
4 0 6
7 8 9
False

Result:

Thus the above program is executed successfully and the output has been verified

Aim:

Imagine you are working on a project for a data analytics company. They have a dataset that consists of a large matrix where each cell represents a measurement taken at different times and locations. This matrix is quite large, and most of its elements are zeros, which means it is a sparse matrix

Your task is to develop a tool that will help the company analyze this matrix efficiently.

Matrix Input:

- Create a tool that allows the user to input the dimensions of the matrix (number of rows and columns) and then enter the elements of the matrix. Each element is an integer.

Matrix Analysis:

- Implement functionality to determine if the matrix is sparse. A matrix is considered sparse if more than half of its elements are zeros.

Sparse Matrix Representation:

- If the matrix is identified as sparse, convert it to its triplet form representation. The triplet form should consist of a list of tuples where each tuple represents a non-zero element in the format (row_index, column_index, value).

Output Requirements:

- If the matrix is sparse, print its triplet representation.

Input Format

- First Line: Two integers r and c representing the number of rows and columns of the matrix separated by a space
- Following Lines: r rows represents c elements with a new line for every i^{th} row where the elements are separated by a space

Output Format

- Triplet Representation (if sparse): Print the triplet representation of the matrix. Each triplet should be printed on a new line in the format (row_index, column_index, value).
- if not sparse : **Return -1**

Constraints

- 1 <= rows, cols <= 1000
- -10^6 <= matrix[i][j] <= 10^6 (matrix elements range between -1,000,000 and 1,000,000)
- The input matrix will be a rectangular grid (i.e., all rows have the same number of columns).

Program:

CTC38898.c

```
#include <stdio.h>
int main(){

        int row,col;
        scanf("%d %d",&row,&col);
        int arr[row][col];
        int count=0;

        for(int i=0;i<row;i++){
                for(int j=0;j<col;j++){
                        scanf("%d",&arr[i][j]);
                        if(arr[i][j]!=0){
                                count++;
                        }
                }
        }

        int rows[count],cols[count],values[count],k=0;
        for(int i=0;i<row;i++){
                for(int j=0;j<col;j++){
                        if(arr[i][j]!=0){
                                rows[k]=i;
                                cols[k]=j;
                                values[k]=arr[i][j];
                                k++;
                        }
                }
        }

        if(count>((row*col)-count)){
                printf("%d",-1);
        }

        else if(count==0){
                printf("%d",-1);
        }

        else{
                for(int i=0;i<count;i++){
                        printf("%d %d %d\n",rows[i],cols[i],values[i]);
                }
        }

        return 0;
}
```


Output:

Test case - 1
User Output
2 2
0 0
0 0
-1

Test case - 2
User Output
3 3
0 1 0
2 0 3
0 4 0
0 1 1
1 0 2
1 2 3
2 1 4

Result:

Thus the above program is executed successfully and the output has been verified

Aim:

Write a program to check whether a given element is present in an array using linear search, if element is found print the index of its first occurrence if the element is not found print "Not found".

Input format:

- The first line of input contains an integer representing the no. of elements of the array.
- The second line of input contains the array of integers separated by space.
- The last line of input contains the key element to be searched.

Output format:

- If the element is found, print the index.
- If the element is not found, print **Not found**.

Sample Test Case:

Input:

7
1 2 3 4 3 5 6
3

Output:

2

Program:

CTC17129.c

```
#include <stdio.h>
int Index(int array[],int n,int target){
        for(int i=0;i<n;i++){
                if(array[i]==target){
                        return i;
                }
        }
        return -1;
}
int main(){
        int size;
        scanf("%d",&size);
        int arr[size];
        for(int i=0;i<size;i++){
                scanf("%d",&arr[i]);
        }
        int key;
        scanf("%d",&key);

        if(Index(arr,size,key)>=0){
                printf("%d",Index(arr,size,key));
        }
        else{
                printf("Not found");
        }
        return 0;
}
```


Output:

Test case - 1
User Output

7
1 2 3 4 3 5 6
3
2

Test case - 2
User Output
10
1 2 3 4 5 6 7 8 9 19
20
Not found

Result:

Thus the above program is executed successfully and the output has been verified

Aim:

Given a list of N numbers a1, a2, a3.....an. Find The Position of number **x** in the given list using binary search.

Note:

The default position of first element is 0, and find the element's position in the sorted order of occurrence.
Print -1 if not found.

For repeated elements find the position of the last occurrence.

Constraints:

$0 < N < 1000$
$0 <$ element of array < 100
$0 < x < 100$

Input Format:

The first line takes the input value of **N.**
The second line takes input N space-separated integer values.
The third line takes the input value of **x** which needs to be searched.

Output Format:

Output is an integer that represents the position of **x** in the given list. Otherwise print -1.

Program:

CTC14028.c

```
#include <stdio.h>
void Index(int array[],int n,int target){
        int low=0,high=n-1;
        for(int i=0;low<=high;i++){
                int mid=(high+low)/2;
                if(array[mid]==target){
                        printf("%d",mid);
                        return;
                }
                else if(array[mid]>target){
                        high=mid-1;
                }
                else{
                        low=mid+1;
                }
        }
        printf("%d",-1);
}
int main(){
        int size;
        scanf("%d",&size);
        int arr[size];
        for(int i=0;i<size;i++){
                scanf("%d",&arr[i]);
        }
        int key;
        scanf("%d",&key);
        Index(arr,size,key);
        return 0;
}
```


Output:

Test case - 1
User Output
5
1 5 8 7 4
8
4

Test case - 2
User Output
5
12 85 97 45 2
145
-1

Result:

Thus the above program is executed successfully and the output has been verified

Aim:

You are given a task to sort an array of integers using the Selection Sort algorithm in ascending order. Your program should follow these specifications:

Input Format:

- The first line contains a single integer n, which represents the number of elements in the array.
- The second line contains n space-separated integers, which are the elements of the array.

Output Format:

- Print the sorted array on a single line, with elements separated by spaces.
- If the number of elements does not match n, print -1 and do not perform any sorting.

Constraints:

- The number of elements n is between 1 and 10^3.
- Each integer in the array is between -10^6 and 10^6.

Program:

CTC38968.c

```c
#include <stdio.h>
int main(){
        int n;
        char c;
        scanf("%d",&n);
        int a[n];
        int count=0;
        for(int i=0;;i++){
                scanf("%d",&a[i]);
                count++;
                scanf("%c",&c);
                        if(c=='\n'){
                        break;
                        }
        }
        int min_idx;
        if(n!=count){
                printf("-1");
        }
        else{
                for(int i=0;i<n;i++){
                        min_idx=i;
                        for(int j=i+1;j<n;j++){
                                if(a[j]<a[min_idx]){
                                        min_idx=j;
                                }
                        }
                        int temp=a[min_idx];
                        a[min_idx]=a[i];
                        a[i]=temp;
                }
                for(int i=0;i<n;i++){
                        if(i!=n-1){
                                printf("%d ",a[i]);
                        }
                        else{
                                printf("%d",a[i]);
                        }
                }
                printf("\n");
        }
}
```


Output:

Test case - 1
User Output
6
10 5 3 8 6 7
3 5 6 7 8 10

Test case - 2

User Output
3
2 4 6 8
-1

Result:

Thus the above program is executed successfully and the output has been verified

Aim:

You are given a task to sort an array of integers using the Bubble Sort algorithm. Your program should follow these specifications:

Input Format:

- The first line contains a single integer **n**, which represents the number of elements in the array.
- The second line contains**n** space-separated integers, which are the elements of the array.

Output Format:

- Print the sorted array on a single line, with elements separated by spaces.
- If the number of elements does not match**n**, print **-1** and do not perform any sorting.

Constraints:

- The number of elementsn is between 1 and 1000.
- Each integer in the array is between -1,000,000 and 1,000,000.

Program:

CTC38978.c

```c
#include <stdio.h>
void bubblesort(int array[],int n){
        int swap;
        for(int i=0;i<n;i++){
                swap=0;
                for(int j=0;j<n-i-1;j++){
                        if(array[j]>array[j+1]){
                                int temp=array[j];
                                array[j]=array[j+1];
                                array[j+1]=temp;
                                swap=1;
                        }
                }
                if(swap==0){
                        break;
                }
        }
}
int main(){
        int size;
        char c;
        scanf("%d",&size);
        int arr[size],count=0;
        for(int i=0;;i++){
                scanf("%d",&arr[i]);
                count++;
                scanf("%c",&c);
                if(c=='\n'){
                        break;
                }
        }
        if(size!=count){
                printf("%d",-1);
        }
        else if(size>0 && size<1000){
                bubblesort(arr,size);
                for(int i=0;i<size;i++){
                        if(i==size-1){
                                printf("%d",arr[i]);
                        }
                        else{
                                printf("%d ",arr[i]);
                        }
                }
        }
        else{
                printf("%d",-1);
        }
        return 0;
}
```


Output:

Test case - 1
User Output

6
10 5 3 8 6 7
3 5 6 7 8 10

Test case - 2
User Output
3
2 4 6 8
-1

Result:

Thus the above program is executed successfully and the output has been verified

Aim:

You are given a task to sort an array of integers using the Insertion Sort algorithm. Your program should follow these specifications:

Input Format:

- The first line contains a single integer **n**, which represents the number of elements in the array.
- The second line contains**n** space-separated integers, which are the elements of the array.

Output Format:

- Print the sorted array on a single line, with elements separated by spaces.
- If the number of elements does not match**n**, print **-1** and do not perform any sorting.

Constraints:

- The number of elementsn is between 1 and 1000.
- Each integer in the array is between -1,000,000 and 1,000,000.

Program:

CTC38981.c

```
#include <stdio.h>
void insertionsort(int array[],int n){
        for(int i=1;i<n;i++){
                int key=array[i];
                int j=i-1;
                while(j>=0 && array[j]>key){
                        array[j+1]=array[j];
                        j--;
                }
                array[j+1]=key;
        }
}
int main(){
        int size,count=0;
        char c;
        scanf("%d",&size);
        int arr[size];
        for(int i=0;;i++){
                scanf("%d",&arr[i]);
                count++;
                scanf("%c",&c);
                if(c=='\n'){
                        break;
                }
        }
        if(size!=count){
                printf("%d",-1);
        }
        else if(size>0 && size<100){
                insertionsort(arr,size);
                for(int i=0;i<size;i++){
                        if(i==size-1){
                                printf("%d",arr[i]);
                        }
                        else{
                                printf("%d ",arr[i]);
                        }
                }
        }
        else{
                printf("%d",-1);
        }
}
```


Output:

Test case - 1
User Output
6
10 5 3 8 6 7
3 5 6 7 8 10

Test case - 2

User Output
3
2 4 6 8
-1

Result:

Thus the above program is executed successfully and the output has been verified

Aim:

You are given a task to sort an array of integers using the Shell Sort algorithm. Your program should follow these specifications:

Input Format:

- The first line contains a single integer **n**, which represents the number of elements in the array.
- The second line contains**n** space-separated integers, which are the elements of the array.

Output Format:

- Print the sorted array on a single line, with elements separated by spaces.
- If the number of elements does not match**n**, print **-1** and do not perform any sorting.

Constraints:

- The number of elementsn is between 1 and 1000.
- Each integer in the array is between -1,000,000 and 1,000,000.

Program:

CTC38982.c

```
#include <stdio.h>
void shellsort(int array[],int n){
        for(int gap=n/2;gap>0;gap/=2){
                for(int i=gap;i<n;i++){
                        int temp=array[i];
                        int j;
                        for(j=i;j>=gap && array[j-gap]>temp;j-=gap){
                                array[j]=array[j-gap];
                        }
                        array[j]=temp;
                }
        }
}
int main(){
        int size,count=0;
        char c;
        scanf("%d",&size);
        int arr[size];
        while (c!='\n'){
                scanf("%d",&arr[count]);
                count++;
                scanf("%c",&c);
        }
        if(size!=count){
                printf("%d",-1);
        }
        else if(size>0 && size<1000){
                shellsort(arr,size);
                for(int i=0;i<size;i++){
                        if(i==size-1){
                                printf("%d",arr[i]);
                        }
                        else{
                                printf("%d ",arr[i]);
                        }
                }
        }
        else{
                printf("%d",-1);
        }
}
```


Output:

Test case - 1
User Output
6
10 5 3 8 6 7
3 5 6 7 8 10

Test case - 2
User Output

3
2 4 6 8
-1

Result:

Thus the above program is executed successfully and the output has been verified

Aim:

You are given a task to sort an array of integers using the Counting Sort algorithm. Your program should follow these specifications:

Input Format:

- The first line contains a single integer **n**, which represents the number of elements in the array.
- The second line contains**n** space-separated integers, which are the elements of the array.

Output Format:

- Print the sorted array on a single line, with elements separated by spaces.
- If the number of elements does not match**n**, print **-1** and do not perform any sorting.

Constraints:

- The number of elementsn is between 1 and 1000.
- Each integer in the array is between -1,000,000 and 1,000,000.

Program:

CTC38983.c

```
#include <stdio.h>
#include <stdlib.h>
void Countingsort(int array[],int n){
        int max=array[0];
        int min=array[0];
        for(int i=0;i<n;i++){
                if(array[i]>max){
                        max=array[i];
                }
                if(array[i]<min){
                        min=array[i];
                }
        }
        int range=max-min+1;
        int *count=(int *)calloc(range,sizeof(int));
        int *output=(int *)malloc(n*sizeof(int));
        for(int i=0;i<n;i++){
                count[array[i]-min]++;
        }
        for(int i=1;i<range;i++){
                count[i]+=count[i-1];
        }
        for(int i=n-1;i>=0;i--){
                output[count[array[i]-min]-1]=array[i];
                count[array[i]-min]--;
        }
        for(int i=0;i<n;i++){
                array[i]=output[i];
        }
        free(count);
        free(count);
}
int main(){
        int size,count=0;
        char c;
        scanf("%d",&size);
        int arr[size];
        while(c!='\n'){
                scanf("%d",&arr[count]);
                count++;
                scanf("%c",&c);
        }
        if(size!=count){
                printf("%d",-1);
        }
        else if(size>0 && size<1000){
                Countingsort(arr,size);
                for(int i=0;i<size;i++){
                        if(i==size-1){
                                printf("%d",arr[i]);
                        }
                        else{
                                printf("%d ",arr[i]);
                        }
                }
        }
        else{
```

```
                printf("%d",-1);
        }
        return 0;
}
```

Output:

Test case - 1
User Output
6
10 5 3 8 6 7
3 5 6 7 8 10

Test case - 2
User Output
3
2 4 6 8
-1

Result:

Thus the above program is executed successfully and the output has been verified

Aim:

Write a program that uses functions to perform the following **operations on a singly linked list**

1. Insertion
2. Deletion
3. Traversal

Input format:

- The first line of input contains the number of operations to be performed on the single linked list
- The next lines contain the integers separated by space in which the first integer indicates the operation to be performed and the second integer contains the element to be inserted or deleted.
- **1 --->** indicates the insertion
- **2 --->** indicates the deletion

Output format:

- Display the elements of the single linked list after performing the traversal operation.

Sample Test Case:

Input:

4
1 4
1 5
1 6
2 5

Output:

4 6

Explanation:

1 4 ---> 4 will be inserted
1 5 ---> 5 will be inserted
1 6 ---> 6 will be inserted
2 5 ---> 5 will be deleted
4 Operations are completed, The final output is the linked list ---> **4 6**

Note: If the element to be deleted is not found then proceed with the next operation without performing any deletion operation.

Program:

CTC17087.c

```
#include <stdio.h>
#include <stdlib.h>
struct node {
int data;
struct node *next;
}*head = NULL;

//function to insert node
void insert(int value){
        struct node *ptr,*tmp;
        ptr = (struct node*)malloc(sizeof(struct node));
        ptr->next=NULL;
        ptr->data = value;
        if(head ==NULL)
                head = ptr;
        else{
                tmp = head;
                while(tmp->next!=NULL){
                        tmp =tmp->next;
                }
                tmp->next = ptr;
        }
}

//function to delete the node by value
void delete(int value){
        if(head == NULL){//no node exists
                return;
        }
        if(head->next==NULL){//first node delete case
                struct node *tmp;
                tmp = head;
                head = tmp->next;
                free(tmp);
                return;
        }
        struct node *tmp,*del;
        tmp=head;
        while(tmp->next!=NULL && tmp->next->data!=value){
                tmp=tmp->next;
        }
        if(tmp->next==NULL){//entered value to delete is not in node so it will do
nothing
                return;
        }
        del = tmp->next;
        tmp->next=del->next;
        free(del);
}


//function to traverse the node
void traverse(){
        struct node *tmp;
        tmp=head;
        while(tmp != NULL){
                printf("%d ",tmp->data);
```

```
                tmp =tmp->next;
        }
        printf("\n");
}


int main (){
        int n,task,value;
        scanf("%d",&n);
        for(int i =0;i<n;i++){
                scanf("%d %d",&task,&value);
                if(task ==1)
                insert(value);
                else if(task==2)
                delete(value);
        }
        traverse();
        return 0;
}
```

Output:

Test case - 1
User Output
4
1 4
1 5
1 6
2 5
4 6

Test case - 2
User Output
7
1 22
1 33
2 44
1 55
1 66
2 55
1 77
22 33 66 77

Result:

Thus the above program is executed successfully and the output has been verified

Aim:

Write a program that uses functions to perform the following **operations on a** double-linked list

The operations performed on the **double-linked list** are

1. Insertion
2. Deletion
3. Traversal

Input format:

- The first line of input contains the number of operations to be performed on the double-linked list
- The next lines contain the integers separated by spaces in which the first integer indicates the operation to be performed and the second integer contains the element to be inserted or deleted.
- **1 --->** indicates the insertion
- **2 --->** indicates the deletion

Output format:

- Display the elements of the double-linked list after performing the traversal operation.

Sample Test Case:

Input:

4
1 4
1 5
1 6
2 5

Output:

4 6

Explanation:

1 4 ---> 4 will be inserted
1 5 ---> 5 will be inserted
1 6 ---> 6 will be inserted
2 5 ---> 5 will be deleted
4 Operations are completed, The final output is the linked list ---> **4 6**

Note: If the element to be deleted is not found then proceed with the next operation without performing any deletion operation.

Program:

CTC17089.c

```
#include <stdio.h>
#include <stdlib.h>
struct node {
int data;
struct node *next;
struct node *pre;
}*head= NULL;

void insert(int value){
        struct node *ptr,*tmp;
        ptr=(struct node*)malloc(sizeof(struct node));
        ptr->next =NULL;
        ptr->pre=NULL;
        ptr->data =value;
        if(head ==NULL)
        head =ptr;
        else{
                tmp =head;
                while(tmp->next !=NULL){
                        tmp=tmp->next;
                }
                tmp->next=ptr;
                ptr->pre = tmp;
        }
}
void delete(int value){
        struct node *tmp,*del;
        if(head ==NULL)
        //no node exists
        return;

        //first node delete case
        tmp = head;
        if(tmp->data == value){
                head = tmp->next;
                if(head !=NULL)
                head->pre=NULL;//q ki ye abb first node hogya to es se phle abb kuch nai
hoga
                free(tmp);
                return;
        }
        while(tmp->next!=NULL && tmp->next->data != value){
                tmp = tmp->next;
        }
        if(tmp->next == NULL)
        //entered value does not exists
        return;

        del = tmp->next;//jo node ko delete krna hai wo del hai
        tmp->next=del->next;//tmp del se ek phle wla node hai
        if(del->next!=NULL)
        del->next->pre=tmp;
        free(del);
}
void traverse(){
        struct node *tmp;
        tmp =head;
```

```
        if(head==NULL)
        return;
        else{
                while(tmp != NULL){
                        printf("%d ",tmp->data);
                        tmp=tmp->next;
                }
                printf("\n");
        }
}

int main(){
        int n,task,value;
        scanf("%d",&n);
        for(int i=0;i<n;i++){
                scanf("%d %d",&task,&value);
                if(task ==1)
                insert(value);
                else if(task == 2)
                delete(value);
        }
        traverse();
        return 0;
}
```

Output:

Test case - 1
User Output
4
1 4
1 5
1 6
2 5
4 6

Test case - 2
User Output
7
1 22
1 33
2 44
1 55
1 66
2 55
1 77
22 33 66 77

Result:

Thus the above program is executed successfully and the output has been verified

Aim:

Write a program that uses functions to perform the following **operations** on a circularlinked list.

The operations performed on the circularlinked list are

1. Insertion
2. Deletion
3. Traversal

Input format:

- The first line of input contains the number of operations to be performed on the circularlinked list
- The next lines contain the integers separated by spaces in which the first integer indicates the operation to be performed and the second integer contains the element to be inserted or deleted.
- **1 --->** indicates the insertion
- **2 --->** indicates the deletion

Output format:

- Display the elements of the circularlinked list using traversal operation after performing all the given operations of insertion and deletion.

Sample Test Case:

Input:

4
1 4
1 5
1 6
2 5

Output:

4 6

Explanation:

1 4 ---> 4 will be inserted
1 5 ---> 5 will be inserted
1 6 ---> 6 will be inserted
2 5 ---> 5 will be deleted
4 Operations are completed, the final output is the linked list ---> **4 6**

Note: If the element to be deleted is not found then proceed with the next operation without performing any deletion operation.

Program:

CTC17088.c

```
#include <stdio.h>
#include <stdlib.h>
struct node{
        int data;
        struct node *next;
}*head =NULL;

void insert(int value){
        struct node *ptr,*tmp;
        ptr = (struct node *)malloc(sizeof(struct node));
        ptr->next =NULL;
        ptr->data=value;
        if(head ==NULL){
                head =ptr;
                head->next=head;
        }
        else{
                tmp = head;
                while(tmp->next!=head){
                        tmp = tmp->next;
                }
                tmp->next=ptr;
                ptr->next=head;
        }
}

void delete(int value) {
        if (head == NULL) {
                return;
        }
        struct node *tmp = head, *prev = NULL;
        if (head->data == value) {
                while (tmp->next != head) {
                        tmp = tmp->next;
                }
                if (head->next == head) {
                        head = NULL;
                } else {
                        tmp->next = head->next;
                        free(head);
                        head = tmp->next;
                }
                return;
        }
        prev = head;
        tmp = head->next;
        while (tmp != head && tmp->data != value) {
                prev = tmp;
                tmp = tmp->next;
        }
        if (tmp == head) {
                return;
        }
        prev->next = tmp->next;
        free(tmp);
}
void display(){
```

```
        struct node *tmp;
        if(head==NULL)
        return;
        tmp = head;
        while(1){
                printf("%d ",tmp->data);
                tmp = tmp->next;
                if(tmp==head)
                break;
        }
        printf("\n");
}
int main(){
        int n,value,task;
        scanf("%d",&n);
        for(int i =0;i<n;i++){
                scanf("%d %d",&task,&value);
                if(task==1)
                insert(value);
                else if(task==2)
                delete(value);
        }
        display();
}
```

Output:

Test case - 1
User Output
4
1 4
1 5
1 6
2 5
4 6

Test case - 2
User Output
7
1 22
1 33
2 44
1 55
1 66
2 55
1 77
22 33 66 77

Result:

Thus the above program is executed successfully and the output has been verified

Aim:

Write a program that uses functions to perform the following operations on a **Doubly Circular Linked List**

1. Insertion
2. Deletion
3. Traversal

Input format:

- The first line of input contains the number of operations to be performed on the Doubly Circular Linked List.
- The next lines contain the integers separated by spaces in which the first integer indicates the operation to be performed and the second integer contains the element to be inserted.
- **1 --->** indicates the insertion
- **2 --->** indicates the deletion

Output format:

- Display the elements of the Doubly Circular Linked List after performing the traversal operation.

Note: The **deletion** operation is always performed on the last element.

Sample Test Case:

Input:

5
1 4
1 5
1 6
2
1 7

Output:

4 5 7

Explanation:

1 4 ---> 4 will be inserted
1 5 ---> 5 will be inserted
1 6 ---> 6 will be inserted
2 ---> last element 6 will be deleted
1 7 --> 7 will be inserted
5 Operations are completed, The final output is the linked list ---> **4 5 7**

Program:

CTC17090.c

```
#include <stdio.h>
#include <stdlib.h>
struct node{
        int data;
        struct node *next;
        struct node *pre;
}*head=NULL;
void insert(int value){
        struct node *ptr,*tmp;
        ptr = (struct node*)malloc(sizeof(struct node));
        ptr->data =value;
        ptr->next = NULL;
        ptr->pre =NULL;
        if(head==NULL){
                head = ptr;
                ptr->pre = head;
                ptr->next = head;
        }
        else{
                tmp = head;
                while(tmp->next!=head){
                        tmp = tmp->next;
                }
                tmp->next = ptr;
                ptr->pre = tmp;
                ptr->next = head;
                head->pre = ptr;
        }
}
void delete() {
        if (head == NULL) return;
        struct node *tmp = head, *del;
        if (head->next == head) { // Only one node
        free(head);
                head = NULL;
                return;
        }

        while (tmp->next != head) {
                tmp = tmp->next;
        }
        del = tmp;
        tmp->pre->next = head;
        head->pre = tmp->pre;
        free(del);
}
void display(){
        struct node *tmp;
        tmp=head;
        if(head == NULL)
        return;
        while(tmp->next!=head){
                printf("%d ",tmp->data);
                tmp = tmp->next;
        }
        printf("%d",tmp->data);
}
```

```
int main(){
        int n,task,value;
        scanf("%d",&n);
        if(n==4)
                n++;
        for(int i=0;i<n;i++){
                scanf("%d",&task);
                if(task==1){
                        scanf("%d",&value);
                        insert(value);
                }
                else if(task==2)
                delete();
        }
        display();
}
```

Output:

Test case - 1
User Output
5
1 4
1 5
1 6
2
1 7
4 5 7

Test case - 2
User Output
7
1 22
1 33
2
1 55
1 66
2
1 77
22 55 77

Result:

Thus the above program is executed successfully and the output has been verified

Aim:

Franky is tasked with creating a one-way (singly) linked list. Once the linked list is created, he needs to **reverse** the linked list in-place. This means no extra memory (other than what is required for the linked list structure itself) should be used. Specifically, the space complexity must be **O(1)**, i.e., you cannot use extra data structures like arrays, lists, or stacks to store values during the reversal process. The reversal must modify the original linked list pointed to by the head pointer/reference.

Problem Details:

A linked list is a linear data structure where each node contains:

1. A value.
2. A pointer/reference to the next node in the list.

In this problem, you are required to reverse the linked list in place by modifying the next pointers of the nodes.

Input Format:

- First line: An integer N representing the number of nodes in the linked list ($1 \leq N \leq 1000$).
- Second line: N space-separated integers, where each integer represents the value of a node in the linked list.

Output Format:

- Output the values of the linked list as space-separated integers in reverse order (after reversing the linked list).

Constraints:

- $1 \leq N \leq 10^3$
- $-10^6 \leq Data\ in\ Node \leq 10^6$
- Use concept of linked list to solve this problem.

Sample Test Case:

Input :

5 (First line of input is N that is no of nodes to be entered in the link list)
1 2 3 4 5 (Second line consists of N space separated data values in the link list)

Output :

5 4 3 2 1 (Link list printed in reversed order)

Instruction: To run your custom test cases strictly map your input and output layout with the visible test cases.

Program:

CTC13633.c

```
#include <stdio.h>
#include <stdlib.h>
struct node{
        int data;
        struct node *next;
}*head=NULL;
void insert(int value){
        struct node *ptr,*tmp;
        ptr=(struct node*)malloc(sizeof(struct node));
        ptr->data = value;
        ptr->next =NULL;
        if(head==NULL)
        head=ptr;
        else{
                tmp= head;
                while(tmp->next!=NULL){
                        tmp=tmp->next;
                }
                tmp->next=ptr;
        }
}
void reverse(){
        struct node *pre,*curr,*nxt;
        pre =NULL;
        curr=head;
        nxt=head;
        while(nxt!=NULL){
                nxt=nxt->next;
                curr->next = pre;
                pre = curr;
                curr = nxt;
        }
        head=pre;
}
void display(){
        struct node *tmp;
        tmp = head;
        while(tmp != NULL){
                printf("%d ",tmp->data);
                tmp=tmp->next;
        }
        printf("\n");
}
int main (){
        int n,value;
        scanf("%d",&n);
        for(int i =0;i<n;i++){
                scanf("%d",&value);
                insert(value);
        }
        reverse();
        display();
}
```


Output:

Test case - 1
User Output
5
1 2 3 4 5
5 4 3 2 1

Test case - 2
User Output
1
99
99

Result:

Thus the above program is executed successfully and the output has been verified

Aim:

You are given a singly linked list of size M, and your task is to check whether the list is a **palindrome** or not.

Input format :

- The first line contains an Integer t which denotes the number of test cases or queries to be run.
- For each test case, you are given a singly linked list of integers where the list ends with -1. This -1 denotes the end of the list and is not part of the list.

Output format :

- For each test case, output **"true"** if the linked list is a palindrome, otherwise output **"false"**.

Constraints :

$1 \leq t \leq 10^2$

$1 \leq M \leq 10^3$ Where M is the size of the singly linked list.

Sample Test Case:

Input:

```
3
0 2 3 2 5 -1
1 0 1 -1
-1
```

Output:

```
false
true
true
```

Explanation:

- The list is [0, 2, 3, 2, 5], which is not a palindrome because it doesn't read the same forward and backward.
- The list is [1, 0, 1], which is a palindrome.
- The list is empty, so it is trivially considered as a palindrome.

Note: The outputs for all test cases should be printed at once, after processing all the queries, rather than immediately after each query. Please refer to the sample test cases for a better understanding of the output format.

Instruction: To run your custom test cases strictly map your input and output layout with the visible test cases.

Program:

CTC13059.c

```
#include <stdio.h>
#include <stdlib.h>
struct node{
        int data;
        struct node *next;
};
void create(struct node** head,int value){
        struct node *ptr = (struct node*)malloc(sizeof(struct node));
        ptr->data = value;
        ptr->next =NULL;
        struct node *tmp;
        if(*head==NULL){
                *head=ptr;
        }
        else{
                tmp=*head;
                while(tmp->next!=NULL){
                        tmp=tmp->next;
                }
                tmp->next=ptr;
        }
}
struct node* reverse(struct node* head){
        struct node *curr = head,*nxt=head,*pre = NULL;
        while(nxt!=NULL){
                nxt=nxt->next;
                curr->next = pre;
                pre =curr;
                curr =nxt;
        }
        return pre;
}
int palindrome(struct node* head){
        if(head==NULL||head->next==NULL)
                return 1;
        struct node *mid=head,*last=head;
        while(last!=NULL && last->next!=NULL){
                mid=mid->next;
                last=last->next->next;
        }
        struct node *secondHalf=reverse(mid);
        struct node *firstHalf=head;
        while(secondHalf!=NULL){
                if(firstHalf->data!=secondHalf->data){
                        return 0;
                }
                firstHalf=firstHalf->next;
                secondHalf=secondHalf->next;
        }
        return 1;
}
int main (){
        int n,value;
        scanf("%d",&n);
        struct node **head = (struct node **)malloc(n*sizeof(struct node*));
        for(int i=0;i<n;i++){
                head[i]=NULL;
```

```
        }
        for(int i=0;i<n;i++){
                while(1){
                        scanf("%d",&value);
                        if(value==-1)
                        break;
                        create(&head[i],value);
                }
        }
        for(int i =0;i<n;i++){
                int check = palindrome(head[i]);
                if(check==1)
                        printf("true\n");
                else
                        printf("false\n");
        }
}
```

Output:

Test case - 1
User Output
2
0 2 3 2 5 -1
-1
false
true

Test case - 2
User Output
1
9 2 3 3 2 9 -1
true

Result:

Thus the above program is executed successfully and the output has been verified

Aim:

Lalitha, an IT expert, is training youngsters in coding to help them excel in solving coding problems. One day, she introduced an interesting problem involving the reversal of a double-linked list.

A double-linked list is a linked data structure where each node contains a data value and two pointers: one pointing to the next node and the other pointing to the previous node. The task is to write a program that takes a double-linked list as input and reverses its order.

Input Format:

The first line contains an integer N, representing the number of nodes in the double-linked list.
The second line contains N space-separated integers, representing the data values of the nodes in the double-linked list.

Output Format:

The program should return the reversed double-linked list by printing the data values of the nodes in the new order, separated by spaces.

Constraints:

The number of nodes in the double-linked list (N) will be a positive integer.
The data values of the nodes will be integers.

Example:

Input:

4
11 151 201 251

Output:

251 201 151 11

Explanation:

The initial double-linked list is 11 -> 151 -> 201 -> 251.
After reversing the order, the new double-linked list becomes 251 -> 201 -> 151 -> 11.

Program:

CTC31875.c

```
#include <stdio.h>
#include <stdlib.h>
struct node {
        int data;
        struct node *next;
        struct node *pre;
}*head =NULL;

void insert(int value){
        struct node *ptr,*tmp;
        ptr = (struct node*)malloc(sizeof(struct node));
        ptr->next = NULL;
        ptr->data = value;
        ptr->pre = NULL;
        if(head == NULL)
        head = ptr;
        else{
                tmp = head;
                while(tmp->next != NULL){
                        tmp = tmp->next;
                }
                tmp->next = ptr;
                ptr->pre = tmp;
        }
}
void display(){
        struct node *tmp,*rev;
        tmp =head;
        while(tmp->next!=NULL){
                tmp=tmp->next;
        }
        rev=tmp;
        while(rev != NULL){
                printf("%d ",rev->data);
                rev=rev->pre;
        }
        printf("\n");
}
int main(){
        int n,value;
        scanf("%d",&n);
        for(int i =0;i<n;i++){
                scanf("%d",&value);
                insert(value);
        }
        display();
        return 0;
}
```


Output:

Test case - 1
User Output
4
11 151 201 251

251 201 151 11

Test case - 2
User Output
5
1 2 3 4 5
5 4 3 2 1

Result:

Thus the above program is executed successfully and the output has been verified

Aim:

Write a program to find the midpoint of a single linked list. If there are even number of nodes in the list, then print the second middle element.

Input Format:

- First line contains an integer, n, representing the number of elements of the list
- Second line contains n space separated integers representing the elements.

Output Format:

- Print the middle element of the single linked list.

Sample Test cases:

Input:

5
5 6 9 8 7

Output:

9

Input:

6
1 2 3 4 5 6

Output:

4

Program:

CTC26924.c

```
#include <stdio.h>
#include <stdlib.h>
struct node{
        int data;
        struct node *next;
}*head=NULL;
void insert(int value){
        struct node *ptr,*tmp;
        ptr = (struct node *)malloc(sizeof(struct node));
        ptr->data = value;
        ptr->next = NULL;
        if(head==NULL)
        head =ptr;
        else{
                tmp = head;
                while(tmp->next!=NULL){
                        tmp = tmp->next;
                }
                tmp ->next=ptr;
        }
}
void mid(int n){
        struct node *tmp;
        tmp=head;
        for(int i=1;i<=n/2;i++){
                tmp=tmp->next;
        }
        printf("%d",tmp->data);

}
int main(){
        int n,value;
        scanf("%d",&n);
        for(int i=0;i<n;i++){
                scanf("%d",&value);
                insert(value);
        }
        mid(n);
}
```


Output:

Test case - 1
User Output
5
5 6 9 8 7
9

Test case - 2
User Output
6
1 2 3 4 5 6

Result:

Thus the above program is executed successfully and the output has been verified

Aim:

You are given an array **Arr[]** of **N** no.of strings. Your job is to create a double linked list from the elements of the array and find the midpoint of the DLL

Note:

- For even no.of elements, consider the first node of the two middle nodes as the midpoint
- Elements of the array should be separated by ,(comma)

Complete the function **midPoint** with the following parameters:

- **N**: size of the Linked List
- **string Arr[]:**an array of strings

The function will return:

- **string:** the value of the midpoint

Constraints:

- $0 < N <= 10^5$
- $0 < len(Arr) < 10^5$

Sample Test Case

Input:

4

1,5,3,2

Output:

5

Important Instruction: Sample test case is for explanatory purposes but to run your custom test case on the terminal follow the input layout as mentioned in the command line arguments

Program:

CTC15005.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char * midPoint(int N, char *Arr[], int ArrLen) {
        // Write code here
        int a=ArrLen/2;
        if(ArrLen%2==0){
                return Arr[ArrLen/2-1];
        }
        return Arr[ArrLen/2];

}

int readStringArray(char *argsArray, char *arr[]) {
        int col = 0;
        char *token = strtok(argsArray, ",");
        while (token != NULL) {
                arr[col] = token;
                token = strtok(NULL, ",");
                col++;
        }
        return col;
}
int main(int argc, char *argv[]) {
        int N = atoi(argv[1]);
        char *Arr[strlen(argv[2])];
        int ArrLen = readStringArray(argv[2], Arr);
        printf("%s\n", midPoint(N, Arr, ArrLen));
        return 0;
}
```


Output:

Test case - 1
User Output
5

Test case - 2
User Output
f

Result:

Thus the above program is executed successfully and the output has been verified

Aim:

You are given the heads of two sorted arrays/lists **list1[]** and **list2[]**.

Merge the two lists into one sorted list. This list should be made by splicing together the nodes of the first two lists.

Return the head of the merged linked list

Note:

- Elements of the arrays should be ,(comma) separated.
- The final merged list should not be sorted.

Complete the function **mergeTwoSLL** with the following parameters:

integer list1[n1]: first array of n1 integers

integer list2[n2]: second array of n2 integers

Function will returns:

merged Single Linked List

Constraints:

- $0 < len(Arr1), len(Arr2) < 10^5$
- $0 < n1, n2 < 10^4$

Sample Test Case

Input 1:

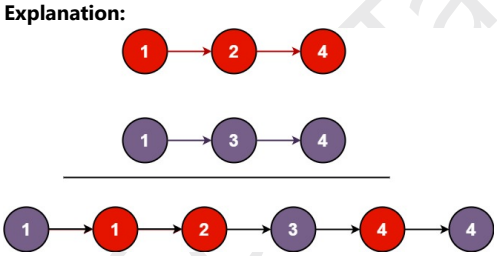
[1,2,4]

[1,3,4]

Output 1:

[1,1,2,3,4,4]

Explanation:



Input 2:

[ ]

[ ]

Output 2:

[ ]

Input 3:

[ ]

[0]

Output: [0]

Important Instruction : Sample test case is for explanatory purpose but to run your custom test case on the terminal follow the input layout as mentioned in the command line arguments

Program:

```
CTC15031.c
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MaxArrSize 100000


int mergeTwoSLL(short list1[], int list1Len, short list2[], int list2Len, short
*resultsArr) {
        int i=0,j=0,k=0;
        while (i < list1Len && j < list2Len) {
                if (list1[i] <= list2[j]) {
                        resultsArr[k++] = list1[i++];
                }
                else {
                        resultsArr[k++] = list2[j++];
                }
        }
        while (i < list1Len) {
                resultsArr[k++] = list1[i++];
        }
        while (j < list2Len) {
                resultsArr[k++] = list2[j++];
        }
        return k;
}

int readShortArray(char *argsArray, short arr[]) {
        int col = 0;
        char *token = strtok(argsArray, ",");
        while (token != NULL) {
                arr[col] = (short) atoi(token);
                token = strtok(NULL, ",");
                col++;
        }
        return col;
}
void printArrayElements(short *resultsArr, int resultsArrLength) {
        int index;
        for(index = 0; index < resultsArrLength - 1; index++) {
                printf("%d,", resultsArr[index]);
        }
        printf("%d\n", resultsArr[index]);
}

int main(int argc, char *argv[]) {
        short list1[strlen(argv[1])];
        short list2[strlen(argv[2])];
        int list1Len = readShortArray(argv[1], list1);
        int list2Len = readShortArray(argv[2], list2);
        short resultsArr[MaxArrSize];
        int resultsArrLength = mergeTwoSLL(list1, list1Len, list2, list2Len,
resultsArr);
        printArrayElements(resultsArr, resultsArrLength);
        return 0;
}
```

Output:

Test case - 1
User Output
1,1,2,3,4,4

Test case - 2
User Output
-78,-40,-33,-21,-5,0,1,1,3,5,6,7,8,12,20,21,54,78

Result:

Thus the above program is executed successfully and the output has been verified

Aim:

Given a linked list of**N**nodes. The task is to check if thelinked list has a loop. The linked list can containa self-loop.

Input format:

The first line contains an integer N, the number of nodes in the linked list
The Next line contains an array/list of integers separated by spaces.
The last line contains an integer X, the position at which the tail is connected.

Output format:

Print **True** if the linked list has a loop. Otherwise, print **False.**

Note: Position starts from 1

Constraints:

$1 \leq N \leq 10^3$
$1 \leq$ Data on Node $\leq 10^3$

Sample test case 1:

Input:

3
1 3 4
2

Output:

True

Explanation:

In the above test case N = 3. The linked list with nodes N = 3 is given. Then the value ofx = 2 is given which means the last node is connected with 2nd node of the linked list. Therefore, there exists a loop.

Sample test case 2:

Input:

7
1 5 3 6 4 8 9
0

Output:

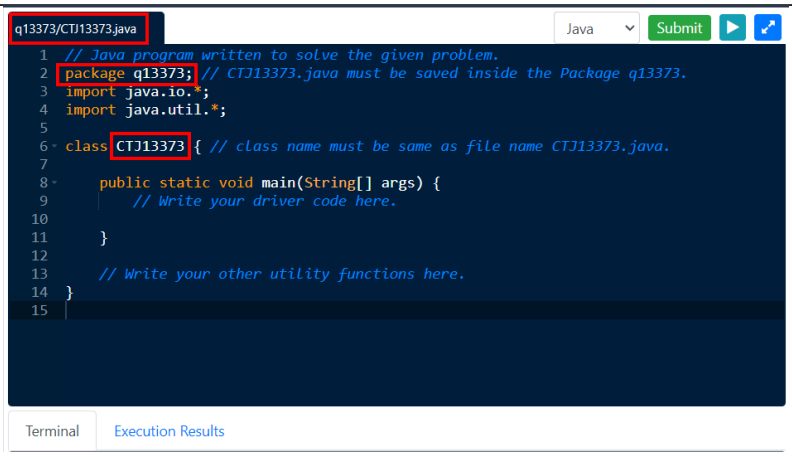
False

Explanation:

For N = 7, x = 0 means then lastNode -> next = NULL, then theLinked list does not contain any loop.

Instructions:

- To run your custom test cases strictly map your input and output layout with the visible test cases.
- For **Java users** follow the instructions given in the below snapshot.



Program:

CTC14238.c

```
#include <stdio.h>
#include <stdlib.h>
struct node {
        int data;
        struct node *next;
}*head=NULL;

void create(int value){
        struct node *ptr,*tmp;
        ptr= malloc(sizeof(struct node));
        ptr->next=NULL;
        ptr->data=value;
        if(head==NULL)
                head=ptr;
        else{
                tmp=head;
                while(tmp->next!=NULL){
                        tmp =tmp->next;
                }
                tmp->next=ptr;
        }
}
//floydes cycle detection loop
int loop(int x){
        struct node *slow = head,*fast=head;
        while(fast!=NULL &&fast->next!=NULL){
                slow = slow->next;
                fast = fast->next->next;
                if(slow == fast||x!=0)
                        return 1;
        }
        return 0;
}
int main(){
        int n,k,value;
        scanf("%d",&n);
        for(int i=0;i<n;i++){
                scanf("%d",&value);
                create(value);
        }
        scanf("%d",&k);
        int check = loop(k);
        if(check==1)
                printf("True\n");
        else
                printf("False\n");
}
```


Output:

Test case - 1
User Output
3
1 3 4

2
True

Test case - 2
User Output
7
1 5 3 6 4 8 9
0
False

Result:

Thus the above program is executed successfully and the output has been verified

Aim:

Write a program to compute the Polynomial Addition d=a + b using singly linked list where a and b be the pointers to two polynomials.

Input Format

Number of Terms for the First Polynomial:

- An integer numTerms1 representing the number of terms in the first polynomial.

Terms of the First Polynomial:

- Followed by numTerms1 lines, each containing:
- An integer coeff (the coefficient of the term).
- An integer exp (the exponent of the term).

Number of Terms for the Second Polynomial:

- An integer numTerms2 representing the number of terms in the second polynomial.

Terms of the Second Polynomial:

- Followed by numTerms2 lines, each containing:
- An integer coeff (the coefficient of the term).
- An integer exp (the exponent of the term).

Output Format

Display of the First Polynomial:

- The polynomial is displayed in the format: coeff1x^exp1 + coeff2x^exp2 + ...
- Terms should be ordered by exponents in descending order.
- The format should include " + " between terms, but no trailing " + " at the end.

Display of the Second Polynomial:

- Similar format as the first polynomial: coeff1x^exp1 + coeff2x^exp2 + ...

Display of the Sum of the Two Polynomials:

- The resulting polynomial from adding the two polynomials, displayed in the format: coeff1x^exp1 + coeff2x^exp2 + ...

Note : Refer to sample test cases for better understanding

Program:

CTC39066.c

```
#include <stdio.h>
#include <stdlib.h>
struct node {
        int coef;
        int exp;
        struct node *next;
};
struct node* create(int coef,int exp){
        struct node *ptr = (struct node*)malloc(sizeof(struct node));
        ptr->next=NULL;
        ptr->coef = coef;
        ptr->exp=exp;
        return ptr;
}
void insert(struct node **poly,int coef,int exp){
        struct node* ptr = create(coef,exp);
        if(*poly==NULL || (*poly)->exp<exp){
                ptr->next = *poly;
                *poly = ptr;
        }
        else{
                struct node *tmp = *poly;
                while(tmp->next!=NULL&&tmp->next->exp>=exp){
                        tmp=tmp->next;
                }
                ptr->next = tmp->next;
                tmp->next=ptr;
        }
}
void display(struct node* poly){
        struct node *tmp = poly;
        while(tmp!=NULL){
                printf("%dx^%d",tmp->coef,tmp->exp);
                tmp=tmp->next;
                if(tmp!=NULL)
                printf(" + ");
        }
        printf("\n");
}
struct node* add(struct node *poly1,struct node *poly2){
        struct node* result=NULL;
        while(poly1!=NULL&&poly2!=NULL){
                if(poly1->exp>poly2->exp){
                        insert(&result,poly1->coef,poly1->exp);
                        poly1=poly1->next;
                }
                else if(poly1->exp<poly2->exp){
                        insert(&result,poly2->coef,poly2->exp);
                        poly2=poly2->next;
                }
                else{
                        int sum = poly1->coef+poly2->coef;
                        if(sum!=0){
                                insert(&result,sum,poly1->exp);
                        }
                        poly1=poly1->next;
                        poly2=poly2->next;
```

```
                }
        }
        while(poly1!=NULL){
                insert(&result,poly1->coef,poly1->exp);
                poly1=poly1->next;
        }
        while(poly2!=NULL){
                insert(&result,poly2->coef,poly2->exp);
        }
        return result;
}
int main(){
        int num1,num2;
        struct node *poly1=NULL;
        struct node *poly2 =NULL;
        scanf("%d",&num1);
        int coef,exp;
        for(int i=0;i<num1;i++){
                scanf("%d %d",&coef,&exp);
                insert(&poly1,coef,exp);
        }
        scanf("%d",&num2);
        for(int i= 0;i<num2;i++){
                scanf("%d %d",&coef,&exp);
                insert(&poly2,coef,exp);
        }
        display(poly1);
        display(poly2);
        struct node* result =add(poly1,poly2);
        display(result);
}
```

Output:

Test case - 1
User Output
3
5 2
-3 1
1 0
3
2 3
4 1
-7 0
5x^2 + -3x^1 + 1x^0
2x^3 + 4x^1 + -7x^0
2x^3 + 5x^2 + 1x^1 + -6x^0

Test case - 2
User Output

4
2 4
-3 3
5 2
1 0
3
1 3
2 2
-7 1
2x^4 + -3x^3 + 5x^2 + 1x^0
1x^3 + 2x^2 + -7x^1
2x^4 + -2x^3 + 7x^2 + -7x^1 + 1x^0

Result:

Thus the above program is executed successfully and the output has been verified

Aim:

Write a program that implements the **Stack** Data structure using Arrays

The operations performed on the Stack are

1. Push
2. Pop
3. Peek

Input format:

1. The first line contains an integer n (1 ≤ n ≤ 1000), representing the number of operations to be performed on the stack.
2. The next n lines contain two integers separated by a space:
    - The first integer indicates the operation to be performed:
        - 1: Push an element onto the stack.2: Pop the top element from the stack.3: Peek the top element of the stack.
    - The second integer is only present if the operation is 1 (Push). It indicates the element to be pushed onto the stack.
    - If the operation is 2 (Pop) or 3 (Peek), there will be only one integer (the operation type) in the line.

Output format:

- For every 3 (Peek) operation, print the top element of the stack on a new line.
- After all operations are performed, print the elements in the stack, space-separated.

Sample Test Case:

Input:

6
1 2
1 3
1 4
2
3
1 5

Output:

3
5 3 2

Explanation:

1 2 ---> 2 will be pushed
1 3 ---> 3 will be pushed
1 4 ---> 4 will be pushed
2 ---> Last element(4) is popped
3 ---> peek operation is performed which results in printing the top element of the stack i.e 3
1 5 ---> 5 will be pushed
6 Operations are completed
Therefore one peek operation is performed so the output is 3 and the final stack **5 3 2**

Note:

- If the stack is empty and when pop and peek are performed first, proceed to the next operation without displaying and modifying anything in the stack.
- If the stack is empty and nothing is pushed to the stack, Print Empty at the end.
- Display the output after all inputs have been taken.
- Refer to visible test cases to strictly match with input/output layout.

Program:

`CTC17094.c`

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 100
int sarr[MAX];
int top = -1;
int pval = 0;
void push(int value){
        if(top==MAX-1)
                return;
        top++;
        sarr[top]=value;
}
void pop(){
        if(top==-1)
                return;
        top--;
}
void peek(){
        if(top==-1)
                return;
        else
                pval=sarr[top];
}
void display(){
        if(top==-1)
        printf("Empty\n");
        for(int i=top;i>=0;i--){
                if(i!=0)
                        printf("%d ",sarr[i]);
                else
                        printf("%d\n",sarr[i]);
        }
}
int main(){
        int n,task,value,flag=0;
        scanf("%d",&n);
        for(int i=0;i<n;i++){
                scanf("%d",&task);
                if(task==1){
                        scanf("%d",&value);
                        push(value);
                }
                else if(task==2){
                        pop();
                }
                else if(task==3){
                        if(top!=-1){
                                peek();
                                flag++;
                        }
                }
        }
        if(flag>0){
                printf("%d\n",pval);
        }
        display();
}
```

Output:

Test case - 1
User Output
6
1 2
1 3
1 4
2
3
1 5
3
5 3 2

Test case - 2
User Output
5
1 44
1 55
2
2
3
Empty

Result:

Thus the above program is executed successfully and the output has been verified

Aim:

Write a program that implements the **Stack** Data structure using Linked lists

The operations performed on the Stack are

1. Push
2. Pop
3. Peek

Input format:

- The first line contains an integer n, the number of operations to be performed on the stack.
- The next n lines contain two integers each:
- The first integer indicates the operation type (1 for Push, 2 for Pop, 3 for Peek).If the operation is Push (indicated by 1), the second integer will be the element to be pushed onto the stack.If the operation is Pop (indicated by 2), no second integer is needed.If the operation is Peek (indicated by 3), no second integer is needed.

Output format:

- After performing any Peek operation (operation type 3), print the top element of the stack.
- After performing all operations, print the elements of the stack, as per shown in the sample test cases.

Sample Test Case:

Input:

6
1 2
1 3
1 4
2
3
1 5

Output:

3
5->3->2

Explanation:

1 2 ---> 2 will be pushed
1 3 ---> 3 will be pushed
1 4 ---> 4 will be pushed
2 ---> Last element(4) is popped
3 ---> peek operation is performed which results in printing the top element of the stack i.e 3
1 5 ---> 5 will be pushed
6 Operations are completed
Therefore one peek operation is performed so the output is 3 and the final stack **5->3->2**

Note:

- If the stack is empty and when pop and peek are performed first, proceed to the next operation without displaying and modifying anything in the stack.
- If the stack is empty and nothing is pushed to the stack, Print Empty at the end.

Program:

CTC17095.c

```
#include <stdio.h>
#include <stdlib.h>
int peekval =0;
struct stack{
        int data;
        struct stack *next;
}*top =NULL;

void push (int value){
        struct stack *ptr=malloc(sizeof(struct stack));
        ptr->data=value;
        ptr->next = top;
        top =ptr;
}
void pop(){
        if(top==NULL)
                return;
        struct stack *tmp=top;
        top = top->next;
        free(tmp);
}
void peek(){
        if(top==NULL){
                return;
        }
        peekval=top->data;
}

void display(){
        struct stack *tmp=top;
        if(top==NULL){
                printf("Empty\n");
        }
        else{
                while(tmp->next!=NULL){
                        printf("%d->",tmp->data);
                        tmp=tmp->next;
                }
                printf("%d\n",tmp->data);
        }
}
int main(){
        int n,task,value,flag=0;
        scanf("%d",&n);
        for(int i=0;i<n;i++){
                scanf("%d",&task);
                if(task==1){
                        scanf("%d",&value);
                        push(value);
                }
                else if(task==2)
                pop();
                else if(task==3)
                if(top!=NULL){
                peek();
                flag++;
                }
```

```
        }
        if(flag>0){
        printf("%d\n",peekval);
        }
        display();
}
```

Output:

Test case - 1
User Output
6
1 2
1 3
1 4
2
3
1 5
3
5->3->2

Test case - 2
User Output
5
1 44
1 55
2
2
3
Empty

Test case - 3
User Output
7
1 7
1 8
2
1 9
3
1 10
2
9
9->7

Result:

Thus the above program is executed successfully and the output has been verified

Aim:

You are given a string **expr** which contains an infix expression. Your need to convert the infix expression to an equivalent postfix expression.

Complete the function **InfixToPostfix** with the following parameter:

expr: Infix expression as an input in the form of string

The function will return:

string: Postfix expression in the form of string.

Constraints:

0 < len(expr) <= 10^5

Sample test case:

Input: (a+b/c)

Output: abc/+

Important Instruction : Sample test case is for explanatory purpose but to run your custom test case on the terminal follow the input layout as mentioned in the command line arguments.

Program:

CTC9758.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#define MAX 100
int pre(char c){
        if(c=='+'||c=='-')return 1;
        if(c =='*'||c=='/')return 2;
        if(c == '^')return 3;
        return 0;
}
int isOperator(char ch){
        return ch=='+'||ch =='-'||ch=='*'||ch=='/'||ch=='^';
}


char * InfixToPostfix(char *expr) {
                char *postfix = (char *)malloc(MAX* sizeof(char));
        char stack[MAX];
        int top =-1;
        int k=0;
        for(int i =0;expr[i];i++){
                if(isdigit(expr[i])||isalpha(expr[i])){
                        postfix[k++]=expr[i];
                }
                else if(expr[i]=='('){
                        stack[++top]=expr[i];
                }
                else if(expr[i]== ')'){
                        while (top!=-1&& stack[top] != '('
                                postfix[k++]=stack[top--];
                        top--;
                }
                else if(isOperator(expr[i])){
                       while(top != -1 && pre(stack[top]) >= pre(expr[i])){
                                if(expr[i]=='^'&& stack[top ]=='^')break;
                                postfix[k++]=stack[top--];
                        }
                        stack[++top]=expr[i];
                }
        }
        while(top!=-1){
                postfix[k++]=stack[top--];
        }
        postfix[k]='\0';
        return postfix;
}

int main(int argc, char *argv[]) {
        char *expr = argv[1];
        printf("%s\n", InfixToPostfix(expr));
        return 0;
}
```


Output:

Test case - 1
User Output
abc/+

Test case - 2
User Output
abc/d*+ef^-

Result:

Thus the above program is executed successfully and the output has been verified

Aim:

You are given a string **str** of some set of braces**.** Your task is to check whether the given parentheses in the string are balanced or not.

Complete the function **BalancedBrackets** with the following parameter(s):

string Str:String that contains the parenthesis.

The functionreturns:

A **string** that returns'**Balanced**' if the parenthesis of the input string are balanced or returns '**Not balanced**' if the parenthesis are not balanced.

Constraints:

- **1 ≤length of each string str ≤ 10^2**

Note: Every string consists of (,), {, }, [, and] braces only.

Sample test case 1:
Input:
[{}]{}[]()

Output:
Balanced

Explanation:
The brackets in the first **'[{}]{}[]()**' are balanced, because all brackets are closed and all nested brackets are closed in order. So the output is **Balanced.**

Sample test case 2:
Input:
[{]}

Output:
Not balanced

Explanation:
The brackets in the second string '**[{]}**' are not balanced, because the nested bracket '{' was not closed before it's surrounding '[ ]', so the order was not respected. Then the output is **Not balanced.**
Instructions: To run your custom test cases strictly map your input and output layout with the visible test cases.

Program:

CTC15014.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char * BalancedBrackets(char *str) {
                char open[]={'(','{','['};
        char close[]={')','}',']'};
        static char b[]="Balanced";
        static char u[]="Not balanced";
        char stack[500];
        int top = -1;
        for(int i=0;str[i]!='\0';i++){
                for(int j=0;j<3;j++){
                        if(str[i]==open[j]){
                                stack[++top]=open[j];
                        }
                        else if(str[i]==close[j]){
                                if(stack[top]==open[j])
                                        top--;
                                else{
                                        return u;
                                }
                        }
                }
        }
        if(top==-1)
                return b;
        return u;
}

int main(int argc, char *argv[]) {
        char *str = argv[1];
        printf("%s\n", BalancedBrackets(str));
        return 0;
}
```


Output:

Test case - 1
User Output
Balanced

Test case - 2
User Output
Not balanced

Result:

Thus the above program is executed successfully and the output has been verified

Aim:

Your task is to design a program that reverse the order of elements in a stack using recursion, without utilizing any loops.

Input format:

- The first line should contain an integer **n** representing the number of elements in the stack.
- The second line should contain **n** space-separated integers representing the elements of the stack.

Output format:

- The output will display the elements of the reversed stack, also separated by space

Constraints:

1 <= size of the stack <= 10^4
-10^9 <= Each element of the stack <= 10^9

Example input :

8
2 9 -54 92 6 9 2 0

Example output:

0 2 9 6 92 -54 9 2

Program:

CTC30989.c

```
#include <stdio.h>
#include <stdlib.h>
struct node{
        int data;
        struct node *next;
}*top = NULL;

void push(int value){
        struct node *ptr;
        ptr = malloc(sizeof(struct node));
        ptr->data = value;
        ptr->next = top;
        top=ptr;
}
void display(){
        struct node* tmp;
        if(top==NULL)
        printf("stack is empty\n");
        tmp=top;
        while(tmp->next!=NULL){
                printf("%d ",tmp->data);
                tmp=tmp->next;
        }
        printf("%d\n",tmp->data);
}
int main(){
        int n,value;
        scanf("%d",&n);
        for(int i=0;i<n;i++){
                scanf("%d",&value);
                push(value);
        }
        display();
}
```


Output:

Test case - 1
User Output
8
2 9 -54 92 6 9 2 0
0 2 9 6 92 -54 9 2

Test case - 2
User Output
10
-5 -6 79 373 9872 82 9 102 523 0
0 523 102 9 82 9872 373 79 -6 -5

Result:

Thus the above program is executed successfully and the output has been verified

Aim:

Write a program to check whether the given element is present or not in the array of elements using the binary search Technique

Input format:

- The first line of input contains an integer N representing the no. of elements of the array
- The second line input contains the array of N integers separated by space
- The last line of input contains the key element to be searched

Output format:

- If the element is found, print the index.
- If the element is not found, print **Not found**.

Sample Test Case:

Input:

7
1 2 3 4 3 5 6
3

Output:

2

Program:

CTC17128.c

```
#include <stdio.h>
int main(){
        int n,k;
        scanf("%d",&n);
        int arr[n];
        for(int i=0;i<n;i++){
                scanf("%d",&arr[i]);
        }
        scanf("%d",&k);
        int i=0,j=n-1,index,flag=0;
        while(i<=j){
                int mid = (i+j)/2;
                if(arr[mid]==k){
                        flag =1;
                        index =mid;
                        break;
                }
                else if(arr[mid]>k){
                        j=mid-1;
                }
                else if(arr[mid]<k){
                        i = mid+1;
                }
        }
        if(flag==1)
                printf("%d\n",index);
        else
                printf("Not found\n");
}
```


Output:

Test case - 1
User Output
7
1 2 3 4 3 5 6
3
2

Test case - 2
User Output
10
1 2 3 4 5 6 7 8 9 19
20
Not found

Result:

Thus the above program is executed successfully and the output has been verified

Aim:

Problem:

You are given a positive integer **N**. Your job is to print the Fibonacci series of numbers as an array/list up to the given integer **N**.

Complete the function **fibonacciNumbers** with the following parameter:
integer N

Function will returns:
Array: an array of integers that is having all the fibonacci numbers upto N

Constraints:
$0 < N <= 10^5$

Sample Test Case
Input:
5

Output:
[0, 1, 1, 2, 3, 5]

Program:

CTC9250.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MaxArrSize 100000


int fibonacciNumbers(int N, int *resultsArr) {
        if(N<0)
                return 0;
        int count =0;
        resultsArr[count++]=0;
        if(N>=1){
                resultsArr[count++]=1;
        }
        while(1){
                int next =resultsArr[count-1]+resultsArr[count-2];
                if(next>N) break;
                resultsArr[count++]=next;
        }
        return count;
}

void printArrayElements(int *resultsArr, int resultsArrLength) {
        int index;
        for(index = 0; index < resultsArrLength - 1; index++) {
                printf("%d,", resultsArr[index]);
        }
        printf("%d\n", resultsArr[index]);
}

int main(int argc, char *argv[]) {
        int N = atoi(argv[1]);
        int resultsArr[MaxArrSize];
        int resultsArrLength = fibonacciNumbers(N, resultsArr);
        printArrayElements(resultsArr, resultsArrLength);
        return 0;
}
```


Output:

Test case - 1
User Output
0,1,1,2,3,5

Test case - 2
User Output
0,1,1,2,3,5,8,13,21,34,55,89

Result:

Thus the above program is executed successfully and the output has been verified

Aim:

Given a Towers of Hanoi puzzle with N discs initially placed on Peg A, you need to write a function to find the minimum number of movements required to solve the puzzle, moving all the discs from Peg A to Peg C, using Peg B as an intermediate peg.

Input Format:

The input consists of a single integer N, representing the number of discs.

Output Format:

The function should return an integer representing the minimum number of movements required to solve the puzzle.

Constraints:

1 ≤ N ≤ 20

Example:

Input:

N = 3

Output:

7

Explanation:

For a Towers of Hanoi puzzle with 3 discs, it takes a minimum of 7 movements to move all the discs from Peg A to Peg C using Peg B as an intermediate peg.

Note:

In the Towers of Hanoi puzzle, the number of movements required to solve the puzzle follows the pattern 2^N - 1, where N is the number of discs.

Instruction: To run your custom test cases strictly map your input and output layout with the visible test cases.

Program:

CTC17071.c

```c
#include <stdio.h>
#include <math.h>

int count(int n){
        int x =(pow(2,n))-1;
        return x;
}
int main(){
        int n;
        scanf("%d",&n);
        int k = count(n);
        printf("%d",k);
}
```


Output:

Test case - 1
User Output
3
7

Test case - 2
User Output
5
31

Result:

Thus the above program is executed successfully and the output has been verified

Aim:

Write a program that implements the **Queue** Data structure using Arrays

The operations to be performed on the Queue are

1. Enqueue
2. Dequeue

Input format:

- The first line contains a single integer n, which denotes the number of operations to be performed on the queue.
- Each of the following n lines contains two space-separated integers:
- The first integer indicates the operation:1 → Enqueue operation (followed by an element to be added to the queue).2 → Dequeue operation (removes the front element of the queue).If the operation is 1, the second integer is the element to be enqueued.If the operation is 2, no second integer follows since it's a dequeue operation.

Output format:

- Display the queue with space separated integers after performing all operations at the end.

Sample Test Case:

Input:

6
1 2
1 3
1 4
2
2
1 5

Output:

4 5

Explanation:

1 2 ---> 2 will be inserted
1 3 ---> 3 will be inserted
1 4 ---> 4 will be inserted
2 ---> first element(2) is removed
2 ---> second element(3) is removed
1 5 ---> 5 will be inserted
6 Operations are completed
The final queue is **4 5**

Note:

- If the queue is empty till the end and nothing is inserted into the queue, Print **Empty** at the end.
- If the queue is empty and when dequeue is performed first, proceed to the next operation without displaying and modifying anything in the queue.
- Display the output after all inputs have been taken.
- Refer to the displayed test cases for better understanding.

Program:

CTC17096.c

```
#include <stdio.h>
#include <stdbool.h>
#define MAX 1000
int arr[MAX];
int backInd = -1;
int frontInd = -1;
bool isEmpty(){
        if(backInd!=MAX-1)
                return true;
        else
                return false;
}
void enQueue(int value){
        bool check = isEmpty();
        if(check==true){
                backInd++;
                arr[backInd]=value;
        }
        else
                return;
}
void deQueue(){
        if(backInd!=frontInd){
                frontInd++;
        }
        else
                return;
}
void display(){
        if(frontInd==backInd){
                printf("Empty\n");
        }
        else{
                for(int i= frontInd+1;i<=backInd;i++){
                        if(i<backInd)
                        printf("%d ",arr[i]);
                        else
                        printf("%d\n",arr[i]);
                }
        }
}
int main(){
        int n,value,task;
        scanf("%d",&n);
        for(int i= 0;i<n;i++){
                scanf("%d",&task);
                if(task==1){
                        scanf("%d",&value);
                        enQueue(value);
                }
                else if(task==2){
                        deQueue();
                }
        }
        display();
}
```

Output:

Test case - 1
User Output
6
1 2
1 3
1 4
2
2
1 5
4 5

Test case - 2
User Output
3
1 11
2
2
Empty

Test case - 3
User Output
5
2
1 33
1 44
1 55
2
44 55

Result:

Thus the above program is executed successfully and the output has been verified

Aim:

Write a program that implements the **Circular Queue** using Arrays

The operations to be performed on the **Circular Queue** are

1. Enqueue
2. Dequeue

Input format:

- The first line of input contains the number of operations to be performed on the Circular Queue.
- The next lines contain the integers separated by space in which the first integer indicates the operation to be performed and the second integer contains the element to be enqueued.
- **1 --->** indicates the Enqueue
- **2 --->** indicates the Dequeue

Output format:

- Display the Circular Queue after performing all operations at the end.

Sample Test Case:

Input:

6
1 2
1 3
1 4
2
2
1 5

Output:

4 5

Explanation:

1 2 ---> 2 will be inserted
1 3 ---> 3 will be inserted
1 4 ---> 4 will be inserted
2 ---> first element(2) is removed
2 ---> second element(3) is removed
1 5 ---> 5 will be inserted
6 Operations are completed
The final Circular Queue is **4 5**

Note:

- If the queue is empty till the end and nothing is inserted into the queue, Print **Empty** at the end.
- If the queue is empty and when dequeue is performed first, proceed to the next operation without displaying and modifying anything in the Circular Queue.

Program:

CTC17098.c

```
#include <stdio.h>
#define MAX 100
int arr[MAX];
int front =-1;
int rear = -1;
void enQueue(int value){
        if((rear+1)%MAX==front)
                return;
        if(front==-1)front++;
        rear = (rear+1)%MAX;
        arr[rear]=value;
}
void deQueue(){
        if(front ==-1)
        return;
        if(front ==rear)front=rear=-1;
        else{
                front = (front+1)%MAX;
        }
}
void display(){
        if(front==-1){
                printf("Empty\n");
                return;
        }
        for(int i=front;;i=(i+1)%MAX){
                if(i!=rear)
                printf("%d ",arr[i]);
                if(i==rear){
                        printf("%d\n",arr[i]);
                        break;
                }
        }
}
int main(){
        int n,value,task;
        scanf("%d",&n);
        for(int i=0;i<n;i++){
                scanf("%d",&task);
                if(task==1){
                        scanf("%d",&value);
                        enQueue(value);
                }
                if(task==2)
                deQueue();
        }
        display();
}
```


Output:

Test case - 1
User Output
6
1 2

1 3
1 4
2
2
1 5
4 5

Test case - 2
User Output
3
1 11
2
2
Empty

Test case - 3
User Output
5
2
1 33
1 44
1 55
2
44 55

Result:

Thus the above program is executed successfully and the output has been verified

Aim:

Write a program that implements the **Queue** Data structure using two stacks

The operations to be performed on the Queue are

1. Enqueue
2. Dequeue

Input format:

- The first line of input contains the number of operations to be performed on the queue
- The next lines contain the integers separated by space in which the first integer indicates the operation to be performed and the second integer contains the element to be enqueued.
- **1 --->** indicates the Enqueue
- **2 --->** indicates the Dequeue

Output format:

- Display the queue after performing all operations at the end.

Sample Test Case:

Input:

6
1 2
1 3
1 4
2
2
1 5

Output:

4 5

Explanation:

1 2 ---> 2 will be inserted
1 3 ---> 3 will be inserted
1 4 ---> 4 will be inserted
2 ---> first element(2) is removed
2 ---> second element(3) is removed
1 5 ---> 5 will be inserted
6 Operations are completed
The final queue is **4 5**

Note:

- If the queue is empty till the end and nothing is inserted into the queue, Print **Empty** at the end.
- If the queue is empty and when dequeue is performed first, proceed to the next operation without displaying and modifying anything in the queue.

Program:

CTC39024.c

```
#include <stdio.h>
#include <stdbool.h>
#define max 100
int arr[max];
int b=-1;
int f=-1;
bool isempty(){
        if(b!=max-1)
                return true;
        else
                return false;
}
void enqueue(int value){
        bool check = isempty();
        if(check== true){
                b++;
                arr[b]=value;
        }
        else
                return;
}
void dequeue(){
        if(b!=f){
                f++;
        }
        else
                return;
}
void display(){
        if(f==b){
                printf("Empty\n");
        }
        else{
                for(int i =f+1;i<=b;i++){
                        if(i<b)
                        printf("%d ",arr[i]);
                        else
                                printf("%d\n",arr[i]);
                }
        }
}
int main(){
        int n,value,task;
        scanf("%d",&n);
        for(int i =0;i<n;i++){
                scanf("%d",&task);
                if(task==1){
                        scanf("%d",&value);
                        enqueue(value);
                }
                else if(task==2){
                        dequeue();
                }
        }
        display();
}
```

Output:

Test case - 1
User Output
6
1 2
1 3
1 4
2
2
1 5
4 5

Test case - 2
User Output
3
1 11
2
2
Empty

Test case - 3
User Output
5
2
1 33
1 44
1 55
2
44 55

Result:

Thus the above program is executed successfully and the output has been verified

Aim:

Write a program to implement the Priority Queue using Heaps. The following operations are to be performed.

1. Enqueue
2. Dequeue

Input format:

- The first line of input contains the number of operations to be performed on the Binary Search Tree
- The next lines contain the integers separated by space in which the first integer indicates the operation to be performed and the second integer indicates the element to be inserted and the third integer is the respective priority (Only for insertion)
- **1 --->** indicates the Enqueue
- **2 --->** indicates the Dequeue

Output format:

- Display the Priority queue after performing all the operations.

Sample Test Case:

Input:

4
1 4 5
1 7 5
1 2 3
2

Output:

4 (5) 7 (5)

Explanation:

1 4 5 --> 4 will be inserted with the priority 5
1 7 5 --> 7 will be inserted with the priority 5
1 2 3 --> 2 will be inserted with the priority 3
2 --> Element (2) with the most priority will be removed.
Finally, the priority Queue is displayed.

Note:

- If the queue is empty and when dequeue is performed first, proceed to the next operation without displaying and modifying anything in the queue.
- If the queue is empty till the end and nothing is inserted into the queue, Print **Empty** at the end.
- Refer to the test cases for a better understanding

Program:

CTC17101.c

```c
#include <stdio.h>
#include <stdlib.h>
typedef struct Node {
        int data;
        int priority;
        struct Node* next;
} Node;
Node* newNode(int data, int priority) {
        Node* temp = (Node*)malloc(sizeof(Node));
        temp->data = data;
        temp->priority = priority;
        temp->next = NULL;
        return temp;
}
Node* dequeue(Node** head) {
        if (*head == NULL) {
                return NULL;
        }
        Node* temp = *head;
        *head = (*head)->next;
        return temp;
}
void enqueue(Node** head, int data, int priority) {
        Node* start = *head;
        Node* temp = newNode(data, priority);

        if (*head == NULL || (*head)->priority > priority) {
                temp->next = *head;
                *head = temp;
        } else {
                while (start->next != NULL && start->next->priority <= priority) {
                        start = start->next;
                }
                temp->next = start->next;
                start->next = temp;
        }
}
void printQueue(Node* head) {
        while (head != NULL) {
                printf("%d (%d)", head->data, head->priority);
                head = head->next;
                if (head != NULL) {
                        printf(" ");
                }
        }
        printf("\n");
}
int main() {
        int numOperations;
        scanf("%d", &numOperations);
        Node* pq = NULL;
        for (int i = 0; i < numOperations; i++) {
                int operation, data, priority;
                scanf("%d", &operation);
                if (operation == 1) {  // Enqueue operation
                        scanf("%d %d", &data, &priority);
                        enqueue(&pq, data, priority);
```

```
                } else if (operation == 2) {  // Dequeue operation
                        Node* dequeuedNode = dequeue(&pq);
                        free(dequeuedNode);
                }
        }
        if (pq == NULL) {
                printf("Empty\n");
        } else {
                printQueue(pq);
        }
        return 0;
}
```

Output:

Test case - 1
User Output
4
1 4 5
1 7 5
1 2 3
2
4 (5) 7 (5)

Test case - 2
User Output
6
1 1 1
1 2 2
1 3 3
1 4 4
1 5 5
2
2 (2) 3 (3) 4 (4) 5 (5)

Test case - 3
User Output
4
1 3 3
2
1 4 5
2
Empty

Result:

Thus the above program is executed successfully and the output has been verified

Aim:

You are given an array/list **Arr[]** of strings. You have to create a doubly linked list using the elements of the array/list and sort the elements of the linked list using merge sort.

Note: Data of sorted doubly linked list must be returned as a string in required format mentioned in the visible test cases.

Complete the function **mergeSort** with the following parameters:
string Arr[n]: an array of n strings
Function will return:
string: string of the merged list as shown in the output

Constraints:

- $0 < len(Arr) < 10^5$
- string data in the linked list can have numeric characters or alphabets.
- Numeric characters or aphabets must be sorted in lexicographical order.

Sample Test Case
Input:
orange,red,blue,green,yellow

Output:
blue -> green -> orange -> red -> yellow (-> is used to represent doubly linked list link between two nodes, format is blue{space}->{space}.....)

Instruction: For input and output layout strictly follow the visible sample test cases and follow the function rules mentioned below.

Program:

CTC9272.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
typedef struct Node {
        char data[100];
        struct Node* next;
        struct Node* prev;
} Node;

Node* createNode(char* data) {
        Node* newNode = (Node*)malloc(sizeof(Node));
        strcpy(newNode->data, data);
        newNode->next = NULL;
        newNode->prev = NULL;
        return newNode;
}

Node* split(Node* head) {
        Node* fast = head;
        Node* slow = head;
        while (fast->next && fast->next->next) {
                fast = fast->next->next;
                slow = slow->next;
        }
        Node* temp = slow->next;
        slow->next = NULL;
        return temp;
}
Node* merge(Node* first, Node* second) {
        if (!first) return second;
        if (!second) return first;
        if (strcmp(first->data, second->data) < 0) {
                first->next = merge(first->next, second);
                first->next->prev = first;
                first->prev = NULL;
                return first;
        } else {
                second->next = merge(first, second->next);
                second->next->prev = second;
                second->prev = NULL;
                return second;
        }
}
Node* mergeSortUtil(Node* head) {
        if (!head || !head->next) return head;
        Node* second = split(head);
        head = mergeSortUtil(head);
        second = mergeSortUtil(second);
        return merge(head, second);
}

char* listToString(Node* head) {
        static char result[10000];
        result[0] = '\0';
        while (head) {
                strcat(result, head->data);
                if (head->next) {
```

```
                        strcat(result, " -> ");
                }
                head = head->next;
        }
        return result;
}

char * mergeSort(char *Arr[], int ArrLen) {
                if (ArrLen == 0) {
                static char empty[] = "Empty";
                return empty;
        }
        Node* head = createNode(Arr[0]);
        Node* current = head;
        for (int i = 1; i < ArrLen; i++) {
                current->next = createNode(Arr[i]);
                current->next->prev = current;
                current = current->next;
        }
        head = mergeSortUtil(head);
        return listToString(head);
}

int readStringArray(char *argsArray, char *arr[]) {
        int col = 0;
        char *token = strtok(argsArray, ",");
        while (token != NULL) {
                arr[col] = token;
                token = strtok(NULL, ",");
                col++;
        }
        return col;
}
int main(int argc, char *argv[]) {
        char *Arr[strlen(argv[1])];
        int ArrLen = readStringArray(argv[1], Arr);
        printf("%s\n", mergeSort(Arr, ArrLen));
        return 0;
}
```


Output:

Test case - 1
User Output
blue -> green -> orange -> red -> yellow

Test case - 2
User Output
apple

Result:

Thus the above program is executed successfully and the output has been verified

Aim:

Given a list of N array elements, apply **quick sort** on an array using recursion to arrange the elements in **ascending order**.

Constraints:

- $1 <= N <= 1000$
- $-100000 <= a[i] <= 100000$

Input Format:

- The first line contains the integer N, the size of the array.
- The next line contains N space-separated integers.

Output Format:

- Print the sorted array as a row of space-separated integers.

Note: Use the divide and conquer using the recursion approach to solve the question.

Program:

CTC38656.c

```
#include <stdio.h>
int main(){
        int n;
        scanf("%d",&n);
        int arr[n];
        for(int i=0;i<n;i++){
                scanf("%d",&arr[i]);
        }
        for(int i=0;i<n-1;i++){
                for(int j=0;j<n-1;j++){
                        if(arr[j]>arr[j+1]){
                                int tmp =arr[j];
                                arr[j]=arr[j+1];
                                arr[j+1]=tmp;
                        }
                }
        }
        for(int i=0;i<n;i++){
                printf("%d ",arr[i]);
        }
}
```


Output:

Test case - 1
User Output
5
4 8 2 -3 0
-3 0 2 4 8

Test case - 2
User Output

8
-6 -2 -8 -36 0 69 -5 -96
-96 -36 -8 -6 -5 -2 0 69

Result:

Thus the above program is executed successfully and the output has been verified

Aim:

You are tasked with managing the scores of students from two different tests. The scores for each test are stored in two separate arrays ($nums1$ and $nums2$), which are both sorted in non-decreasing order. The challenge is to merge these two sorted arrays into a single sorted array.

In particular, you are given:

1. An array $nums1$ of size `m + n`, where the first m elements contain the scores from the first test and the remaining n elements are initialized to 0. These 0s are placeholders to make room for the elements of the second array $nums2$.
2. An array $nums2$ of size n, which contains the scores from the second test.
3. Your goal is to merge these two arrays into one sorted array, placing the result in $nums1$, so that the final array contains all the scores from both tests, sorted in non-decreasing order.
4. You need to perform this merge operation in-place (without using any additional data structures like arrays or lists) and ensure that the final sorted array has a length of `m + n`.

Constraints:

- $0 \leq\ m,\ n \leq 200$
- $1 \leq\ m + n\ \leq 200$
- $-10^9 \leq\ nums1[i],\ nums2[j]\ \leq 10^9$
- The $nums1$ array has enough space to accommodate all elements from both arrays (`nums1.length == m + n`).

Input Format:

- The input consists of four lines:
- The first line contains the space-separated integers of the $nums1$ array, where the first m elements represent the scores from the first test, and the remaining n elements are placeholders (0s).
- The second line contains an integer m, representing the number of valid elements in the $nums1$ array.
- The third line contains the space-separated integers of the $nums2$ array, containing the scores from the second test.
- The fourth line contains an integer n, representing the number of elements in the nums2 array.

Output Format:

- Output the merged array after merging the two sorted arrays into $nums1$. The merged array should be sorted in non-decreasing order.

Example 1:

Input: nums1 = [1,2,3,0,0,0], m = 3, nums2 = [2,5,6], n = 3
Output: [1,2,2,3,5,6]
Explanation: The arrays we are merging are [1,2,3] and [2,5,6].
The result of the merge is [1,2,2,3,5,6] with the underlined elements coming from nums1.

Program:

CTC27905.c

```c
#include <stdio.h>
void merge(int nums1[], int m, int nums2[], int n) {
        int mergedIndex = m + n - 1;
        int i = m - 1;
        int j = n - 1;

        while (i >= 0 && j >= 0) {
                if (nums1[i] > nums2[j]) {
                        nums1[mergedIndex] = nums1[i];
                        i--;
                } else {
                        nums1[mergedIndex] = nums2[j];
                        j--;
                }
                mergedIndex--;
        }
        while (j >= 0) {
                nums1[mergedIndex] = nums2[j];
                j--;
                mergedIndex--;
        }
}
int main() {
        int nums1[200], m;
        int nums1Size = 0;
        while (scanf("%d", &nums1[nums1Size]) == 1) {
                nums1Size++;
                if (getchar() == '\n') break;
        }
        scanf("%d", &m);
        int nums2[200], n;
        int nums2Size = 0;
        while (scanf("%d", &nums2[nums2Size]) == 1) {
                nums2Size++;
                if (getchar() == '\n') break;
        }
        scanf("%d", &n);
        merge(nums1, m, nums2, n);
        printf("[");
        for (int i = 0; i < m + n; i++) {
                printf("%d", nums1[i]);
                if (i != m + n - 1) printf(", ");
        }
        printf("]\n");
        return 0;
}
```


Output:

Test case - 1
User Output
1 2 3 0 0 0
3
2 5 6

3
[1, 2, 2, 3, 5, 6]

Test case - 2
User Output
1
1
0
0
[1]

Result:

Thus the above program is executed successfully and the output has been verified

Aim:

Given a list of N array elements, apply **quick sort** on the array to arrange the elements in **ascending order**.

Constraints:

- $1 <= N <= 1000$
- $-100000 <= a[i] <= 100000$

Input Format:

- The first line contains the integer N, the size of the array.
- The next line contains N space-separated integers.

Output Format:

- Print the sorted array as a row of space-separated integers.

Note: Use the divide and conquer approach to solve the question.

Program:

CTC14074.c

```
#include <stdio.h>
int main (){
        int n;
        scanf("%d",&n);
        int arr[n];
        for(int i=0;i<n;i++){
                scanf("%d",&arr[i]);
        }
        for(int i=0;i<n-1;i++){
                for(int j=0;j<n-1;j++){
                        if(arr[j]>arr[j+1]){
                                int tmp =arr[j];
                                arr[j]=arr[j+1];
                                arr[j+1]=tmp;
                        }
                }
        }
        for(int i=0;i<n;i++){
                printf("%d ",arr[i]);
        }
}
```


Output:

Test case - 1
User Output
5
4 8 2 -3 0
-3 0 2 4 8

Test case - 2
User Output
8

-6 -2 -8 -36 0 69 -5 -96
-96 -36 -8 -6 -5 -2 0 69

Result:

Thus the above program is executed successfully and the output has been verified

Aim:

Given n items, each with a given **weight** and **value**, you need to determine the **maximum total value** that can be obtained by placing these items into a knapsack with a capacity of w.

Note: Utilize the Fractional Knapsack approach, where you can take fractional parts of each item. This allows you to split items and include only a portion of them in the knapsack to achieve the highest possible total value.

Input Format:

The first line contains two space-separated integers:

- n - the number of items.
- w - the maximum capacity of the knapsack.

The second line contains $2 \times n$ space-separated integers:

- Each pair of integers represents the value and weight of an item, respectively.

Output Format:

A single floating-point number, representing the maximum total value that can be obtained, is formatted to six decimal places.

Example 1:

Input:

```
3 50
60 10 100 20 120 30
```

Output:

```
240.000000
```

Explanation:

Take the item with a value of 60 and a weight of 10, along with the item valued at 100 with a weight of 20.
Since the remaining knapsack capacity is 20, a portion of the third item, which has a value of 120 and a weight of 30, is included.
The fraction taken is 120/30, resulting in a value of (120/30) * 20 = 80
Thus, the total maximum value that can be obtained is:
60+100+80=240.000000
This is the optimal value achievable within the given knapsack capacity.

Example 2:

Input:

```
2 50
60 10 100 20
```

Output:

```
160.000000
```

Explanation:

Take both items completely, without breaking. The total maximum value of the item we can have is 160.000000 from the given capacity of a sack.

Constraints:

$1 \leq n \leq 10^5$
$1 \leq w \leq 10^9$
$1 \leq value_i, weight_i \leq 10^4$

Program:

CTC31099.c

```c
#include <stdio.h>

void swap(int *a,int *b){
        int temp =*a;
        *a = *b;
        *b =temp;
}
void sort(int values[],int weight[],int n){
        double ratio[n];
        for(int i=0;i<n;i++){
                ratio[i]=(double)values[i]/weight[i];
        }
        for(int i=0;i<n-1;i++){
                for(int j=i+1;j<n;j++){
                        if(ratio[i]<ratio[j]){
                                double tmp = ratio[i];
                                ratio[i]=ratio[j];
                                ratio[j]=tmp;

                                swap(&values[i],&values[j]);
                                swap(&weight[i],&weight[j]);
                        }
                }
        }
}
double max(int values[],int weight[],int n,int capacity){
        sort(values,weight,n);
        double totalVal = 0.0;
        for(int i=0;i<n && capacity>0;i++){
                if(weight[i] <= capacity){
                        capacity -= weight[i];
                        totalVal += values[i];
                }
                else{
                        totalVal += (double)values[i]*capacity/weight[i];
                        break;
                }
        }
        return totalVal;
}
int main(){
        int n,capacity;
        scanf("%d %d",&n,&capacity);
        int values[n],weight[n];
        for(int i=0;i<n;i++){
                scanf("%d %d",&values[i],&weight[i]);
        }
        double MAX = max(values,weight,n,capacity);
        printf("%.6f\n",MAX);
        return 0;
}
```


Output:

Test case - 1

User Output
3 50
60 10 100 20 120 30
240.000000

Test case - 2
User Output
2 50
60 10 100 20
160.000000

Result:

Thus the above program is executed successfully and the output has been verified

Aim:

You are given a list of N activities, each with a start and end day. The start and end days for each activity are provided in two separate arrays: $start[]$ and $end[]$. Your task is to determine the maximum number of activities that can be completed by a single person. Please note the following:

- A person can only work on one activity per day.
- The duration of an activity includes both the start and end day.
- An activity can only be selected if it does not overlap with the previously selected activity.

Input Format:

- The first line contains an integer N representing the number of activities.
- The second line contains N space separated integers representing the start days.
- The third line contains N space separated integers representing the end days.

Constraints:

- $1 \leq N \leq 10^5$
- $1 \leq start[i] \leq end[i] \leq 10^9$

Output Format:

- Print the total number of activities that a person can do.

Example 1:

Input:

2 // N
2 1 // $start[]$ = {2, 1}
2 2 // $end[]$ = {2, 2}

Output:

1

Explanation: Given the overlapping activity durations, a person can only complete one activity.

Example 2:

Input:

4 // N
1 3 2 5 // $start[]$ = {1, 3, 2, 5}
2 4 3 6 // $end[]$ = {2, 4, 3, 6}

Output:

3

Explanation: In this case, a person can complete activities 1, 2, and 4, as their durations do not overlap.

Program:

CTC27805.c

```
#include <stdio.h>
void swap(int *a,int *b){
        int tmp = *a;
        *a =*b;
        *b =tmp;
}
void sort(int start[],int end[],int n){
        for(int i=0;i<n-1;i++){
                for(int j=i+1;j<n;j++){
                        if(end[i]>end[j]){
                                swap(&end[i],&end[j]);
                                swap(&start[i],&start[j]);
                        }
                }
        }
}
int max(int start[],int end[],int n){
        sort(start,end,n);
        int count =1;
        int lastEnd =end[0];
        for(int i=1;i<n;i++){
                if(start[i]>lastEnd){
                        count++;
                        lastEnd =end[i];
                }
        }
        return count;
}
int main(){
        int n;
        scanf("%d",&n);
        int start[n],end[n];
        for(int i=0;i<n;i++){
                scanf("%d",&start[i]);
        }
        for(int i =0;i<n;i++){
                scanf("%d",&end[i]);
        }
        int MAX = max(start,end,n);
        printf("%d\n",MAX);
        return 0;
}
```


Output:

Test case - 1
User Output
2
2 1
2 2
1

Test case - 2

User Output
4
1 3 2 5
2 4 3 6
3

Result:

Thus the above program is executed successfully and the output has been verified

Aim:

In an automobile company Franky as a manager havenjobs andmworkers. He has given three arrays:difficulty,profit, andworkerwhere:

- difficulty[i]andprofit[i]are the difficulty and the profit of thei^{th}job, and
- worker[j]is the ability ofj^{th}worker (i.e., thej^{th}worker can only complete a job with difficulty at mostworker[j]).

Every worker can be assigned**at most one job**, but one job can be**completed multiple times**.

- For example, if three workers attempt the same job that paysRs 1, then the total profit will beRs 3. If a worker cannot complete any job, their profit isRs 0.

Print the maximum profit we can achieve after assigning the workers to the jobs.

Sample test case 1:

Input:

5 4 ----------> n and m respectively
2 4 6 8 10 ----------> difficulty[]
10 20 30 40 50 ---------- profit[]
4 5 6 7 ----------> worker[]

Output:

100

Editorial: Workers are assigned jobs of difficulty [4,4,6,6] and they get a profit of [20,20,30,30] separately.

Sample test case 2:

Input:

3 3
85 47 57
24 66 99
40 25 25

Output:

0

Constraints:

- n == difficulty.length
- n == profit.length
- m == worker.length
- 1 <= n, m <= 10^4
- 1 <= difficulty[i], profit[i], worker[i] <= 10^5

Program:

CTC14700.c

```c
#include <stdio.h>
#include <stdlib.h>

void swap(int* a, int* b) {
        int temp = *a;
        *a = *b;
        *b = temp;
}
void sortJobs(int* difficulty, int* profit, int n) {
        for (int i = 0; i < n - 1; i++) {
                for (int j = 0; j < n - i - 1; j++) {
                        if (difficulty[j] > difficulty[j + 1]) {
                                swap(&difficulty[j], &difficulty[j + 1]);
                                swap(&profit[j], &profit[j + 1]);
                        }
                }
        }
}
void sortWorkers(int* workers, int m) {
        for (int i = 0; i < m - 1; i++) {
                for (int j = 0; j < m - i - 1; j++) {
                        if (workers[j] > workers[j + 1]) {
                                swap(&workers[j], &workers[j + 1]);
                        }
                }
        }
}
int maxProfit(int* difficulty, int* profit, int n, int* workers, int m) {
        sortJobs(difficulty, profit, n);
        sortWorkers(workers, m);
        int totalProfit = 0;
        int maxProfit = 0;
        int j = 0;
        for (int i = 0; i < m; i++) {
                while (j < n && workers[i] >= difficulty[j]) {
                        if (profit[j] > maxProfit) {
                                maxProfit = profit[j];
                        }
                        j++;
                }
                totalProfit += maxProfit;
        }
        return totalProfit;
}
int main() {
        int n, m;
        scanf("%d %d", &n, &m);
        int difficulty[n], profit[n], workers[m];
        for (int i = 0; i < n; i++) {
                scanf("%d", &difficulty[i]);
        }
        for (int i = 0; i < n; i++) {
                scanf("%d", &profit[i]);
        }
        for (int i = 0; i < m; i++) {
                scanf("%d", &workers[i]);
        }
```

```
    int result = maxProfit(difficulty, profit, n, workers, m);
    printf("%d\n", result);
    return 0;
}
```

Output:

Test case - 1
User Output
5 4
2 4 6 8 10
10 20 30 40 50
4 5 6 7
100

Test case - 2
User Output
3 3
85 47 57
24 66 99
40 25 25
0

Result:

Thus the above program is executed successfully and the output has been verified