S.No: 1	Exp. Name: ***Program in C for Recursive Linear Search***	Date: 2023-10-18

Aim:

AIM/OBJECTIVE: Program in ‘C’ for Recursive Linear Search

THEORY: Linear search is a search that finds an element in the list by searching the element sequentially until the element is found in the list.

ALGORITHM:
Linear Search (Array A, Value x)

```
Step 1: Set i to 1
Step 2: if i > n then go to step 7
Step 3: if A[i] = x then go to step 6
Step 4: Set i to i + 1
Step 5: Go to Step 2
Step 6: Print Element x Found at index i and go to step 8
Step 7: Print element not found
Step 8: Exit
```

Source Code:

`linearSearch.c`

```
#include<stdio.h>
int recu(int arr[] , int val, int idx,int n){
        int pos = 0;
        if(idx>=n){
                return 0;
        }
        else if(arr[idx] == val){
                pos = idx+1;

        }
        else{
                return recu(arr,val,idx+1,n);
        }
        return pos;
}
int main(){
        int n ,val,pos,m=0,arr[100];
        printf("Enter the total elements in the array: ");
        scanf("%d",&n);
        printf("Enter the array elements: ");
        for(int i=0;i<n;i++)
        {
                scanf("%d",&arr[i]);
        }
        printf("Enter the element to search: ");
                scanf("%d",&val);

                pos = recu(arr,val,0,n);
                if(pos!=0){
                        printf("Element found at position %d ",pos);
                }
                else{
                        printf("Element not found");
                }
        return 0;
}
```

Execution Results - All test cases have succeeded!

Test Case - 1

User Output

```
Enter the total elements in the array:
7
Enter the array elements:
1 2 4 5 6 7 8
Enter the element to search:
6
Element found at position 5
```

Test Case - 2

User Output

```
Enter the total elements in the array:
3
Enter the array elements:
5 6 9
Enter the element to search:
4
Element not found
```

S.No: 2	Exp. Name: ***Program in C for Recursive Binary Search***	**Date: 2023-10-18**

Aim:

AIM/OBJECTIVE: Program in ‘C’ for Recursive Binary Search

THEORY: On the other hand, a binary search is a search that finds the middle element in the list recursively until the middle element is matched with a searched element.

Binary Search Algorithm:

STEP1: Find the midpoint of the array; this will be the element at arr[size/2]. The midpoint divides the array into two smaller arrays: the lower half of the array consisting of elements 0 to midpoint - 1, and the upper half of the array consisting of elements midpoint to size - 1.
STEP 2:Compare key to arr[midpoint] by calling the user function cmp_proc.
STEP 3: If the key is a match, return arr[midpoint]; otherwise
STEP 4: If the array consists of only one element return NULL, indicating that there is no match; otherwise
STEP 5: If the key is less than the value extracted from arr[midpoint] search the lower half of the array by recursively calling search; otherwise
STEP 6: Search the upper half of the array by recursively calling search

Source Code:

```
binarySearch.c
```

```
#include<stdio.h>
int binary(int arr[],int size,int key)
{
        int first=0;
        int last=size-1;
        while(first<=last)
        {
                int mid=(first+last)/2;
                if(arr[mid]==key)
                {
                        return mid;
                }
                else if(arr[mid]<key)
                {
                        first=mid+1;
                }
                else
                {
                        last=mid-1;
                }
        }
        return -1;
}
int main()
{
        int arr[100],size,key;
        printf("Enter size of a list: ");
        scanf("%d",&size);
        printf("Enter elements: ");
        for(int i=0;i<size;i++)
        {
                scanf("%d",&arr[i]);
        }
        printf("Enter key to search: ");
        scanf("%d",&key);
        int binarys=binary(arr,size,key);
        if(binarys==-1)
        {
                printf("Key not found\n");
        }
        else
        {
                printf("Key found\n");
        }
        return 0;
}
```

Execution Results - All test cases have succeeded!

Test Case - 1

User Output

```
Enter size of a list:
4
```

```
Enter elements:
2 3 4 5
Enter key to search:
3
Key found
```

Test Case - 2

User Output

```
Enter size of a list:
3
Enter elements:
5 6 9
Enter key to search:
10
Key not found
```

S.No: 3	Exp. Name: ***Write a Program in 'C' for Selection Sort***	Date: 2023-10-25

Aim:

OBJECTIVE: Write a Program in 'C' for Selection Sort

THEORY: In selection sort, the smallest value among the unsorted elements of the array is selected in every pass and inserted to its appropriate position into the array.

First, find the smallest element of the array and place it on the first position. Then, find the second smallest element of the array and place it on the second position. The process continues until we get the sorted array.

The array with n elements is sorted by using n-1 pass of selection sort algorithm.

1. In 1st pass, smallest element of the array is to be found along with its index pos. then, swap A[0] and A[pos]. Thus A[0] is sorted, we now have n -1 elements which are to be sorted.
2. In 2nd pas, position pos of the smallest element present in the sub-array A[n-1] is found. Then, swap, A[1] and A[pos]. Thus A[0] and A[1] are sorted, we now left with n-2 unsorted elements.
3. In n-1th pass, position pos of the smaller element between A[n-1] and A[n-2] is to be found. Then, swap, A[pos] and A[n-1].
4. Therefore, by following the above explained process, the elements A[0], A[1], A[2],...., A[n-1] are sorted.

ALGORITHM:

```
SELECTION SORT(ARR, N)
Step 1: Repeat Steps 2 and 3 for K = 1 to N-1
Step 2: CALL SMALLEST(ARR, K, N, POS)
Step 3: SWAP A[K] with ARR[POS]
[END OF LOOP]
Step 4: EXIT
SMALLEST (ARR, K, N, POS)
Step 1: [INITIALIZE] SET SMALL = ARR[K]
Step 2: [INITIALIZE] SET POS = K
Step 3: Repeat for J = K+1 to N -1
IF SMALL > ARR[J]
SET SMALL = ARR[J]
SET POS = J
[END OF IF]
[END OF LOOP]
Step 4: RETURN POS
```

Source Code:

```
selectionSort.c
```

```c
#include<stdio.h>
void selection_sort(int arr[],int size)
{
	for(int i=0;i<size-1;i++)
	{
		int min_idx=i;
		for(int j=i+1;j<size;j++)
		{
			if(arr[j]<arr[min_idx]){
			min_idx=j;
			}
		}
		int temp=arr[i];
		arr[i]=arr[min_idx];
		arr[min_idx]=temp;
	}
}
void main()
{
	int arr[100],size;
	printf("Enter the total elements in the array: ");
	scanf("%d",&size);
	printf("Enter the array elements: ");
	for(int i=0;i<size;i++)
	{
		scanf("%d",&arr[i]);
	}
	selection_sort(arr,size);
	printf("printing sorted elements...\n");
	for(int i=0;i<size;i++)
	{
		printf("%d ",arr[i]);
	}
}
```

Execution Results - All test cases have succeeded!

Test Case - 1

User Output

```
Enter the total elements in the array:
4
Enter the array elements:
10 9 7 101
printing sorted elements...
7 9 10 101
```

Test Case - 2

User Output

```
Enter the total elements in the array:
```

```
Enter the array elements:
1 9 5 78
printing sorted elements...
1 5 9 78
```

S.No: 4	Exp. Name: ***Write a Program in ‘C’ for Insertion Sort***	**Date: 2023-11-01**

Aim:

OBJECTIVE: Write a Program in ‘C’ forInsertion Sort

THEORY: Insertion sort is the simple sorting algorithm which is commonly used in the daily lives while ordering a deck of cards. In this algorithm, we insert each element onto its proper place in the sorted array. This is less efficient than the other sort algorithms like quick sort, merge sort, etc.

ALGORITHM:

```
Step 1: Repeat Steps 2 to 5 for K = 1 to N-1
Step 2: SET TEMP = ARR[K]
Step 3: SET J = K - 1
Step 4: Repeat while TEMP <=ARR[J]
SET ARR[J + 1] = ARR[J]
SET J = J - 1
[END OF INNER LOOP]
Step 5: SET ARR[J + 1] = TEMP
[END OF LOOP]
Step 6: EXIT
```

Source Code:

insertionSort.c

```
#include<stdio.h>

void main()
{
	int a[100];
	int i,j,n;
	int temp;
	printf("Enter the total elements in the array: ");
	scanf("%d",&n);
	printf("Enter the array elements: ");
	for(i=0;i<n;i++)
	{
		scanf("%d",&a[i]);
	}
	printf("printing sorted elements...\n");
	for(i=1;i<n;i++)
	{
		temp=a[i];
		j=i-1;
		while(j>=0 && temp<=a[j])
		{
			a[j+1]=a[j];
			j=j-1;
		}
			a[j+1]=temp;
	}
	for(i=0;i<n;i++)
	{
		printf("%d ",a[i]);
	}
}
```

Execution Results - All test cases have succeeded!

Test Case - 1

User Output

```
Enter the total elements in the array:
4
Enter the array elements:
10 9 7 101
printing sorted elements...
7 9 10 101
```

Test Case - 2

User Output

```
Enter the total elements in the array:
3
Enter the array elements:
-9 -78 -90
printing sorted elements...
-90 -78 -9
```

S.No: 5	Exp. Name: ***Write a Program in 'C' for Heap Sort***	**Date: 2023-12-13**

Aim:

OBJECTIVE: Write a Program in 'C' for Heap Sort

THEORY: Heap sort processes the elements by creating the min heap or max heap using the elements of the given array. Min heap or max heap represents the ordering of the array in which root element represents the minimum or maximum element of the array. At each step, the root element of the heap gets deleted and stored into the sorted array and the heap will again be heapified.

The heap sort basically recursively performs two main operations.

- Build a heap H, using the elements of ARR.
- Repeatedly delete the root element of the heap formed in phase 1.
-

ALGORITHM:

```
HEAP_SORT(ARR, N)
Step1: [Build Heap H]
Repeat for i=0 to N-1
CALL INSERT_HEAP(ARR, N, ARR[i])
[END OF LOOP]
Step 2: Repeatedly Delete the root element
Repeat while N > 0
CALL Delete_Heap(ARR,N,VAL)
SET N = N+1
[END OF LOOP]
Step 3: END
```

Source Code:

`heapSort.c`

```
#include<stdio.h>
int temp;
void heapify(int arr[], int size, int i)
{
        int largest = i; int left = 2*i + 1;
        int right = 2*i + 2;
        if (left < size && arr[left] >arr[largest])
        largest = left;
        if (right < size && arr[right] > arr[largest])
        largest = right;
        if (largest != i)
        {
                temp = arr[i];
                arr[i]= arr[largest];
                arr[largest] = temp;
                heapify(arr, size, largest);
        }
}
void heapSort(int arr[], int size)
{
        int i;
        for (i = size / 2 - 1; i >= 0; i--)
        heapify(arr, size, i);
        for (i=size-1; i>=0; i--) {
                temp = arr[0];
                arr[0]= arr[i];
                arr[i] = temp;
                heapify(arr, i, 0);
        }
}
void main() {
        int arr[100] ;
        int i;
        int size;
        printf("Enter array size : ");
        scanf("%d",&size);
        printf("Enter %d elements : ",size);
        for(i=0; i<size; i++)
        scanf("%d",&arr[i]);
        printf("Before sorting the elements are : ");
        for(i=0; i<size; i++)
        printf("%d ",arr[i]);
        heapSort(arr, size);
        printf("\nAfter sorting the elements are : ");
        for (i=0; i<size; i++) {
                printf("%d ",arr[i]);
        } printf("\n");
        return 0;
}
```

Execution Results - All test cases have succeeded!

Test Case - 1

User Output

```
Enter array size :
10
Enter 10 elements :
1 10 2 3 4 1 2 100 23 2
Before sorting the elements are : 1 10 2 3 4 1 2 100 23 2
After sorting the elements are : 1 1 2 2 2 3 4 10 23 100
```

S.No: 6	Exp. Name: ***Write a Program in 'C' for Merge Sort***	**Date: 2023-12-13**

Aim:

OBJECTIVE: Write a Program in 'C' for Merge Sort

THEORY: Merge sort is the algorithm which follows divide and conquer approach. Consider an array A of n number of elements. The algorithm processes the elements in 3 steps.

5. If A Contains 0 or 1 elements then it is already sorted, otherwise, Divide A into two sub-array of equal number of elements.
6. Conquer means sort the two sub-arrays recursively using the merge sort.
7. Combine the sub-arrays to form a single final sorted array maintaining the ordering of the array.

The main idea behind merge sort is that, the short list takes less time to be sorted.

ALGORITHM:

```
MERGE Step 1: [INITIALIZE] SET I = BEG, J = MID + 1, INDEX = 0
Step 2: Repeat while (I <= MID) AND (J<=END)
IF ARR[I] < ARR[J]
SET TEMP[INDEX] = ARR[I]
SET I = I + 1
ELSE
SET TEMP[INDEX] = ARR[J]
SET J = J + 1
[END OF IF]
SET INDEX = INDEX + 1
[END OF LOOP]
Step 3: [Copy the remaining elements of right sub-array, if any]
IF I > MID
Repeat while J <= END
SET TEMP[INDEX] = ARR[J]
SET INDEX = INDEX + 1, SET J = J + 1
[END OF LOOP]
[Copy the remaining elements of left sub-array, if any]
ELSE
Repeat while I <= MID
SET TEMP[INDEX] = ARR[I]
SET INDEX = INDEX + 1, SET I = I + 1
[END OF LOOP]
[END OF IF]
Step 4: [Copy the contents of TEMP back to ARR] SET K = 0
Step 5: Repeat while K < INDEX
SET ARR[K] = TEMP[K]
SET K = K + 1
[END OF LOOP]
Step 6: Exit
MERGE_SORT(ARR, BEG, END)
```

Source Code:

`mergeSort.c`

```
#include<stdio.h>
int PrintArray(int *Array, int Size){
        for(int i=0;i<Size;i++)
        {
                printf("%d ",Array[i]);
        }
printf("\n");
}
void Merge(int *Array, int p, int q, int r){
        int n1 = q-p +1;
        int n2 = r-q;
        int L[n1], M[n2];
        for(int i=0; i<n1; i++){
                L[i] = Array[p+i];
        }
        for(int j=0; j<n2; j++){
                M[j] = Array[q+j+1];
        }
        int i, j, k;
        i =0;
        j=0;
        k=p;
        while(i<n1 && j<n2){
                if(L[i] <= M[j]){
                        Array[k] = L[i];
                        i++; }
                        else{
                                Array[k] = M[j];
                                j++; }
                                k++; }
                                while(i<n1){
                                        Array[k] = L[i];
                                        i++;
                                        k++; }
                                        while(j < n2){
                                                Array[k] = M[j];
                                                j++;
                                                k++; }
}
void MergeSort(int *Array, int Left, int Right){
        if(Left < Right){
                int Mid = Left +(Right-Left)/2;
                MergeSort(Array, Left, Mid);
                MergeSort(Array, Mid+1, Right);
                Merge(Array, Left, Mid, Right); }
}
int main(){
        int Array[50], i, Size;
        printf("Enter array size : ");
        scanf("%d",&Size);
        printf("Enter %d elements : ",Size);
        for(i=0; i<Size; i++){
                scanf("%d",&Array[i]);
        }
        printf("Before sorting the elements are : ");
        printf("After sorting the elements are : ");
        PrintArray(Array, Size);
        return 0; }
```

Execution Results - All test cases have succeeded!

Test Case - 1

User Output

```
Enter array size :
5
Enter 5 elements :
2 5 4 6 7
Before sorting the elements are : 2 5 4 6 7
After sorting the elements are : 2 4 5 6 7
```

Test Case - 2

User Output

```
Enter array size :
3
Enter 3 elements :
23 15 0
Before sorting the elements are : 23 15 0
After sorting the elements are : 0 15 23
```

S.No: 7	Exp. Name: ***Write a Program in 'C' for Quick Sort***	**Date: 2023-12-13**

Aim:
OBJECTIVE: Write a Program in 'C' forQuick Sort

THEORY: Like Merge Sort, QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

8. Always pick first element as pivot.
9. Always pick last element as pivot (implemented below)
10. Pick a random element as pivot.
11. Pick median as pivot.

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

ALGORITHM:

SOURCE CODE:

```
PARTITION (ARR, BEG, END, LOC)
Step 1: [INITIALIZE] SET LEFT = BEG, RIGHT = END, LOC = BEG, FLAG =
Step 2: Repeat Steps 3 to 6 while FLAG =
Step 3: Repeat while ARR[LOC] <=ARR[RIGHT]
AND LOC != RIGHT
SET RIGHT = RIGHT - 1
[END OF LOOP]
Step 4: IF LOC = RIGHT
SET FLAG = 1
ELSE IF ARR[LOC] > ARR[RIGHT]
SWAP ARR[LOC] with ARR[RIGHT]
SET LOC = RIGHT
[END OF IF]
Step 5: IF FLAG = 0
Repeat while ARR[LOC] >= ARR[LEFT] AND LOC != LEFT
SET LEFT = LEFT + 1
[END OF LOOP]
Step 6:IF LOC = LEFT
SET FLAG = 1
ELSE IF ARR[LOC] < ARR[LEFT]
SWAP ARR[LOC] with ARR[LEFT]
SET LOC = LEFT
[END OF IF]
[END OF IF]
Step 7: [END OF LOOP]
Step 8: END
```

QUICK_SORT (ARR, BEG, END)

```
Step 1: IF (BEG < END)
CALL PARTITION (ARR, BEG, END, LOC)
CALL QUICKSORT(ARR, BEG, LOC - 1)
CALL QUICKSORT(ARR, LOC + 1, END)
[END OF IF]
Step 2: END
```

Source Code:

quickSort.c

```c
#include<stdio.h>
void PrintArray(int *Array, int Size){
        for(int i=0; i<Size; i++){
                printf("%d ",Array[i]);
        }
        printf("\n");
}
void Swap(int *a, int *b){
        int temp = *a;
        *a = *b;
        *b = temp;
}
int Partition(int *Array, int Low, int High){
        int Pivot = Array[High];
        int i = Low-1;
        for(int j = Low; j<High; j++){
                if(Array[j] <= Pivot){
                        i++;
                        Swap(&Array[i], &Array[j]);
                }
        }
        Swap(&Array[i+1], &Array[High]);
        return i+1;
}
void QuickSort(int *Array, int Low, int High){
        if(Low<High){
                int Pivot = Partition(Array, Low, High);
                QuickSort(Array, Low, Pivot-1);
                QuickSort(Array, Pivot+1, High); }
}
void main()
{
        int Size,i,Array[100];
printf("Enter array size : ");
scanf("%d",&Size);
printf("Enter %d elements : ",Size);
for(i=0; i<Size; i++){
        scanf("%d",&Array[i]);
}
printf("Before sorting the elements are : ");
PrintArray(Array,Size);
QuickSort(Array, 0, Size-1);
printf("After sorting the elements are : ");
PrintArray(Array,Size);
return 0;
}
```

Execution Results - All test cases have succeeded!

Test Case - 1

User Output

```
Enter array size :
5
Enter 5 elements :
12 35 78 96 5
Before sorting the elements are : 12 35 78 96 5
After sorting the elements are : 5 12 35 78 96
```

Test Case - 2

User Output

```
Enter array size :
3
Enter 3 elements :
-56 -4 -3
Before sorting the elements are : -56 -4 -3
After sorting the elements are : -56 -4 -3
```

S.No: 8	Exp. Name: ***Write a Program in 'C' for Counting Sort***	Date: 2023-12-13

Aim:

OBJECTIVE: Write a Program in 'C' forCounting Sort

THEORY: Counting sort is a sorting technique based on keys between a specific range. Itsorts the elements of an array by counting the number of occurrences of each unique element in the array. The count is stored in an auxiliary array and the sorting is done by mapping the count as an index of the auxiliary array..

ALGORITHM:

```
Counting Sort (array P, array Q, int k)
1. For i ← 1 to k
2. do C [i] ← 0 to k
3. for j← 1 to length [A]
4. do C[A[j]] ← C [A [j]]+1[θ(n) times]
5.for i← 2 to k
6. do C [i]←C [i] + C[i-1] [θ(k) times]
7.for j ← length [A] down to 1 [θ(n) times]
8. do B[C[A[j]←A [j]
9. C[A[j]←C[A[j]-1
```

Source Code:

`countingSort.c`

```
#include<stdio.h>
void PrintArray(int *Array, int Size){
	for(int i=0; i<Size; i++){
		printf("%d ", Array[i]);
	}
	printf("\n");
}
void CountingSort(int *Array, int Size){
	int Output[50];
	int Max = Array[0];
	for(int i =1 ; i<Size; i++){
		if(Array[i] > Max){
			Max = Array[i];
		}
	}
	int Count[50];
	for(int i=0; i<=Max; i++){
		Count[i] = 0;
	}
	for(int i=0; i<Size; i++){
		Count[Array[i]]++;
	}
	for(int i=1; i<=Max; i++){
		Count[i] +=Count[i-1];
	}
	for(int i =Size-1; i>=0; i--){
		Output[Count[Array[i]] -1] = Array[i];
		Count[Array[i]]--; }
		for(int i=0; i<Size; i++){
			Array[i] = Output[i];
		}
}
int main(){
	int Array[50], i, Size;
	printf("Enter the number of inputs : ");
	scanf("%d",&Size);
	printf("Enter the elements to be sorted : ");
	for(i=0; i<Size; i++){
		scanf("%d",&Array[i]);
	}
	CountingSort(Array, Size);
	printf("The Sorted array is : ");
	PrintArray(Array, Size);
}
```

Execution Results - All test cases have succeeded!

Test Case - 1

User Output

```
Enter the number of inputs : 
8
Enter the elements to be sorted : 
```

```
8 4 2 2 8 3 3 1
The Sorted array is : 1 2 2 3 3 4 8 8
```

Test Case - 2

User Output

```
Enter the number of inputs :
3
Enter the elements to be sorted :
4 9 1
The Sorted array is : 1 4 9
```

S.No: 9	Exp. Name: ***Write a Program in 'C' for Travelling Salesman Problem using Dynamic Programming***	**Date: 2023-12-13**

Aim:

OBJECTIVE: Write a Program in 'C' for Travelling Salesman Problem using Dynamic Programming

THEORY: In the traveling salesman Problem, a salesman must visits n cities. We can say that salesman wishes to make a tour or Hamiltonian cycle, visiting each city exactly once and finishing at the city he starts from. There is a non-negative cost c (i, j) to travel from the city i to city j. The goal is to find a tour of minimum cost. We assume that every two cities are connected. Such problems are called Traveling-salesman problem (TSP).

ALGORITHM:

```
Algorithm: Traveling-Salesman-Problem
C ({1}, 1) = 0
for s = 2 to n do
for all subsets S Є {1, 2, 3, … , n} of size s and containing 1
C (S, 1) = ∞
for all j Є S and j ≠ 1
C (S, j) = min {C (S – {j}, i) + d(i, j) for i Є S and i ≠ j}
Return minj C ({1, 2, 3, …, n}, j) + d(j, i)
```

Source Code:

`travellingSalesMan.c`

```
#include<stdio.h>
int ary[10][10],completed[10],n,cost=0;
void takeInput()
{
        int i,j;
        printf("Enter the number of villages: ");
        scanf("%d",&n);
        printf("Enter the Cost Matrix\n");
        for(i=0;i < n;i++)
        {
                printf("Enter Elements of Row: %d\n",i+1);
                for( j=0;j < n;j++)
                scanf("%d",&ary[i][j]);
                completed[i]=0;
        }
        printf("The cost list is:");
        for( i=0;i < n;i++)
        {
                printf("\n");
                for(j=0;j < n;j++)
                printf("\t%d",ary[i][j]);
        }
}
void mincost(int city)
{
        int i,ncity;
        completed[city]=1;
        printf("%d--->",city+1);
        ncity=least(city);
        if(ncity==999) {
                ncity=0;
                printf("%d",ncity+1);
                cost+=ary[city][ncity];
                return;
        }
        mincost(ncity);
}
int least(int c) {
        int i,nc=999;
        int min=999,kmin;
        for(i=0;i < n;i++) {
                if((ary[c][i]!=0)&&(completed[i]==0))
                if(ary[c][i]+ary[i][c] < min) {
                        min=ary[i][0]+ary[c][i];
                        kmin=ary[c][i];
                        nc=i;
                }
        }
        if(min!=999)
        cost+=kmin;
        return nc;
}
int main() {
        takeInput();
        printf("\nThe Path is:\n");
        return 0;
}
```

Execution Results - All test cases have succeeded!

Test Case - 1

User Output

```
Enter the number of villages:
4
Enter the Cost Matrix
Enter Elements of Row: 1
0 4 1 3
Enter Elements of Row: 2
4 0 2 1
Enter Elements of Row: 3
1 2 0 5
Enter Elements of Row: 4
3 1 5 0
The cost list is:
        0       4       1       3
        4       0       2       1
        1       2       0       5
        3       1       5       0
The Path is:
1--->3--->2--->4--->1
Minimum cost is 7
```

S.No: 10	Exp. Name: ***Write a Program for Find Minimum Spanning Tree using Kruskal's Algorithm***	**Date: 2023-12-14**

Aim:

OBJECTIVE: Write a Program forFind Minimum Spanning Tree using Kruskal's Algorithm

THEORY: Given a connected and undirected graph, a spanning tree of that graph is a subgraph that is a tree and connects all the vertices together. A single graph can have many different spanning trees. A minimum spanning tree (MST) or minimum weight spanning tree for a weighted, connected, undirected graph is a spanning tree with a weight less than or equal to the weight of every other spanning tree. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree.

ALGORITHM:

```
Algorithm Kruskal(G,w)
{
A := ∅ for each vertex v ∈ V[G]
do MAKE-SET(v)
Sort the edges of E into nondecreasing order by weight w
for each edge (u, v) ∈ E, taken in nondecreasing order by weight do
if FIND-SET(u) = FIND-SET(v) then
{
A:= A ∪ {(u, v)}
UNION(u, v)
}
return A
```

Source Code:

`kruskal'sAlgorithm.c`

```
#include<stdio.h>
int i, j, k, n,a, b, u, v, ne=1;
int min , mincost =0, cost[9][9], parent[9];
int find(int i){
        while(parent[i])
        i = parent[i];
        return i;
}
int uni(int i, int j){
        if(i!=j){
                return 1;
        }
        return 0;
        }
        int main(){
                printf("Implementation of Kruskal's algorithm\n");
                printf("Enter the no. of vertices: ");
                scanf("%d",&n);
                printf("Enter the cost adjacency matrix\n");
                for(i=1; i<=n; i++){
                        for(j=1; j<=n; j++){
                                scanf("%d",&cost[i][j]);
                                if(cost[i][j] ==0)
                                cost[i][j] = 999;
                        }
                }
                printf("The edges of Minimum Cost Spanning Tree are: ");
                while(ne<n){
                        for(i=1, min =999; i<=n; i++){
                                for(j=1; j<=n; j++){
                                        if(cost[i][j] <min){
                                                min = cost[i][j];
                                                a=u=i;
                                                b=v=j;
                                        }
                                }
                        }
                        u= find(u);
                        v=find(v);
                        if(uni(u,v)){
                                printf("%d edge (%d,%d) =%d\n",ne++,a,b,min);
                                mincost +=min;
                        }
                        cost[a][b] = cost[b][a] = 999;
                }
                printf("Minimum cost = %d\n",mincost);
                return 0;
        }
```

Execution Results - All test cases have succeeded!

Test Case - 1
User Output

```
Implementation of Kruskal's algorithm
Enter the no. of vertices:
4
Enter the cost adjacency matrix
0 20 10 50
20 0 60 999
10 60 0 40
50 999 40 0
The edges of Minimum Cost Spanning Tree are: 1 edge (1,3) =10
2 edge (1,2) =20
3 edge (3,4) =40
Minimum cost = 70
```

S.No: 11	Exp. Name: ***Write a Program for Implement N Queen Problem using Backtracking***	**Date: 2023-12-14**

Aim:

OBJECTIVE: Write a Program for Implement N Queen Problem using Backtracking

THEORY: In the search for fundamental principles of algorithm design, backtracking represents one of the most general techniques. Many problems which deal with searching for a set of solutions or which ask for an optimal solution satisfying some constraints can be solved using the backtracking formation.
A clasic combinatorial problem is to place eight queens on an 8x8 chessboard so that no two “attack” that is, so that no two of them are on the same row, column, or diagonal

ALGORITHM:

```
Algorithm NQueens(k,n)
// Using backtracking, this procedure prints all possible placements of n queens on
// an n*n chessboard so that they are nonattacking.
{
for i:=1 to n do
{
if(Place(k,i) then
{
x[k]:=i;
if(k=n) then write(x[1:n]);
}
}
}
Algorithm Place(k,i)
// Returns true if a queen can be placed in kth row and ith column. Otherwise it
// returns false. X[] is a gloabal array whose first (k-1) values have been set. Abs( r )
// returns the absolute value of r.
{
for j:= 1 to k-1 do
if((x[j]=i) //Two in the same column
or(Abs(x[j]-i) = Abs(j-k))) // or in the same diagonal
then return false;
return true;
}
```

Source Code:

`NQueenProblem.c`

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
int a[30],count=0;
int place(int pos)
{
        int i;
        for(i=1;i<pos;i++)
        {
                if((a[i]==a[pos])||((abs(a[i]- a[pos])==abs(i-pos))))
                return 0;
        }
        return 1;
}
void print_sol(int n) {
        int i,j;
        count++;
        printf("Solution #%d:\n",count);
        for(i=1;i<=n;i++) {
                for(j=1;j<=n;j++) {
                        if(a[i]==j)
                        printf("Q\t");
                        else
                        printf("*\t");
                }
                printf("\n");
        }
}
void queen(int n) {
        int k=1;
        a[k]=0;
        while(k!=0) {
                a[k]=a[k]+1;
                while((a[k]<=n)&&!place(k))
                a[k]++;
                if(a[k]<=n) {
                        if(k==n)
                        print_sol(n);
                        else {
                                k++;
                                a[k]=0;
                        }
                }
                else
                k--;
        }
}
void main( ) {
        int i,n;
        printf("Enter the number of Queens\n");
        scanf("%d",&n);
        queen(n);
        printf("Total solutions=%d",count);
}
```

Execution Results - All test cases have succeeded!

Test Case - 1

User Output

```
Enter the number of Queens
4
Solution #1:
*	Q	*	*
*	*	*	Q
Q	*	*	*
*	*	Q	*
Solution #2:
*	*	Q	*
Q	*	*	*
*	*	*	Q
*	Q	*	*
Total solutions=2
```

Test Case - 2

User Output

```
Enter the number of Queens
3
Total solutions=0
```

Test Case - 3

User Output

```
Enter the number of Queens
5
Solution #1:
Q	*	*	*	*
*	*	Q	*	*
*	*	*	*	Q
*	Q	*	*	*
*	*	*	Q	*
Solution #2:
Q	*	*	*	*
*	*	*	Q	*
*	Q	*	*	*
*	*	*	*	Q
*	*	Q	*	*
Solution #3:
*	Q	*	*	*
*	*	*	Q	*
Q	*	*	*	*
*	*	Q	*	*
*	*	*	*	Q
Solution #4:
```

```
*       *       *       *       Q
*       *       Q       *       *
Q       *       *       *       *
*       *       *       Q       *
Solution #5:
*       *       Q       *       *
Q       *       *       *       *
*       *       *       Q       *
*       Q       *       *       *
*       *       *       *       Q
Solution #6:
*       *       Q       *       *
*       *       *       *       Q
*       Q       *       *       *
*       *       *       Q       *
Q       *       *       *       *
Solution #7:
*       *       *       Q       *
Q       *       *       *       *
*       *       Q       *       *
*       *       *       *       Q
*       Q       *       *       *
Solution #8:
*       *       *       Q       *
*       Q       *       *       *
*       *       *       *       Q
*       *       Q       *       *
Q       *       *       *       *
Solution #9:
*       *       *       *       Q
*       Q       *       *       *
*       *       *       Q       *
Q       *       *       *       *
*       *       Q       *       *
Solution #10:
*       *       *       *       Q
*       *       Q       *       *
Q       *       *       *       *
*       *       *       Q       *
*       Q       *       *       *
Total solutions=10
```

S.No: 12	Exp. Name: ***Implement 0/1 Knapsack Problem by Dynamic Programming***	**Date: 2023-12-14**

Aim:

AIM/OBJECTIVE: Implement 0/1 Knapsack Problem by Dynamic Programming.

Theory:

- You are given the following - A knapsack (kind of shoulder bag) with a limited weight capacity.

Few items each having some weight and value.

The problem states - Which items should be placed into the knapsack such that-

- The value or profit obtained by putting the items into the knapsack is maximum.
- And the weight limit of the knapsack does not exceed.

Algorithm:

```
KNAPSACK (n, W)
1. for w = 0, W
2. do V [0,w] ← 0
3. for i=0, n
4. do V [i, 0] ← 0
5. for w = 0, W
6. do if (w i ≤ w &amp; v i + V [i-1, w - w i ]&gt; V [i -1,W])
7. then V [i, W] ← v i + V [i - 1, w - w i ]
8. else V [i, W] ← V [i - 1, w]
```

Source Code:

`Knapsack.c`

```c
#include<stdio.h>
int max(int a, int b)
{
        return (a > b)? a : b;
}
int knapSack(int W, int wt[], int val[], int n)
{
        int i, w;
        int K[n+1][W+1];
        for (i = 0; i <= n; i++)
        {
                for (w = 0; w <= W; w++)
                {
                        if (i==0 || w==0)
                        K[i][w] = 0;
                        else if (wt[i-1] <= w)
                        K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]], K[i-1][w]);
                        else
                        K[i][w] = K[i-1][w];
                }
        }
        return K[n][W];
}
int main()
{
        int i, n, val[20], wt[20], W;
        printf("Enter number of items: ");
        scanf("%d", &n);
        printf("Enter value and weight of items:\n");
        for(i = 0;i < n; ++i) {
                scanf("%d%d", &val[i], &wt[i]);
        }
        printf("Enter size of knapsack: ");
        scanf("%d", &W);
        printf("%d", knapSack(W, wt, val, n));
        return 0;
}
```

Execution Results - All test cases have succeeded!

Test Case - 1
User Output
Enter number of items:
4
Enter value and weight of items:
40 20
60 50
80 70
90 10
Enter size of knapsack:
30

```
130
```

Test Case - 2

User Output

```
Enter number of items:
3
Enter value and weight of items:
100 20
50 10
150 30
Enter size of knapsack:
50
250
```

S.No: 13	Exp. Name: ***Implement Fractional Knapsack Problem by Greedy Method***	**Date: 2023-12-14**

Aim:

OBJECTIVE: Implement Fractional Knapsack Problem by Greedy Method

Theory: Fractions of items can be taken rather than having to make binary (0-1) choices for each item. Fractional Knapsack Problem can be solvable by greedy strategy whereas 0 - 1 problem is not.

Algorithm:

```
or i = 1 to n
do x[i] = 0
weight = 0
for i = 1 to n
if weight + w[i] ≤ W then
x[i] = 1
weight = weight + w[i]
else
x[i] = (W - weight) / w[i]
weight = W
break
return x
```

Source Code:

knapsack.c

```
#include <stdio.h>
void main()
{
        int capacity, no_items, cur_weight, item;
        int used[10];
        float total_profit;
        int i;
        int weight[10];
        int value[10];
        printf("Enter the capacity of knapsack: ");
        scanf("%d", &capacity);
        printf("Enter the number of items: ");
        scanf("%d", &no_items);
        printf("Enter the weight and value of %d item:\n", no_items);
        for (i = 0; i < no_items; i++)
        {
                printf("Weight[%d]:\t", i);
                scanf("%d", &weight[i]);
                printf("Value[%d]:\t", i);
                scanf("%d", &value[i]);
        }
        for (i = 0; i < no_items; ++i)
        used[i] = 0;
        cur_weight = capacity;
        while (cur_weight > 0)
        {
                item = -1;
                for (i = 0; i < no_items; ++i)
                if ((used[i] == 0) && ((item == -1) || ((float)value[i] / weight[i] >
(float)value[item] / weight[item])))
                item = i;
                used[item] = 1;
                cur_weight -= weight[item];
                total_profit += value[item];
                if (cur_weight >= 0)
                printf("Added object %d (%d Rs., %dKg) completely in the bag. Space left:
%d.\n", item + 1, value[item], weight[item], cur_weight);
                else
                {
                        int item_percent = (int)((1 + (float)cur_weight / weight[item]) *
100);
                        printf("Added %d%% (%d Rs., %dKg) of object %d in the bag.\n",
item_percent,
                        value[item], weight[item], item + 1);
                        total_profit -= value[item];
                        total_profit += (1 + (float)cur_weight / weight[item]) *
value[item];
                        }
                        }
                        printf("Filled the bag with objects worth %.2f Rs.\n",
total_profit);
                        }
```

Execution Results - All test cases have succeeded!

Test Case - 1

User Output

```
Enter the capacity of knapsack:
50
Enter the number of items:
3
Enter the weight and value of 3 item:
Weight[0]:
30
Value[0]:
5
Weight[1]:
20
Value[1]:
50
Weight[2]:
35
Value[2]:
12
Added object 2 (50 Rs., 20Kg) completely in the bag. Space left: 30.
Added 85% (12 Rs., 35Kg) of object 3 in the bag.
Filled the bag with objects worth 60.29 Rs.
```

S.No: 15	Exp. Name: ***Find Minimum Cost Spanning Tree of a given connected undirected graph using Kruskal's algorithm***	**Date: 2023-12-14**

Aim:
Objective: Find Minimum Cost Spanning Tree of a given connected undirected graph using Kruskal's algorithm

THEORY: Kruskal's Algorithm is used to find the minimum spanning tree for a connected weighted graph. The main target of the algorithm is to find the subset of edges by using which, we can traverse every vertex of the graph. Kruskal's algorithm follows greedy approach which finds an optimum solution at every stage instead of focusing on a global optimum.

ALGORITHM:

```
MST-KRUSKAL(G, w)
A ← Ø
for each vertex v V[G]
do MAKE-SET(v)
sort the edges of E into nondecreasing order by weight w
for each edge (u, v) E, taken in nondecreasing order by weight
do if FIND-SET(u) ≠ FIND-SET(v)
then A ← A {(u, v)}
UNION(u, v)
return A
```

Source Code:

`Kruskalsalgorithm.c`

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
int i,j,k,a,b,u,v,n,ne=1;
int min,mincost=0,cost[9][9],parent[9];
int find(int);
int uni(int,int);
void main()
{
        printf("Implementation of Kruskal's algorithm: ");
        printf("\nEnter the no. of vertices:");
        scanf("%d",&n);
        printf("Enter the cost adjacency matrix:");
        for(i=1;i<=n;i++)
        {
                for(j=1;j<=n;j++)
                {
                        scanf("%d",&cost[i][j]);
                        if(cost[i][j]==0)cost[i][j]=999;
                }
        }
        printf("The edges of Minimum Cost Spanning Tree are\n");
        while(ne<n)
        {
                for(i=1,min=999;i<=n;i++)
                {
                for(j=1;j<=n;j++) {
                        if(cost[i][j]<min) {
                                min=cost[i][j];
                                a=u=i;b=v=j;
                        }
                }
                }
                u=find(u);
                v=find(v);
                if(uni(u,v)) {
                        printf("%d edge (%d,%d) =%d\n",ne++,a,b,min);
                        mincost +=min;
                }
                cost[a][b]=cost[b][a]=999;
        }
        printf("Minimum cost = %d\n",mincost);
}
int find(int i) {
        while(parent[i])i=parent[i];
        return i;
}
int uni(int i,int j) {
        if(i!=j) {
                parent[j]=i;
                return 1;
        }
        return 0;
}
```

Execution Results - All test cases have succeeded!

Test Case - 1

User Output

```
Implementation of Kruskal's algorithm:
Enter the no. of vertices:
6
Enter the cost adjacency matrix:
0 3 1 6 0 0
3 0 5 0 3 0
1 5 0 5 6 4
6 0 5 0 0 2
0 3 6 0 0 6
0 0 4 2 6 0
The edges of Minimum Cost Spanning Tree are
1 edge (1,3) =1
2 edge (4,6) =2
3 edge (1,2) =3
4 edge (2,5) =3
5 edge (3,6) =4
Minimum cost = 13
```

S.No: 16	Exp. Name: ***Find Minimum Cost Spanning Tree of a given undirected graph using Prim's algorithm***	**Date: 2023-12-14**

Aim:

OBJECTIVE: Find Minimum Cost Spanning Tree of a given undirected graph using Prim's algorithm

THEORY: Prim's Algorithm is used to find the minimum spanning tree from a graph. Prim's algorithm finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized.
Prim's algorithm starts with the single node and explore all the adjacent nodes with all the connecting edges at every step. The edges with the minimal weights causing no cycles in the graph got selected.

ALGORITHM:

```
MST-PRIM (G, w, r)
1. for each u ∈ V [G]
2. do key [u] ← ∞
3. π [u] ← NIL
4. key [r] ← 0
5. Q ← V [G]
6. While Q ? ∅
7. do u ← EXTRACT - MIN (Q)
8. for each v ∈ Adj [u]
9. do if v ∈ Q and w (u, v) < key [v]
10. then π [v] ← u
11. key [v] ← w (u, v)
```

Source Code:

`PrimsAlgorithm.c`

```
#include<stdio.h>
int a,b,u,v,n,i,j,ne=1;
int visited[10]= { 0 },min,mincost=0,cost[10][10];
void main()
{
        printf("Enter the number of nodes:");
        scanf("%d",&n);
        printf("Enter the adjacency matrix:");
        for (i=1;i<=n;i++)
        for (j=1;j<=n;j++)
        {
                scanf("%d",&cost[i][j]);
                if(cost[i][j]==0)
                cost[i][j]=999;
        }
        visited[1]=1;
        printf("\n");
        while(ne<n)
        {
                for (i=1,min=999;i<=n;i++)
                for (j=1;j<=n;j++)
                if(cost[i][j]<min)
                if(visited[i]!=0)
                {
                        min=cost[i][j];
a=u=i;
b=v=j;
}
if(visited[u]==0 || visited[v]==0) {
        printf("Edge %d:(%d %d) cost:%d\n",ne++,a,b,min);
        mincost+=min;
        visited[b]=1;
}
cost[a][b]=cost[b][a]=999;
}
printf("Minimun cost=%d",mincost);
}
```

Execution Results - All test cases have succeeded!

Test Case - 1

User Output

```
Enter the number of nodes:
6
Enter the adjacency matrix:
0 3 1 6 0 0
3 0 5 0 3 0
1 5 0 5 6 4
6 0 5 0 0 2
0 3 6 0 0 6
0 0 4 2 6 0
```

```
Edge 1:(1 3) cost:1
Edge 2:(1 2) cost:3
Edge 3:(2 5) cost:3
Edge 4:(3 6) cost:4
Edge 5:(6 4) cost:2
Minimun cost=13
```

S.No: 17	Exp. Name: ***Write programs to (a) Implement All-Pairs Shortest Paths problem using Floyd -Warshall algorithm***	**Date: 2023-12-14**

Aim:
Objective: Write programs to (a) Implement All-Pairs Shortest Paths problem using Floyd -Warshall algorithm

THEORY: Floyd-Warshall Algorithm is an algorithm for finding the shortest path between all the pairs of vertices in a weighted graph. This algorithm works for both the directed and undirected weighted graphs. But, it does not work for the graphs with negative cycles (where the sum of the edges in a cycle is negative)

ALGORITHM:
```
FLOYD - WARSHALL (W)
n ← rows [W].
D0 ← W
for k ← 1 to n
do for i ← 1 to n
do for j ← 1 to n
do dij(k) ← min (dij(k-1),dik(k-1)+dkj(k-1) )
return D(n)
```
Source Code:

`WarshallAlgorithm.c`

```
#include<stdio.h>
#include<conio.h>
int min(int,int);
void floyds(int p[10][10],int n)
{
        int i,j,k;
        for (k=1;k<=n;k++)
        for (i=1;i<=n;i++)
        for (j=1;j<=n;j++)
        if(i==j)
        p[i][j]=0;
        else
        p[i][j]=min(p[i][j],p[i][k]+p[k][j]);
}
int min(int a,int b)
{
        if(a<b)
        return(a);
        else
        return(b);
}
void main()
{
        int p[10][10],w,n,e,u,v,i,j;
        printf("Enter the number of vertices:");
        scanf("%d",&n);
        printf("Enter the number of edges: ");
        scanf("%d",&e);
for (i=1;i<=n;i++)
{
        for (j=1;j<=n;j++)
        p[i][j]=999;
}
for (i=1;i<=e;i++)
{
        printf("Enter the end vertices of edge%d with its weight \n",i);
        scanf("%d%d%d",&u,&v,&w);
        p[u][v]=w;
}
printf("Matrix of input data:\n");
for (i=1;i<=n;i++)
{
        for (j=1;j<=n;j++)
        printf("%d \t",p[i][j]);
        printf("\n");
}
floyds(p,n);
printf("Transitive closure:\n");
for (i=1;i<=n;i++)
{
        for (j=1;j<=n;j++)
        printf("%d \t",p[i][j]);
        printf("\n");
}
printf("The shortest paths are:\n");
{
        if(i!=j)
        printf("<%d,%d>=%d",i,j,p[i][j]);
}
}
```

Execution Results - All test cases have succeeded!

Test Case - 1

User Output

```
Enter the number of vertices:
4
Enter the number of edges:
4
Enter the end vertices of edge1 with its weight
1
2
20
Enter the end vertices of edge2 with its weight
2
3
15
Enter the end vertices of edge3 with its weight
1
3
10
Enter the end vertices of edge4 with its weight
4
5
18
Matrix of input data:
999     20      10      999
999     999     15      999
999     999     999     999
999     999     999     999
Transitive closure:
0       20      10      999
999     0       15      999
999     999     0       999
999     999     999     0
The shortest paths are:
<1,2>=20<1,3>=10<1,4>=999<2,1>=999<2,3>=15<2,4>=999<3,1>=999<3,2>=999<3,4>=999<4,1>=999<4,
```

S.No: 18	Exp. Name: ***Design and implement to find a subset of a given set S = {Sl, S2,.......,Sn} of n positive integers whose SUM is equal to a given positive integer d.***	**Date: 2023-12-14**

Aim:

OBJECTIVE: Design and implement to find a subset of a given set S = {Sl, S2,.......,Sn} of n positive integers whose SUM is equal to a given positive integer d.

THEORY: The Subset-Sum Problem is to find a subset's' of the given set S = (S1 S2 S3...Sn) where the elements of the set S are n positive integers in such a manner that s'∈S and sum of the elements of subset's' is equal to some positive integer 'X.'

The Subset-Sum Problem can be solved by using the backtracking approach. In this implicit tree is a binary tree. The root of the tree is selected in such a way that represents that no decision is yet taken on any input.

ALGORITHM:

```
void subset_sum(int list[], int sum, int starting_index, int target_sum)
{
if( target_sum == sum )
{
subset_count++;
if(starting_index < list.length)
subset_sum(list, sum - list[starting_index-1], starting_index, target_sum);
}
else
{
for( int i = starting_index; i < list.length; i++ )
{
subset_sum(list, sum + list[i], i + 1, target_sum);
}
}
}
```

Source Code:

findingSubset.c

```c
#include<stdio.h>
#include<conio.h>
int s[10] , x[10],d ;
void sumofsub ( int , int , int ) ;
void main ()
{
	int n , sum = 0 ;
	int i ;
	printf ( "Enter the size of the set : " ) ;
	scanf ( "%d" , &n ) ;
	printf ( "Enter the set in increasing order: " ) ;
	for ( i = 1 ; i <= n ; i++ )
	scanf ("%d", &s[i] ) ;
	printf ( "Enter the value of d : " ) ;
	scanf ( "%d" , &d ) ;
	for ( i = 1 ; i <= n ; i++ )
	sum = sum + s[i] ;
	if ( sum < d || s[1] > d )
	printf ( "No subset possible : " ) ;
	else sumofsub ( 0 , 1 , sum );
}
void sumofsub ( int m , int k , int r ) {
	int i=1 ;
	x[k] = 1 ;
	if ( ( m + s[k] ) == d ) {
		printf("Subset:");
		for ( i = 1 ; i <= k ; i++ )
		if ( x[i] == 1 )
		printf ( "\t%d" , s[i] ) ;
		printf ( "\n" ) ;
	}
	else if ( m + s[k] + s[k+1] <= d )
	sumofsub ( m + s[k] , k + 1 , r - s[k] ) ;
	if ( ( m + r - s[k] >= d ) && ( m + s[k+1] <=d ) ) {
		x[k] = 0;
		sumofsub ( m , k + 1 , r - s[k] ) ;
	}
}
```

Execution Results - All test cases have succeeded!

Test Case - 1

User Output

```
Enter the size of the set :
6
Enter the set in increasing order:
5 10 12 13 15 18
Enter the value of d :
30
Subset: 5       10       15
Subset: 5       12       13
Subset: 12      18
```

Test Case - 2

User Output

```
Enter the size of the set :
4
Enter the set in increasing order:
12 13 14 15
Enter the value of d :
80
No subset possible :
```

S.No: 19	Exp. Name: ***Design and implement to find all Hamiltonian Cycles in a connected undirected Graph G of n vertices using backtracking principle***	**Date: 2023-12-14**

Aim:

OBJECTIVE: Design and implement to find all Hamiltonian Cycles in a connected undirected Graph G of n vertices using backtracking principle

THEORY: Hamiltonian Path in an undirected graph is a path that visits each vertex exactly once. A Hamiltonian cycle (or Hamiltonian circuit) is a Hamiltonian Path such that there is an edge (in the graph) from the last vertex to the first vertex of the Hamiltonian Path. Determine whether a given graph contains Hamiltonian Cycle or not. If it contains, then prints the path. Following are the input and output of the required function.

ALGORITHM:

```
Begin
if there is no edge between node(k-1) to v, then
return false
if v is already taken, then
return false
return true; //otherwise it is valid
End
```

Source Code:

`Hamiltonian.c`

```c
#include<stdio.h>
#include <stdbool.h>
#define NODE 5
int graph[NODE][NODE];
int path[NODE];
void displayCycle()
{
	printf("The Hamilton Cycle : " );
	for (int i = 0; i < NODE; i++)
	printf("%d ", path[i]);
	printf("%d ", path[0]);
}
bool isValid(int v, int k)
{
	if (graph [path[k-1]][v] == 0)
	return false;
	for (int i = 0; i < k; i++)
	if (path[i] == v)
	return false;
	return true;
}
bool cycleFound(int k)
{
	if (k == NODE)
	{
		if (graph[path[k-1]][ path[0] ] == 1 )
		return true;
		else
	return false; }
	for (int v = 1; v < NODE; v++) {
		if (isValid(v,k)) {
			path[k] = v;
			if (cycleFound (k+1) == true)
			return true;
			path[k] = -1; }}
			return false;
}
bool hamiltonianCycle() {
	for (int i = 0; i < NODE; i++)
	path[i] = -1;
	path[0] = 0;
	if ( cycleFound(1) == false ) {
		printf("Solution does not exist");
		return false; }
		displayCycle();
		return true;
}
int main() {
	int i,j;
	printf("Enter the Graph : " );
	printf("\n");
	for (i=0;i<NODE;i++) {
		for (j=0;j<NODE;j++) {
			printf("Graph G ( %d, %d ): ", (i+1), (j+1));
			scanf("%d", &graph[i][j]);
```

```
                printf("\n");
                for (i=0;i<NODE;i++) {
                        for (j=0;j<NODE;j++) {
                                printf("%d\t ", graph [i][j]); }
                                printf("\n"); }
                                hamiltonianCycle();
}
```

Execution Results - All test cases have succeeded!

Test Case - 1

User Output

```
Enter the Graph :
Graph G ( 1, 1 ):
0
Graph G ( 1, 2 ):
1
Graph G ( 1, 3 ):
0
Graph G ( 1, 4 ):
1
Graph G ( 1, 5 ):
0
Graph G ( 2, 1 ):
1
Graph G ( 2, 2 ):
0
Graph G ( 2, 3 ):
1
Graph G ( 2, 4 ):
1
Graph G ( 2, 5 ):
1
Graph G ( 3, 1 ):
0
Graph G ( 3, 2 ):
1
Graph G ( 3, 3 ):
0
Graph G ( 3, 4 ):
0
Graph G ( 3, 5 ):
1
Graph G ( 4, 1 ):
1
Graph G ( 4, 2 ):
1
```

```
0
Graph G ( 4, 4 ):
0
Graph G ( 4, 5 ):
1
Graph G ( 5, 1 ):
0
Graph G ( 5, 2 ):
1
Graph G ( 5, 3 ):
1
Graph G ( 5, 4 ):
1
Graph G ( 5, 5 ):
0
The Graph :
0       1       0       1       0
1       0       1       1       1
0       1       0       0       1
1       1       0       0       1
0       1       1       1       0
The Hamilton Cycle : 0 1 2 4 3 0
```

Test Case - 2

User Output

```
Enter the Graph :
Graph G ( 1, 1 ):
0
Graph G ( 1, 2 ):
1
Graph G ( 1, 3 ):
0
Graph G ( 1, 4 ):
1
Graph G ( 1, 5 ):
0
Graph G ( 2, 1 ):
1
Graph G ( 2, 2 ):
0
Graph G ( 2, 3 ):
1
Graph G ( 2, 4 ):
1
Graph G ( 2, 5 ):
1
Graph G ( 3, 1 ):
```

```
1
Graph G ( 3, 3 ):
0
Graph G ( 3, 4 ):
0
Graph G ( 3, 5 ):
1
Graph G ( 4, 1 ):
1
Graph G ( 4, 2 ):
1
Graph G ( 4, 3 ):
0
Graph G ( 4, 4 ):
0
Graph G ( 4, 5 ):
0
Graph G ( 5, 1 ):
0
Graph G ( 5, 2 ):
1
Graph G ( 5, 3 ):
1
Graph G ( 5, 4 ):
0
Graph G ( 5, 5 ):
0
The Graph :
0       1       0       1       0
1       0       1       1       1
0       1       0       0       1
1       1       0       0       0
0       1       1       0       0
Solution does not exist
```