

Program-1

Objective - WAP to implement recursive Binary Search

Algorithm -

- Step 1 - Find the element at $\text{arr}[\text{size}/2]$, which will be the array's midpoint. The array is split halfway, with the lower half consisting of items 0 to midpoint -1 and the top half consisting of elements midpoint to size -1.
- Step 2 - Compare the key to $\text{arr}[\text{midpoint}]$ by calling the user function.
- Step 3 - If the key is a match, return $\text{arr}[\text{midpoint}]$;
- Step 4 - Otherwise return NULL, indicating that there is no match if the array has only one element.
- Step 5 - Search the lower half of the array by repeatedly executing search if the key is less than the value taken from $\text{arr}[\text{midpoint}]$.
- Step 6 - Call search recursively to search the upper half of the array.

Code -

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

```
int binarySearch(int arr[], int low, int high, int number)
{
    if (low > high)
    {
        return -1;
    }
    int mid = (low + high)/2;
    if (number == arr[mid])
    {
        return mid;
    }
    else if (number < arr[mid])
    {
        return binarySearch(arr, low, mid - 1, number);
    }
}
```

```

else
{
    return binarySearch(arr, mid + 1, high, number);
}
}

int main(void)
{
    int number;
    int size, i;
    printf("Enter the value of arr size ");
    scanf("%d",&size);
    int arr [size];
    for( i=0;i<size;i++)
    {
        printf("Enter array value\t");
        scanf("%d",&arr[i]);
    }
    printf("Enter the target value: ");
    scanf("%d", &number);

    int n = sizeof(arr)/sizeof(arr[0]);

    int low = 0, high = n - 1;
    int index = binarySearch(arr, low, high, number);
    if (index != -1)
    {
        printf("Element found at index %d", index);
    }
    else
    {
        printf("Element not found in the array");
    }
    return 0;
}

```

Output -

 C:\Users\user\Desktop\binarysearch.exe

Enter size of a list: 5

Enter elements

23

34

45

56

67

Enter key to search

56

Key found at 3

Process exited after 17.08 seconds with return value 14

Press any key to continue . . .

Program 2

Objective - WAP to implement recursive Linear Search

Algorithm -

- Step 1:[set the flag]
flag=0
- Step 2:Repeat through step 3 for k=1,.n
- Step 3:If(list[k] == key)
Flag = 1
Output “Search successful”,element found at location k
- Step 4:If (flag==0)
Output “Search unsuccessful”

Code -

```
#include<stdio.h>

int main()
{
    int n,search,count=0,i;
    printf("Enter the number of elements in array: ");
    scanf("%d",&n);
    printf("%d\n",n);

    int array[n];
    printf("Enter the number in array: ");

    for( i=0;i<n;i++)
    {
        scanf("%d",&array[i]);
    }

    printf("\nEnter the search number: ");
    scanf("%d",&search);

    for( i=0;i<n;i++)
    {
        if(linearSearch(array, n, i, search) == i )
        {
```

```

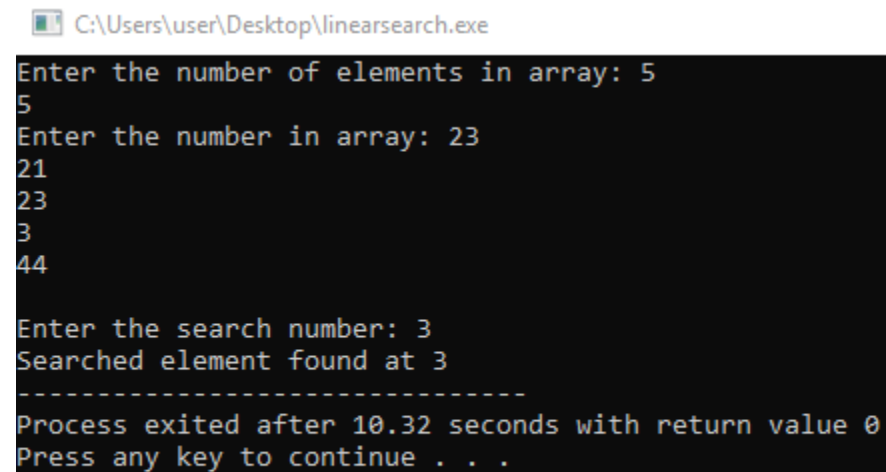
        printf(" The Linear Search for the Index: %d.\n",i);
    }
}

return 0;
}

int linearSearch(int arr[], int size,int x, int s) {
    if(arr[x] ==s)
    {
        return x;
    }
    else
    {
        return -1;
    }
}
}

```

Output -



The screenshot shows a Windows command prompt window titled "C:\Users\user\Desktop\linearssearch.exe". The user has entered the number of elements in the array as 5, and the numbers in the array as 21, 23, 3, and 44. They then entered the search number as 3. The program output indicates that the searched element was found at index 3. The process exited after 10.32 seconds with a return value of 0. The prompt asks to press any key to continue.

```

C:\Users\user\Desktop\linearssearch.exe
Enter the number of elements in array: 5
5
Enter the number in array: 23
21
23
3
44

Enter the search number: 3
Searched element found at 3
-----
Process exited after 10.32 seconds with return value 0
Press any key to continue . . .

```

Program 3

Objective - WAP to implement Selection Sort

Algorithm -

- Step 1: Repeat Steps 2 and 3 for i = 0 to n-1
- Step 2: CALL SMALLEST(arr, i, n, pos)
- Step 3: SWAP arr[i] with arr[pos]
[END OF LOOP]
- Step 4: EXIT

SMALLEST (arr, i, n, pos)

- Step 1: [INITIALIZE] SET SMALL = arr[i]
- Step 2: [INITIALIZE] SET pos = i
- Step 3: Repeat for j = i+1 to n
if (SMALL > arr[j])
SET SMALL = arr[j]
SET pos = j
[END OF if]
[END OF LOOP]
- Step 4: RETURN pos

Code -

```
#include<stdio.h>

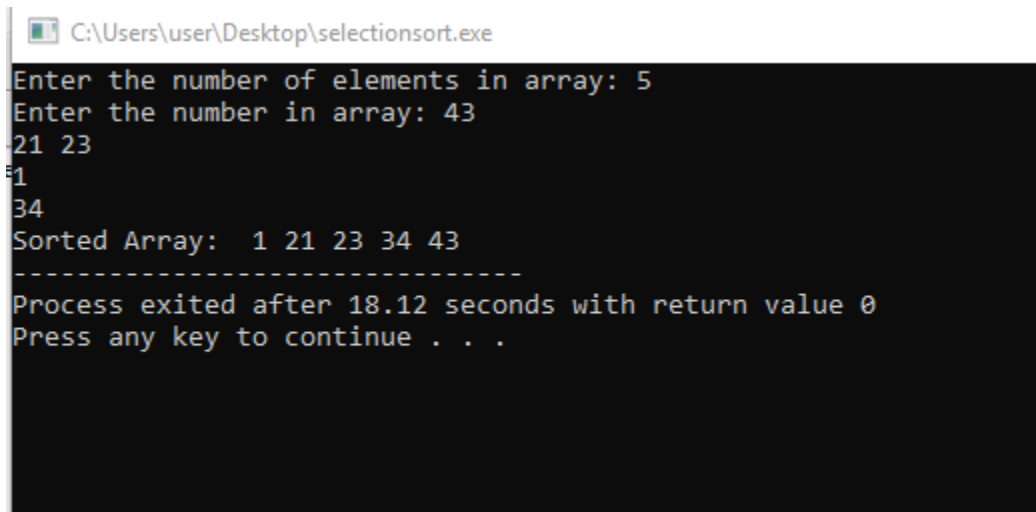
int main() {
    int arr[100], num, x, y, pos, z;
    printf("Enter number of elements to be sorted: \n");
    scanf("%d", &num);
    printf("Enter %d Numbers: \n", num);
    for (x = 0; x < num; x++)
        scanf("%d", &arr[x]);
    for(x = 0; x < num - 1; x++)
    {
        pos=x;
        for(y = x + 1; y < num; y++)
        {
```

```

        if(arr[pos] > arr[y])
            pos=y;
    }
    if(pos != x)
    {
        z=arr[x];
        arr[x]=arr[pos];
        arr[pos]=z;
    }
}
printf("Sorted Array:\n");
for(x = 0; x < num; x++)
    printf(" %d ", arr[x]);
return 0;
}

```

Output -



```

C:\Users\user\Desktop\selectionsort.exe
Enter the number of elements in array: 5
Enter the number in array: 43
21 23
1
34
Sorted Array:  1 21 23 34 43
-----
Process exited after 18.12 seconds with return value 0
Press any key to continue . . .

```

Program 4

Objective - WAP to implement Insertion Sort

Algorithm -

- Step 1 - If the element is the first element, assume that it is already sorted. Return 1.
- Step2 - Pick the next element, and store it separately in a key.
- Step3 - Now, compare the key with all elements in the sorted array.
- Step 4 - If the element in the sorted array is smaller than the current element, then move to the next element. Else, shift greater elements in the array towards the right.
- Step 5 - Insert the value.
- Step 6 - Repeat until the array is sorted.

Code -

```
#include <stdio.h>

int main()
{
    int n, array[1000], c, d, t, flag = 0;

    printf("Enter number of elements\n");
    scanf("%d", &n);

    printf("Enter %d integers\n", n);

    for (c = 0; c < n; c++)
        scanf("%d", &array[c]);

    for (c = 1 ; c <= n - 1; c++) {
        t = array[c];

        for (d = c - 1 ; d >= 0; d--) {
            if (array[d] > t) {
```



```

        array[d+1] = array[d];
        flag = 1;
    }
    else
        break;
}
if (flag)
    array[d+1] = t;
}

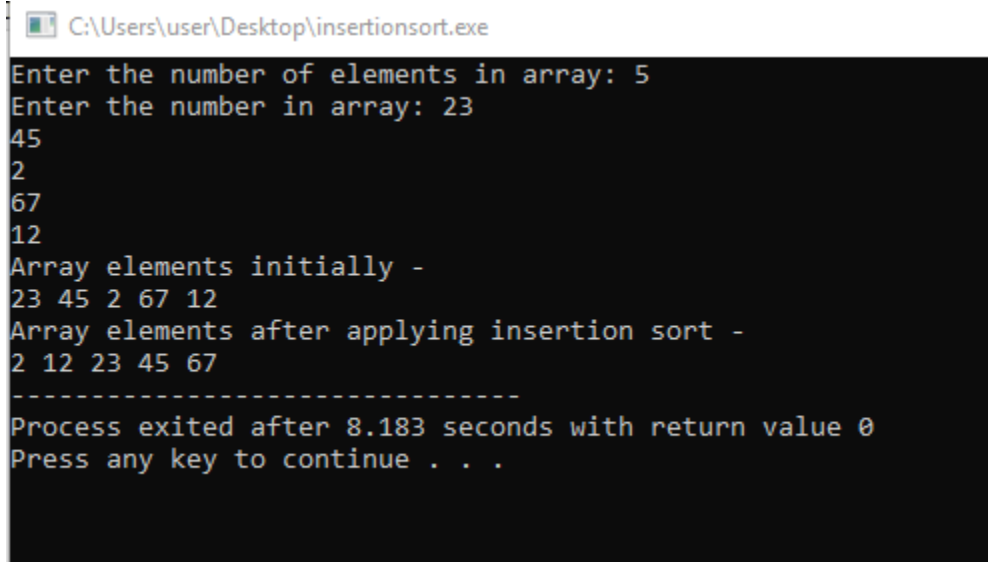
printf("Numbers by Insertion Sort :\n");

for (c = 0; c <= n - 1; c++) {
    printf("%d\n", array[c]);
}

return 0;
}

```

Output -



```

C:\Users\user\Desktop\insertionsort.exe
Enter the number of elements in array: 5
Enter the number in array: 23
45
2
67
12
Array elements initially -
23 45 2 67 12
Array elements after applying insertion sort -
2 12 23 45 67
-----
Process exited after 8.183 seconds with return value 0
Press any key to continue . . .

```

Program-5

Objective - WAP to implement Merge Sort

Algorithm -

- Step 1 - if beg < end
 set mid = (beg + end)/2
- Step 2 - MERGE_SORT(arr, beg, mid)
- Step 3 - MERGE_SORT(arr, mid + 1, end)
- Step 4 - MERGE (arr, beg, mid, end)
- Step 5 - end of if

Code -

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

void merge(int arr[], int l, int m, int r)
{
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;

    int L[n1], R[n2];

    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    i = 0;
    j = 0;
    k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        }
        else {
```

```

        arr[k] = R[j];
        j++;
    }
    k++;
}

while (i < n1) {
    arr[k] = L[i];
    i++;
    k++;
}

while (j < n2) {
    arr[k] = R[j];
    j++;
    k++;
}
}

void mergeSort(int arr[], int l, int r)
{
    if (l < r) {
        int m = l + (r - l) / 2;

        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);
    }
}

void printArray(int A[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        printf("%d ", A[i]);
    printf("\n");
}

```

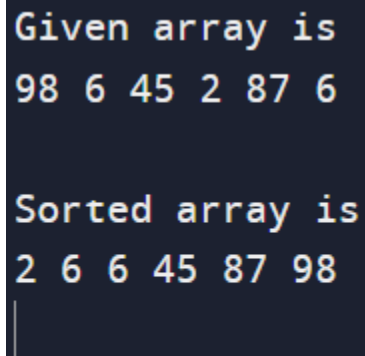
```
int main()
{
    int arr[] = {98,6,45,2,87,6};
    int arr_size = sizeof(arr) / sizeof(arr[0]);

    printf("Given array is \n");
    printArray(arr, arr_size);

    mergeSort(arr, 0, arr_size - 1);

    printf("\nSorted array is \n");
    printArray(arr, arr_size);
    return 0;
}
```

Output -



```
Given array is
98 6 45 2 87 6

Sorted array is
2 6 6 45 87 98
|
```

Program-6

Objective - WAP to implement Quick Sort

Algorithm -

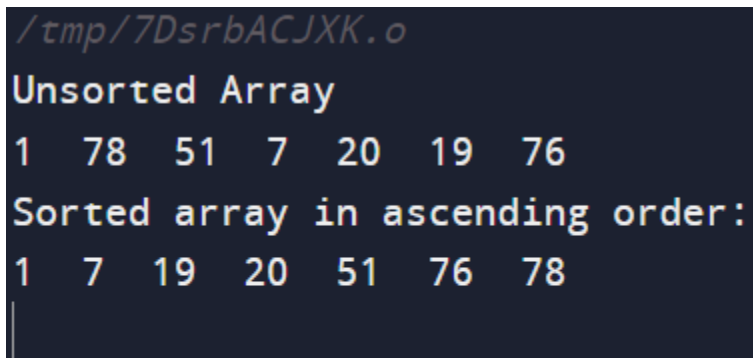
- Step 1 - if (start < end)
- Step 2 - p = partition(A, start, end)
- Step 3 - QUICKSORT (A, start, p - 1)
- Step 4 - QUICKSORT (A, p + 1, end)
- Step 5 - End

Code -

```
#include<stdio.h>
void quicksort(int number[25],int first,int last){
    int i, j, pivot, temp;
    if(first<last){
        pivot=first;
        i=first;
        j=last;
        while(i<j){
            while(number[i]<=number[pivot]&& i<last)
                i++;
            while(number[j]>number[pivot])
                j--;
            if(i<j){
                temp=number[i];
                number[i]=number[j];
                number[j]=temp;
            }
        }
        temp=number[pivot];
        number[pivot]=number[j];
        number[j]=temp;
        quicksort(number,first,j-1);
        quicksort(number,j+1,last);
    }
}
int main(){
```

```
int i, count, number[25];
printf("How many elements are u going to enter?: ");
scanf("%d",&count);
printf("Enter %d elements: ", count);
for(i=0;i<count;i++)
scanf("%d",&number[i]);
quicksort(number,0,count-1);
printf("Order of Sorted elements: ");
for(i=0;i<count;i++)
printf(" %d",number[i]);
return 0;
}
```

Output -



```
/tmp/7DsrBACJXK.o
Unsorted Array
1 78 51 7 20 19 76
Sorted array in ascending order:
1 7 19 20 51 76 78
|
```

Program-7

Objective - WAP to implement Heap Sort

Algorithm -

- Step 1: Build a heap from the input data. Build a max heap to sort in increasing order, and build a min heap to sort in decreasing order.
- Step 2: Swap the root element with the last item of the heap.
- Step 3: Reduce the heap size by 1.
- Step 4: Heapify the remaining elements into a heap of the new heap size by calling heapify on the root node.
- Step 5: Repeat steps 2,3,4 as long as the heap size is greater than 2.

Code -

```
#include <stdio.h>
void swap(int* a, int* b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
void heapify(int arr[], int N, int i)
{
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;
    if (left < N && arr[left] > arr[largest])
        largest = left;
    if (right < N && arr[right] > arr[largest])
        largest = right;
    if (largest != i) {
        swap(&arr[i], &arr[largest]);
        heapify(arr, N, largest);
    }
}
void heapSort(int arr[], int N)
```

```

{
    for (int i = N / 2 - 1; i >= 0; i--)
        heapify(arr, N, i);
    for (int i = N - 1; i >= 0; i--) {
        swap(&arr[0], &arr[i]);
        heapify(arr, i, 0);
    }
}
void printArray(int arr[], int N)
{
    for (int i = 0; i < N; i++)
        printf("%d ", arr[i]);
    printf("\n");
}
int main()
{
    int arr[] = { 62, 75, 322, 855, 4, 77 };
    printf("Unsorted Array is: \n");
    int N = sizeof(arr) / sizeof(arr[0]);
    printArray(arr, N);

    heapSort(arr, N);
    printf("Sorted array is\n");
    printArray(arr, N);
}

```

Output -

```

Output
/tmp/TTVj84ifJd.o
Unsorted Array is:
62 75 322 855 4 77
Sorted array is:
4 62 75 77 322 855

```


Program-8

Objective - WAP to implement Knapsack problem using Greedy Algorithm

Algorithm -

- Step 1: Calculate the ratio (profit/weight) for each item.
- Step 2: Sort all the items in decreasing order of the ratio.
- Step 3: Initialize `knap_profit = 0`, `curr_weight = 0`.
- Step 4: Do the following for every item `i` in the sorted order:
- Step 5: If the weight of the current item is less than or equal to the remaining capacity then add the value of that item into the result
- Step 6: Else add the current item as much as we can and break out of the loop.
- Step 7: Return `knap_profit`.

Code -

```
#include<stdio.h>
int main()
{
    float weight[50],profit[50],ratio[50],temp,capacity,amount,rem=0.0,
curr_weight=0.0,knap_profit=0.0, x[50];
    int n,i,j;
    printf("Enter the number of items :");
    scanf("%d",&n);
    for (i = 0; i < n; i++)
    {
        printf("Enter Weight and Profit for item[%d] :\n",i);
        scanf("%f %f", &weight[i], &profit[i]);
    }
    for(i=0; i<n; i++){
        x[i]=0;
    }
    printf("Enter the capacity of knapsack :\n");
    scanf("%f",&capacity);

    for(i=0;i<n;i++)
        ratio[i]=profit[i]/weight[i];

    for (i = 0; i < n; i++) {
```

```

for (j = i + 1; j < n; j++) {
    if (ratio[i] < ratio[j])
    {
        temp = ratio[j];
        ratio[j] = ratio[i];
        ratio[i] = temp;

        temp = weight[j];
        weight[j] = weight[i];
        weight[i] = temp;

        temp = profit[j];
        profit[j] = profit[i];
        profit[i] = temp;
    }
}

printf("Knapsack problems using Greedy Algorithm:\n");
for (i = 0; i < n; i++)
{
    if(curr_weight>=capacity){
        break;
    }
    if(curr_weight + weight[i] < capacity){
        x[i] = 1;
        curr_weight = curr_weight + weight[i];
        knap_profit =knap_profit + (profit[i] * x[i]);
    }
    else{
        rem = capacity - curr_weight;
        curr_weight = curr_weight+rem;
        x[i]= rem/weight[i];
        knap_profit = knap_profit + (profit[i]*x[i]);
    }
}
for(i=0;i<n;i++){
    printf("%f ",x[i]);
}
printf("\nThe maximum value is :%f\n",knap_profit);

```

```
    return 0;  
}
```

Output -

```
/tmp/609AGZp7BT.o  
Enter the number of items : 7  
7  
Enter Weight and Profit for item[0] :  
2 10  
Enter Weight and Profit for item[1] :  
3 5  
Enter Weight and Profit for item[2] :  
5 15  
Enter Weight and Profit for item[3] :  
7 7  
Enter Weight and Profit for item[4] :  
1 6  
Enter Weight and Profit for item[5] :  
4 18  
Enter Weight and Profit for item[6] :  
1 3  
Enter the capacity of knapsack :  
15  
Knapsack problems using Greedy Algorithm:  
1.000000 1.000000 1.000000 1.000000 1.000000 0.666667 0.000000  
The maximum value is :55.333332
```


Program-9

Objective - Perform Travelling Salesman Problem

Algorithm -

- Traveling salesman problem takes a graph $G \{V, E\}$ as an input and declares another graph as the output (say G') which will record the path the salesman is going to take from one node to another.
- The algorithm begins by sorting all the edges in the input graph G from the least distance to the largest distance.
- The first edge selected is the edge with least distance, and one of the two vertices (say A and B) being the origin node (say A).
- Then among the adjacent edges of the node other than the origin node (B), find the least cost edge and add it onto the output graph.
- Continue the process with further nodes making sure there are no cycles in the output graph and the path reaches back to the origin node A .
- However, if the origin is mentioned in the given problem, then the solution must always start from that node only.

Code -

```
#include <stdio.h>
int tsp_g[10][10] = {
    {3, 13, 3, 13, 45},
    {6, 2, 9, 15, 18},
    {8, 3, 5, 6, 122},
    {2, 2, 16, 10, 5},
    {1, 4, 3, 24, 2}
};
int visited[10], n, cost = 0;
void travellingsalesman(int c){
    int k, adj_vertex = 999;
    int min = 999;
    visited[c] = 1
    printf("%d ", c + 1);

    for(k = 0; k < n; k++) {
        if((tsp_g[c][k] != 0) && (visited[k] == 0)) {
```

```

        if(tsp_g[c][k] < min) {
            min = tsp_g[c][k];
        }
        adj_vertex = k;
    }
}
if(min != 999) {
    cost = cost + min;
}
if(adj_vertex == 999) {
    adj_vertex = 0;
    printf("%d", adj_vertex + 1);
    cost = cost + tsp_g[c][adj_vertex];
    return;
}
travellingsalesman(adj_vertex);
}
int main(){
    int i, j;
    n = 5;
    for(i = 0; i < n; i++) {
        visited[i] = 0;
    }
    printf("Shortest Path: ");
    travellingsalesman(0);
    printf("\nMinimum Cost: ");
    printf("%d\n", cost);
    return 0;
}

```

Output -

```
/tmp/EZMsWyRZ7U.o
```

```
Shortest Path: 1 5 4 3 2 1
```

```
Minimum Cost: 67
```

Program-10

Objective - WAP to find Minimum Spanning Tree using Kruskal's Algorithm

Algorithm -

- Step 1: Sort all edges in increasing order of their edge weights.
- Step 2: Pick the smallest edge.
- Step 3: Check if the new edge creates a cycle or loop in a spanning tree.
- Step 4: If it doesn't form the cycle, then include that edge in MST. Otherwise, discard it.
- Step 5: Repeat from step 2 until it includes $|V| - 1$ edges in MST.

Code -

```
#include <stdio.h>
#include <stdlib.h>
int comparator(const void* p1, const void* p2)
{
    const int(*x)[3] = p1;
    const int(*y)[3] = p2;
    return (*x)[2] - (*y)[2];
}
void makeSet(int parent[], int rank[], int n)
{
    for (int i = 0; i < n; i++) {
        parent[i] = i;
        rank[i] = 0;
    }
}
int findParent(int parent[], int component)
{
    if (parent[component] == component)
        return component;
    return parent[component]
        = findParent(parent, parent[component]);
}
void unionSet(int u, int v, int parent[], int rank[], int n)
{
    u = findParent(parent, u);
```



```

        v = findParent(parent, v);
        if (rank[u] < rank[v]) {
            parent[u] = v;
        }
        else if (rank[u] > rank[v]) {
            parent[v] = u;
        }
        else {
            parent[v] = u;
            rank[u]++;
        }
    }
}

void kruskalAlgo(int n, int edge[n][3])
{
    qsort(edge, n, sizeof(edge[0]), comparator);
    int parent[n];
    int rank[n];
    makeSet(parent, rank, n);
    int minCost = 0;
    printf(
        "Following are the edges in the constructed MST\n");
    for (int i = 0; i < n; i++) {
        int v1 = findParent(parent, edge[i][0]);
        int v2 = findParent(parent, edge[i][1]);
        int wt = edge[i][2];
        if (v1 != v2) {
            unionSet(v1, v2, parent, rank, n);
            minCost += wt;
            printf("%d -- %d == %d\n", edge[i][0],
                edge[i][1], wt);
        }
    }
    printf("Minimum Cost Spanning Tree: %d\n", minCost);
}

int main()
{
    int edge[5][3] = { { 2, 6, 10 },
                        { 1, 0, 6 },
                        { 5, 0, 2 },

```

```
        { 3, 3, 0 },  
        { 0, 3, 7 } };  
    kruskalAlgo(5, edge);  
    return 0;  
}
```

Output -

```
Following are the edges in the constructed MST  
2 -- 3 == 4  
0 -- 3 == 5  
0 -- 1 == 10  
Minimum Cost Spanning Tree: 19  
  
...Program finished with exit code 0  
Press ENTER to exit console.
```

Program-11

Objective - Implement N Queen Problem using Backtracking

Algorithm -

- Start in the leftmost column
- If all queens are placed return true
- Try all rows in the current column. Do the following for every row.
- If the queen can be placed safely in this row
- Then mark this [row, column] as part of the solution and recursively check if placing queen here leads to a solution.
- If placing the queen in [row, column] leads to a solution then return true.
- If placing queen doesn't lead to a solution then unmark this [row, column] then backtrack and try other rows.
- If all rows have been tried and valid solution is not found return false to trigger backtracking.

Code -

```
#define N 4
#include <stdbool.h>
#include <stdio.h>

void printSolution(int board[N][N])
{
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            if(board[i][j])
                printf("Q ");
            else
                printf(". ");
        }
        printf("\n");
    }
}

bool isSafe(int board[N][N], int row, int col)
```

```

{
    int i, j;

    for (i = 0; i < col; i++)
        if (board[row][i])
            return false;

    for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
        if (board[i][j])
            return false;

    for (i = row, j = col; j >= 0 && i < N; i++, j--)
        if (board[i][j])
            return false;

    return true;
}

```

```

bool solveNQUtil(int board[N][N], int col)
{
    if (col >= N)
        return true;

    for (int i = 0; i < N; i++) {

        if (isSafe(board, i, col)) {

            board[i][col] = 1;

            if (solveNQUtil(board, col + 1))
                return true;

            board[i][col] = 0; // BACKTRACK
        }
    }

    return false;
}

```

```

}

bool solveNQ()
{
    int board[N][N] = { { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 } };

    if (solveNQUtil(board, 0) == false) {
        printf("Solution does not exist");
        return false;
    }

    printSolution(board);
    return true;
}

int main()
{
    solveNQ();
    return 0;
}

```

Output -

```

/tmp/EZMsWyRZ7U.o
. . Q .
Q . . .
. . . Q
. Q . .
|

```

Program-14

Objective - WAP to implement 0/1 knapsack problem using Dynamic Programming

Algorithm -

- Step 1: Create a table of values and weights.
- Step 2: Use the formula $V[i,j]=\max\{V[i-1,j],v_i+V[i-1,j-w_i]\}$ to find each entry.
- Step 3: The final value in the bottom right of the table is the maximum profit value of the knapsack.

Code -

```
#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;
}
```

Output -

```
/tmp/EZMsWyRZ7U.o
Enter number of items: 3
3
Enter value and weight of items:
3 6
6 1
6 7
Enter size of knapsack: 20
20
15
```

Program-15

Objective - WAP for a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra's algorithm.

Algorithm -

- Step 1: Create a set sptSet (shortest path tree set) that keeps track of vertices included in the shortest path tree, i.e., whose minimum distance from the source is calculated and finalized. Initially, this set is empty.
- Step 2: Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign the distance value as 0 for the source vertex so that it is picked first.
- Step 3: While sptSet doesn't include all vertices
- Step 4: Pick a vertex u that is not there in sptSet and has a minimum distance value.
- Step 5: Include u to sptSet.
- Step 6: Then update the distance value of all adjacent vertices of u.
- Step 7: To update the distance values, iterate through all adjacent vertices.
- Step 8: For every adjacent vertex v, if the sum of the distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

Code -

```
#include <limits.h>
#include <stdbool.h>
#include <stdio.h>
#define V 9
int minDistance(int dist[], bool sptSet[])
{
    int min = INT_MAX, min_index;
    for (int v = 0; v < V; v++)
        if (sptSet[v] == false && dist[v] <= min)
            min = dist[v], min_index = v;
    return min_index;
}
void printSolution(int dist[])
{
    }
```



```

    printf("Vertex \t\t Distance from Source\n");
    for (int i = 0; i < V; i++)
        printf("%d \t\t\t %d\n", i, dist[i]);
}
void dijkstra(int graph[V][V], int src)
{
    int dist[V];
    bool sptSet[V];
    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX, sptSet[i] = false;
    dist[src] = 0;
    for (int count = 0; count < V - 1; count++) {
        int u = minDistance(dist, sptSet);
        sptSet[u] = true;
        for (int v = 0; v < V; v++)
            if (!sptSet[v] && graph[u][v]
                && dist[u] != INT_MAX
                && dist[u] + graph[u][v] < dist[v])
                dist[v] = dist[u] + graph[u][v];
    }
    printSolution(dist);
}
int main()
{
    int graph[V][V] = { { 4, 4, 1, 1, 1, 1, 1, 5, 1 },
                        { 8, 5, 8, 1, 1, 1, 1, 11, 1 },
                        { 0, 8, 0, 7, 0, 4, 0, 7, 2 },
                        { 0, 2, 7, 0, 9, 14, 0, 4, 0 },
                        { 5, 0, 0, 9, 0, 10, 0, 1, 0 },
                        { 7, 4, 4, 4, 0, 0, 2, 0, 9 },
                        { 9, 6, 0, 0, 1, 2, 0, 1, 6 },
                        { 6, 11, 0, 0, 0, 0, 1, 0, 7 },
                        { 1, 0, 2, 0, 0, 0, 6, 7, 0 } };

    dijkstra(graph, 0);
    return 0;
}

```

Output -

```
/tmp/J9tpxs6yW.o  
Vertex      Distance from Source  
0           0  
1           9  
2           2  
3           9  
4          18  
5           6  
6           8  
7           8  
8           7  
|
```

Program-16

Objective - WAP to find Minimum Spanning Tree using Kruskal's Algorithm

Algorithm -

- Step 1: Sort all edges in increasing order of their edge weights.
- Step 2: Pick the smallest edge.
- Step 3: Check if the new edge creates a cycle or loop in a spanning tree.
- Step 4: If it doesn't form the cycle, then include that edge in MST. Otherwise, discard it.
- Step 5: Repeat from step 2 until it includes $|V| - 1$ edges in MST.

Code -

```
#include <stdio.h>
#include <stdlib.h>
int comparator(const void* p1, const void* p2)
{
    const int(*x)[3] = p1;
    const int(*y)[3] = p2;
    return (*x)[2] - (*y)[2];
}
void makeSet(int parent[], int rank[], int n)
{
    for (int i = 0; i < n; i++) {
        parent[i] = i;
        rank[i] = 0;
    }
}
int findParent(int parent[], int component)
{
    if (parent[component] == component)
        return component;
    return parent[component]
        = findParent(parent, parent[component]);
}
void unionSet(int u, int v, int parent[], int rank[], int n)
{
    u = findParent(parent, u);
```

```

        v = findParent(parent, v);
        if (rank[u] < rank[v]) {
            parent[u] = v;
        }
        else if (rank[u] > rank[v]) {
            parent[v] = u;
        }
        else {
            parent[v] = u;
            rank[u]++;
        }
    }
}

void kruskalAlgo(int n, int edge[n][3])
{
    qsort(edge, n, sizeof(edge[0]), comparator);
    int parent[n];
    int rank[n];
    makeSet(parent, rank, n);
    int minCost = 0;
    printf(
        "Following are the edges in the constructed MST\n");
    for (int i = 0; i < n; i++) {
        int v1 = findParent(parent, edge[i][0]);
        int v2 = findParent(parent, edge[i][1]);
        int wt = edge[i][2];
        if (v1 != v2) {
            unionSet(v1, v2, parent, rank, n);
            minCost += wt;
            printf("%d -- %d == %d\n", edge[i][0],
                edge[i][1], wt);
        }
    }
    printf("Minimum Cost Spanning Tree: %d\n", minCost);
}

int main()
{
    int edge[5][3] = { { 3, 10, 1 },
                        { 9, 0, 6 },

```

```
        { 5, 5, 5 },  
        { 11, 13, 10 },  
        { 8, 13, 8 } };  
  
    kruskalAlgo(5, edge);  
    return 0;  
}
```

Output -

```
Following are the edges in the constructed MST  
2 -- 3 == 4  
0 -- 3 == 5  
0 -- 1 == 10  
Minimum Cost Spanning Tree: 19  
  
...Program finished with exit code 0  
Press ENTER to exit console.
```

Program-17

Objective - Find Minimum Cost Spanning Tree of a given undirected graph using Prim's algorithm.

Algorithm -

- Step 1: Determine an arbitrary vertex as the starting vertex of the MST.
- Step 2: Follow steps 3 to 5 till there are vertices that are not included in the MST (known as fringe vertex).
- Step 3: Find edges connecting any tree vertex with the fringe vertices.
- Step 4: Find the minimum among these edges.
- Step 5: Add the chosen edge to the MST if it does not form any cycle.
- Step 6: Return the MST and exit.

Code -

```
#include <limits.h>
#include <stdbool.h>
#include <stdio.h>
#define V 5
int minKey(int key[], bool mstSet[])
{
    int min = INT_MAX, min_index;
    for (int v = 0; v < V; v++)
        if (mstSet[v] == false && key[v] < min)
            min = key[v], min_index = v;
    return min_index;
}
int printMST(int parent[], int graph[V][V])
{
    printf("Edge \tWeight\n");
    for (int i = 1; i < V; i++)
        printf("%d - %d \t%d \n", parent[i], i,
            graph[i][parent[i]]);
}
void primMST(int graph[V][V])
{
    int key[V];
```

```

bool mstSet[V];
for (int i = 0; i < V; i++)
    key[i] = INT_MAX, mstSet[i] = false;
key[0] = 0;
parent[0] = -1;
for (int count = 0; count < V - 1; count++) {
    int u = minKey(key, mstSet);
    mstSet[u] = true;
    for (int v = 0; v < V; v++)
    {
        if (graph[u][v] && mstSet[v] == false
            && graph[u][v] < key[v])
            parent[v] = u, key[v] = graph[u][v];
    }
    printMST(parent, graph);
}
int main()
{
    int graph[V][V] = {8, 1, 3, 6, 8},
    {5, 10, 5, 7, 2},
    {2, 3, 10, 11, 16},
    {2, 9, 5, 0, 1},
    {0, 2, 6, 31, 0}};
    primMST(graph);
    return 0;
}

```

Output -

```
/tmp/J9tpxs6yW.o
```

| Edge | Weight |
|-------|--------|
| 0 - 1 | 7 |
| 1 - 2 | 1 |
| 1 - 3 | 1 |
| 2 - 4 | 9 |

Program-18

Objective - WAP to implement All-Pairs Shortest Paths problem using Floyd's algorithm.

Algorithm -

- Step 1: Initialize the solution matrix same as the input graph matrix as a first step.
- Step 2: Then update the solution matrix by considering all vertices as an intermediate vertex.
- Step 3: The idea is to pick all vertices one by one and updates all shortest paths which include the picked vertex as an intermediate vertex in the shortest path.
- Step 4: When we pick vertex number k as an intermediate vertex, we already have considered vertices {0, 1, 2, .. k-1} as intermediate vertices.
- Step 5: For every pair (i, j) of the source and destination vertices respectively, there are two possible cases.
- Step 6: k is not an intermediate vertex in shortest path from i to j. We keep the value of $\text{dist}[i][j]$ as it is.
- Step 7: k is an intermediate vertex in shortest path from i to j. We update the value of $\text{dist}[i][j]$ as $\text{dist}[i][k] + \text{dist}[k][j]$, if $\text{dist}[i][j] > \text{dist}[i][k] + \text{dist}[k][j]$

Code -

```
#include <stdio.h>
#define V 4
#define INF 99999
void printSolution(int dist[][V]);
void floydWarshall(int dist[][V])
{
    int i, j, k;
    for (k = 0; k < V; k++) {
        for (i = 0; i < V; i++) {
            for (j = 0; j < V; j++) {
                if (dist[i][k] + dist[k][j] < dist[i][j])
                    dist[i][j] = dist[i][k] + dist[k][j];
            }
        }
    }
}
```



```

    }
    printSolution(dist);
}
void printSolution(int dist[][V])
{
    printf(
        "The following matrix shows the shortest distances"
        " between every pair of vertices \n");
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            if (dist[i][j] == INF)
                printf("%7s", "INF");
            else
                printf("%7d", dist[i][j]);
        }
        printf("\n");
    }
}
int main()
{
    int graph[V][V] = { { INF, 0, INF, 0 },
                        { 0, 0, 3, INF },
                        { 0, INF, 0, INF },
                        { 0, INF, 1, 0 } };

    floydWarshall(graph);
    return 0;
}

```

Output -

/tmp/J9tpxs6yW.o

The following matrix shows the shortest

| | | | |
|-----|-----|-----|---|
| 5 | 5 | 8 | 9 |
| 0 | 0 | 3 | 4 |
| INF | INF | 0 | 1 |
| INF | INF | INF | 0 |

Program-19

Objective - WAP to find a subset of a given set $S = \{S_1, S_2, \dots, S_n\}$ of n positive integers whose SUM is equal to a given positive integer d

Algorithm -

- Step 1: Initialize a 2D array of size $(n+1) \times (d+1)$ with all elements set to false.
- Step 2: Set $dp[0][i] = \text{true}$ for all $i = 0$ to n . An empty subset has a sum of 0.
- Step 3: for i from 1 to n :
 - for j from 1 to d :
 - $dp[i][j] = dp[i-1][j]$ or $(j \geq S[i] \text{ and } dp[i-1][j - S[i]])$
- Step 4: if $dp[n][d]$ is false:
- return "No subset found."

Code -

```
#include<stdio.h>
#include<conio.h>
int s[10],d,n,set[10],count=0;
void display(int);
int flag=0;
void main()
{
    int subset(int,int);
    int i;
    clrscr();
    printf("Enter the number of elements in set\n");
    scanf("%d",&n);
    printf("Enter the set values\n");
    for(i=0;i<n;++i)
        scanf("%d",&s[i]);
    printf("Enter the sum\n");
    scanf("%d",&d);
    printf("The program output is\n");
    subset(0,0);
    if(flag==0)
        printf("there is no solution");
```

```

        getch();
    }
    int subset(int sum,int i)
    {
        if(sum==d)
        {
            flag=1;
            display(count);
            return;
        }
        if(sum>d||i>=n)
            return;
        else
        {
            set[count]=s[i];
            count++;
            subset(sum+s[i],i+1);
            count--;
            subset(sum,i+1);
        }
    }
}

void display(int count)
{
    int i;
    printf("{");
    for(i=0;i<count;i++)
        printf("%d",set[i]);
    printf("}");
}

```

Output -

```
enter the number of elements of set
5
enter the elements of set
5
10
15
20
15
enter the positive integer sum15
Subset={15,}=15Subset={15,}=15Subset={5,10,}=15|
```

Program-20

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

Algorithm -

- Step 1: Initialize an empty list to store Hamiltonian Cycles: hamiltonianCycles
- Step 2: Create an array to keep track of visited vertices: visitedVertices[1..n], initially all set to false.
- Step 3: For each vertex v in G:
 - a. Mark v as visited (visitedVertices[v] = true).
 - b. Call the recursive function HamiltonianCycleUtil with the current path [v], hamiltonianCycles, visitedVertices.
 - c. Mark v as unvisited (visitedVertices[v] = false) to explore other possibilities.
- Step 4: Return the list of Hamiltonian Cycles: hamiltonianCycles.

Code -

```
#include<stdio.h>
#define V 5
void printSolution(int path[]);
bool isSafe(int v, bool graph[V][V], int path[], int pos)
{
    if (graph [ path[pos-1] ][ v ] == 0)
        return false;
    for (int i = 0; i < pos; i++)
        if (path[i] == v)
            return false;
    return true;
}
bool hamCycleUtil(bool graph[V][V], int path[], int pos)
{
    if (pos == V)
    {
        if ( graph[ path[pos-1] ][ path[0] ] == 1 )
            return true;
        else
```

```

        return false;
    }
    for (int v = 1; v < V; v++)
    {
        if (isSafe(v, graph, path, pos))
        {
            path[pos] = v;
            if (hamCycleUtil (graph, path, pos+1) == true)
                return true;
            // then remove it */
            path[pos] = -1;
        }
    }
    return false;
}

bool hamCycle(bool graph[V][V])
{
    int *path = new int[V];
    for (int i = 0; i < V; i++)
        path[i] = -1;
    path[0] = 0;
    if ( hamCycleUtil(graph, path, 1) == false )
    {
        printf("\nSolution does not exist");
        return false;
    }
    printSolution(path);
    return true;
}

void printSolution(int path[])
{
    printf ("Solution Exists:"
           " Following is one Hamiltonian Cycle \n");
    for (int i = 0; i < V; i++)
        printf(" %d ", path[i]);
    printf(" %d ", path[0]);
    printf("\n");
}

int main()
{

```

```

bool graph1[V][V] = {{1, 1, 0, 1, 0},
                      {0, 0, 1, 1, 1},
                      {1, 1, 0, 1, 1},
                      {0, 1, 0, 0, 1},
                      {0, 1, 0, 1, 1},
                      };
hamCycle(graph1);

bool graph2[V][V] = {{0, 1, 0, 1, 0},
                      {1, 1, 1, 1, 1},
                      {0, 1, 1, 0, 1},
                      {1, 0, 1, 0, 0},
                      {0, 1, 0, 0, 0},
                      };
hamCycle(graph2);
return 0;
}

```

Output -

```

/tmp/Ms85qx2HUy.o
0 1 2 3 4 5 0
0 1 5 4 3 2 0
0 2 1 5 4 3 0
0 2 3 4 1 5 0
0 2 3 4 5 1 0
0 5 1 2 3 4 0
0 5 1 4 3 2 0
0 5 4 1 2 3 0
0 5 4 3 2 1 0
|

```