**VISVESVARAYA TECHNOLOGICAL UNIVERSITY**

**“JnanaSangama”, Belgaum -590014, Karnataka.**

****

**LAB REPORT**

**On**

**ANALYSIS AND DESIGN OF ALGORITHMS (23CS4PCADA)**

**Submitted by**

**Shreya Jaganatha Gowda**

**1BM23CS352**

**in partial fulfillment for the award of the degree of**

**BACHELOR OF ENGINEERING**

**in**

**COMPUTER SCIENCE AND ENGINEERING**



**B.M.S. COLLEGE OF ENGINEERING**

**(Autonomous Institution under VTU)**

**BENGALURU-560019**

**February-May 2025**

**B. M. S. College of Engineering,**

**Bull Temple Road, Bangalore 560019**

**(Affiliated To Visvesvaraya Technological University, Belgaum)**

**Department of Computer Science and Engineering**

****

This is to certify that the Lab work entitled **“ANALYSIS AND DESIGN OF ALGORITHMS”** carried out by **Shreya Jaganatha Gowda 1BM23CS352** who is bonafide student of **B. M. S. College of Engineering**. It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum during the year 2024-25. The Lab report has been approved as it satisfies the academic requirements in respect of Analysis and Design of Algorithms Lab - **(23CS4PCADA)** work prescribed for the said degree.

**Prof. Namratha M** **Dr. Kavitha Sooda**

Assistant Professor Professor and Head

Department of CSE Department of CSE

BMSCE, Bengaluru BMSCE, Bengaluru

**Index Sheet**

|  |  |  |
| --- | --- | --- |
| **Sl.**  **No.** | **Experiment Title** | **Page No.** |
| 1 | Write program to obtain the Topological ordering of vertices in a given digraph.  LeetCode Program related to Topological sorting | 5-10 |
| 2 | Implement Johnson Trotter algorithm to generate permutations. | 11-14 |
| 3 | Sort a given set of N integer elements using Merge Sort technique and compute its time taken. Run the program for different values of N and record the time taken to sort.  LeetCode Program related to sorting. | 15-20 |
| 4 | Sort a given set of N integer elements using Quick Sort technique and compute its time taken.  LeetCode Program related to sorting. | 21-23 |
| 5 | Sort a given set of N integer elements using Heap Sort technique and compute its time taken. | 24-26 |
| 6 | Implement 0/1 Knapsack problem using dynamic programming.  LeetCode Program related to Knapsack problem or Dynamic Programming. | 27-31 |
| 7 | Implement All Pair Shortest paths problem using Floyd’s algorithm.  LeetCode Program related to shortest distance calculation. | 32-36 |
| 8 | Find Minimum Cost Spanning Tree of a given undirected graph using Prim’s algorithm.  Find Minimum Cost Spanning Tree of a given undirected graph using Kruskal’s algorithm. | 37-43 |
| 9 | Implement Fractional Knapsack using Greedy technique.  LeetCode Program related to Greedy Technique algorithms. | 44-47 |
| 10 | From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra’s algorithm. | 48-50 |
| 11 | Implement “N-Queens Problem” using Backtracking. | 51-53 |

**Course outcomes:**

|  |  |
| --- | --- |
| CO1 | Analyze time complexity of Recursive and Non-recursive algorithms using asymptotic notations. |
| CO2 | Apply various design techniques for the given problem. |
| CO3 | Apply the knowledge of complexity classes P, NP, and NP-Complete and prove certain problems are NP-Complete |
| CO4 | Design efficient algorithms and conduct practical experiments to solve problems. |

**Lab program 1:**

**Program full details**

Write program to obtain the Topological ordering of vertices in a given digraph.

**Code**

#include<stdio.h>

#include<stdlib.h>

int\* topologicalsort(int\*\* Matrix,int matrixSize)

{

    int\* indegree=(int\*)calloc(matrixSize,sizeof(int));

    for(int i=0;i<matrixSize;i++)

    {

        for(int j=0;j<matrixSize;j++)

        {

            if(Matrix[i][j]==1)

            {

                indegree[j]++;

            }

        }

    }

    int\* queue=(int\*)malloc(matrixSize\*sizeof(int));

    int front=0,rear=0;

    for(int i=0;i<matrixSize;i++)

    {

        if(indegree[i]==0)

        {

            queue[rear++]=i;

        }

    }

    int k=0;

    int\* result=(int\*)malloc(matrixSize\*sizeof(int));

    while(front<rear)

    {

        int curr=queue[front++];

        result[k++]=curr;

        for(int i=0;i<matrixSize;i++)

        {

            if(Matrix[curr][i]==1)

            {

                indegree[i]--;

                if(indegree[i]==0)

                {

                    queue[rear++]=i;

                }

            }

        }

    }

    free(queue);

    free(indegree);

    return result;

}

int main()

{

    printf("enter the number of vertices:\n");

    int n;

    scanf("%d",&n);

    int\*\* Matrix=(int\*\*)malloc(n\*sizeof(int));

    for(int i=0;i<n;i++)

    {

        Matrix[i]=(int\*)malloc(n\*sizeof(int));

    }

    printf("enter the adjacency matrix\n");

    for(int i=0;i<n;i++)

    {

        for(int j=0;j<n;j++)

        {

            scanf("%d",&Matrix[i][j]);

        }

    }

    int\* result=(int\*)malloc(n\*sizeof(int));

    result=topologicalsort(Matrix,n);

    printf("Sorted:\n");

    for(int i=0;i<n;i++)

    {

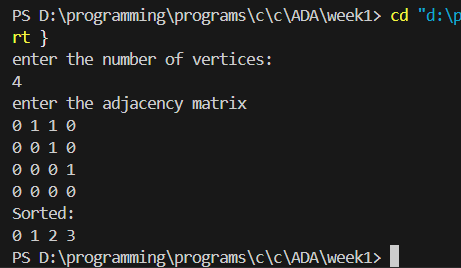
        printf("%d ",result[i]);

    }

    return 0;

}

**Screenshot of Output**



LeetCode Program related to Topological sorting

**Code**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

bool canFinish(int numCourses, int\*\* prerequisites, int prerequisitesSize, int\* prerequisitesColSize) {

// Step 1: Build graph and in-degree array

int\* inDegree = (int\*)calloc(numCourses, sizeof(int));

int\*\* adj = (int\*\*)malloc(numCourses \* sizeof(int\*));

int\* adjSize = (int\*)calloc(numCourses, sizeof(int));

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

adj[i] = (int\*)malloc(numCourses \* sizeof(int)); // Max possible neighbors

}

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

int a = prerequisites[i][0];

int b = prerequisites[i][1];

adj[b][adjSize[b]++] = a; // b → a

inDegree[a]++;

}

// Step 2: BFS queue for Kahn’s Algorithm

int\* queue = (int\*)malloc(numCourses \* sizeof(int));

int front = 0, rear = 0;

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

if (inDegree[i] == 0)

queue[rear++] = i;

}

int count = 0;

while (front < rear) {

int course = queue[front++];

count++;

for (int i = 0; i < adjSize[course]; i++) {

int neighbor = adj[course][i];

inDegree[neighbor]--;

if (inDegree[neighbor] == 0) {

queue[rear++] = neighbor;

}

}

}

// Cleanup

for (int i = 0; i < numCourses; i++)

free(adj[i]);

free(adj);

free(adjSize);

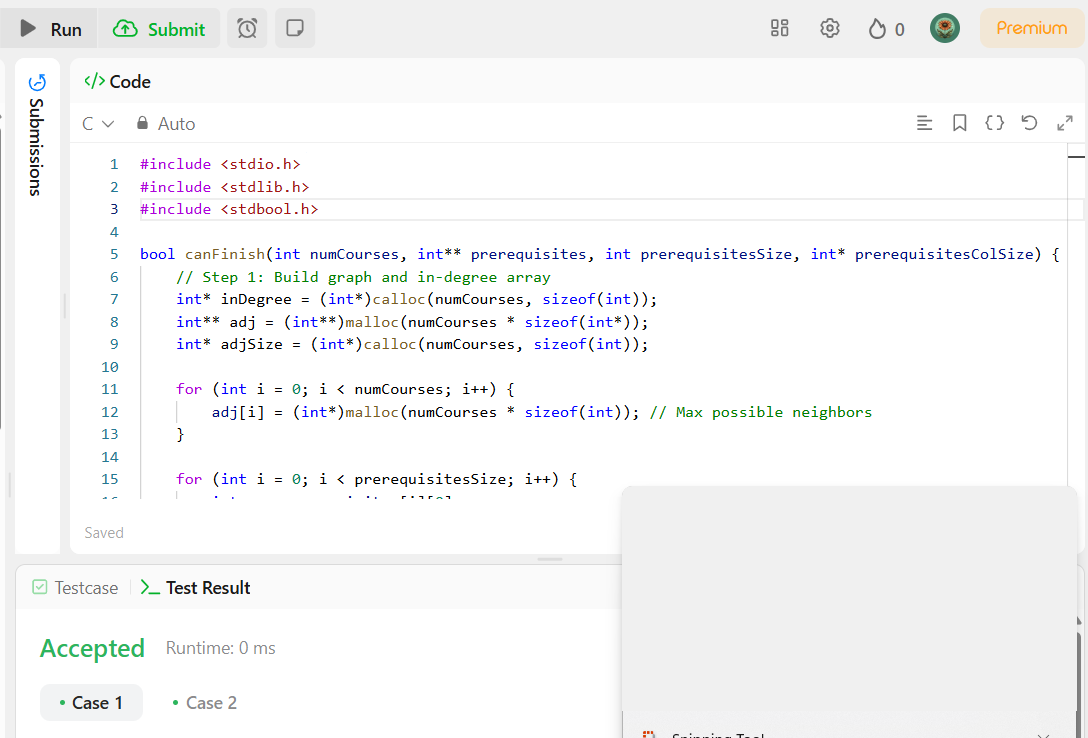
free(queue);

free(inDegree);

return count == numCourses;

}

**Screenshot of Output**

****

**Program full details**

Implement Johnson Trotter algorithm to generate permutations.

**Code**

#include <stdio.h>

#include <stdlib.h>

#define LEFT -1

#define RIGHT 1

// Structure to represent a mobile element

typedef struct {

int value;

int direction; // -1 for LEFT, +1 for RIGHT

} Element;

// Find the largest mobile element index

int largestMobile(Element arr[], int n) {

int maxIndex = -1;

int maxValue = -1;

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

int dir = arr[i].direction;

int adj = i + dir;

if (adj >= 0 && adj < n) {

if (arr[i].value > arr[adj].value && arr[i].value > maxValue) {

maxValue = arr[i].value;

maxIndex = i;

}

}

}

return maxIndex;

}

// Swap two elements

void swap(Element \*a, Element \*b) {

Element temp = \*a;

\*a = \*b;

\*b = temp;

}

// Print current permutation

void printPermutation(Element arr[], int n) {

for (int i = 0; i < n; i++)

printf("%d ", arr[i].value);

printf("\n");

}

void johnsonTrotter(int n) {

// Initialize elements and directions (all to LEFT initially)

Element arr[n];

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

arr[i].value = i + 1;

arr[i].direction = LEFT;

}

printPermutation(arr, n);

while (1) {

int mobileIndex = largestMobile(arr, n);

if (mobileIndex == -1) break; // No mobile element means all permutations generated

int dir = arr[mobileIndex].direction;

int adj = mobileIndex + dir;

// Swap with adjacent element in its direction

swap(&arr[mobileIndex], &arr[adj]);

// Update the index after swap

mobileIndex = adj;

// Reverse direction of all elements greater than moved element's value

int movedValue = arr[mobileIndex].value;

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

if (arr[i].value > movedValue) {

arr[i].direction = -arr[i].direction;

}

}

printPermutation(arr, n);

}

}

int main() {

int n;

printf("Enter number of elements: ");

scanf("%d", &n);

if (n <= 0) {

printf("Invalid input.\n");

return 1;

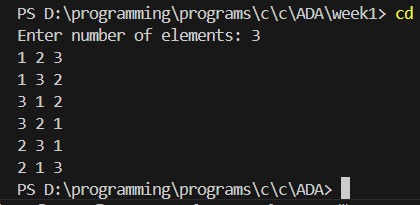
}

johnsonTrotter(n);

return 0;

}

**Screenshot of Output**

****

**Program full details**

Sort a given set of N integer elements using Merge Sort technique and compute its time taken. Run the program for different values of N and record the time taken to sort.

**Code**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

void merge(int a[], int l, int m, int r)

{

int n1 = m - l + 1;

int n2 = r - m;

int\* left = (int\*)malloc(n1 \* sizeof(int));

int\* right = (int\*)malloc(n2 \* sizeof(int));

for (int i = 0; i < n1; i++)

{

left[i] = a[l + i];

}

for (int j = 0; j < n2; j++)

{

right[j] = a[m + 1 + j];

}

int i = 0, j = 0, k = l;

while (i < n1 && j < n2)

{

if (left[i] <= right[j])

{

a[k++] = left[i++];

}

else

{

a[k++] = right[j++];

}

}

while (i < n1)

{

a[k++] = left[i++];

}

while (j < n2)

{

a[k++] = right[j++];

}

free(left);

free(right);

}

void mergesort(int a[], int l, int r)

{

if (l < r)

{

int m = l + (r - l) / 2;

mergesort(a, l, m);

mergesort(a, m + 1, r);

merge(a, l, m, r);

}

}

int main()

{

int a[] = {5, 2, 9, 3, 7};

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

clock\_t start, end;

double cpu\_time\_used;

start = clock();

mergesort(a, 0, n - 1);

end = clock();

cpu\_time\_used = ((double)(end - start)) / CLOCKS\_PER\_SEC;

printf("Sorted Array:\n");

for (int i = 0; i < n; i++)

{

printf("%d ", a[i]);

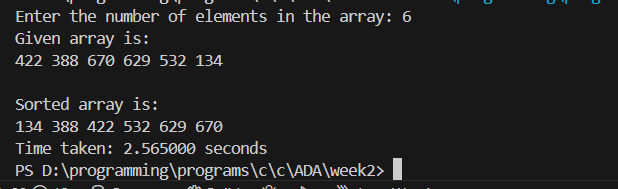
}

printf("\nTime taken: %f seconds\n", cpu\_time\_used);

return 0;

}

**Screenshot of Output**



LeetCode Program related to sorting.

**Code**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

void merge(int\* nums, int l, int m, int r) {

int n1 = m - l + 1;

int n2 = r - m;

int\* left = (int\*)malloc(n1 \* sizeof(int));

int\* right = (int\*)malloc(n2 \* sizeof(int));

for (int i = 0; i < n1; i++)

left[i] = nums[l + i];

for (int j = 0; j < n2; j++)

right[j] = nums[m + 1 + j];

int i = 0, j = 0, k = l;

while (i < n1 && j < n2) {

if (left[i] <= right[j])

nums[k++] = left[i++];

else

nums[k++] = right[j++];

}

while (i < n1)

nums[k++] = left[i++];

while (j < n2)

nums[k++] = right[j++];

free(left);

free(right);

}

void mergesort(int\* nums, int l, int r) {

if (l < r) {

int m = l + (r - l) / 2;

mergesort(nums, l, m);

mergesort(nums, m + 1, r);

merge(nums, l, m, r);

}

}

int\*\* threeSum(int\* nums, int numsSize, int\* returnSize, int\*\* returnColumnSizes) {

mergesort(nums, 0, numsSize - 1);

int\*\* result = (int\*\*)malloc(numsSize \* sizeof(int\*)); // allocate for max 1000 triplets

\*returnColumnSizes = (int\*)malloc(numsSize \* sizeof(int));

\*returnSize = 0;

for (int i = 0; i < numsSize - 2; i++) {

if (i > 0 && nums[i] == nums[i - 1]) continue; // skip duplicates

int left = i + 1;

int right = numsSize - 1;

while (left < right) {

int sum = nums[i] + nums[left] + nums[right];

if (sum == 0) {

result[\*returnSize] = (int\*)malloc(3 \* sizeof(int));

result[\*returnSize][0] = nums[i];

result[\*returnSize][1] = nums[left];

result[\*returnSize][2] = nums[right];

(\*returnColumnSizes)[\*returnSize] = 3;

(\*returnSize)++;

// Skip duplicates

while (left < right && nums[left] == nums[left + 1]) left++;

while (left < right && nums[right] == nums[right - 1]) right--;

left++;

right--;

} else if (sum < 0)

left++;

else

right--;

}

}

return result;

}

**Screenshot of Output:**

****

**Program full details**

Sort a given set of N integer elements using Quick Sort technique and compute its time taken.

**Code**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

void swap(int \*a, int \*b)

{

int t = \*a;

\*a = \*b;

\*b = t;

}

int partition(int a[], int l, int r)

{

int i = l - 1;

int p = a[r];

for (int j = l; j < r; j++)

{

if (a[j] < p)

{

i++;

swap(&a[i], &a[j]);

}

}

swap(&a[i + 1], &a[r]);

return i + 1;

}

void quickSort(int a[], int l, int r)

{

if (l < r)

{

int p = partition(a, l, r);

quickSort(a, l, p - 1);

quickSort(a, p + 1, r);

}

}

int main()

{

int n;

printf("Enter the number of elements in the array:\n");

scanf("%d", &n);

int a[n];

printf("Enter array elements:\n");

for (int i = 0; i < n; i++)

{

scanf("%d", &a[i]);

}

clock\_t start, end;

double cpu\_time\_used;

start = clock();

quickSort(a, 0, n - 1);

usleep(2000);

end = clock();

cpu\_time\_used = ((double)(end - start)) / CLOCKS\_PER\_SEC;

printf("Sorted Array:\n");

for (int i = 0; i < n; i++)

{

printf("%d ", a[i]);

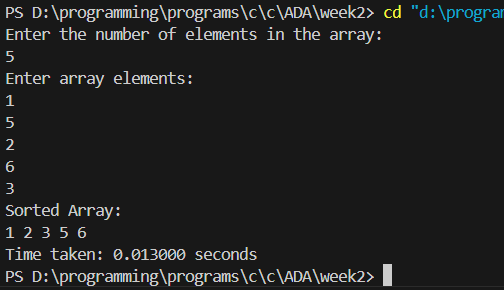
}

printf("\nTime taken: %f seconds\n", cpu\_time\_used);

return 0;

}

**Screenshot of Output**

****

**Program full details**

Sort a given set of N integer elements using Heap Sort technique and compute its time taken.

**Code**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include<unistd.h>

void swap(int \*a, int \*b)

{

int t = \*a;

\*a = \*b;

\*b = t;

}

void heapify(int a[], int n, int i)

{

int largest = i;

int left = 2 \* i + 1;

int right = 2 \* i + 2;

if (left < n && a[left] > a[largest])

largest = left;

if (right < n && a[right] > a[largest])

largest = right;

if (largest != i)

{

swap(&a[largest], &a[i]);

heapify(a, n, largest);

}

}

void heapsort(int a[], int n)

{

for (int i = n / 2 - 1; i >= 0; i--)

heapify(a, n, i);

for (int i = n - 1; i >= 0; i--)

{

swap(&a[i], &a[0]);

heapify(a, i, 0);

}

}

int main()

{

int a[] = {5, 8, 2, 0, 1};

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

clock\_t start, end;

double cpu\_time\_used;

start = clock();

heapsort(a, n);

usleep(2000);

end = clock();

cpu\_time\_used = ((double)(end - start)) / CLOCKS\_PER\_SEC;

printf("Sorted Array:\n");

for (int i = 0; i < n; i++)

{

printf("%d ", a[i]);

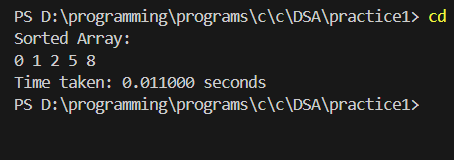
}

printf("\nTime taken: %f seconds\n", cpu\_time\_used);

return 0;

}

**Screenshot of Output**

****

**Program full details**

Implement 0/1 Knapsack problem using dynamic programming.

**Code**

#include <stdio.h>

#include <stdlib.h>

// Utility function to get max of two integers

int max(int a, int b) {

return (a > b) ? a : b;

}

// Knapsack function

void knapSack(int\*\* WeightCost, int size, int capacity) {

int\*\* table = (int\*\*)malloc((size + 1) \* sizeof(int\*));

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

table[i] = (int\*)calloc(capacity + 1, sizeof(int));

}

for (int i = 1; i <= size; i++) {

for (int w = 1; w <= capacity; w++) {

int weight = WeightCost[i - 1][0];

int value = WeightCost[i - 1][1];

if (weight <= w) {

table[i][w] = max(table[i - 1][w], table[i - 1][w - weight] + value);

} else {

table[i][w] = table[i - 1][w];

}

}

}

printf("Knapsack Table\n");

for(int i=0;i<=size;i++)

{

for(int j=0;j<=capacity;j++)

{

printf("%d ",table[i][j]);

}

printf("\n");

}

// Final result

printf("Maximum profit: %d\n", table[size][capacity]);

// Cleanup

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

free(table[i]);

}

free(table);

}

// Driver code

int main() {

int size, capacity;

printf("Enter number of items: ");

scanf("%d", &size);

printf("Enter knapsack capacity: ");

scanf("%d", &capacity);

int\*\* WeightCost = (int\*\*)malloc(size \* sizeof(int\*));

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

WeightCost[i] = (int\*)malloc(2 \* sizeof(int));

printf("Enter weight and value for item %d: ", i + 1);

scanf("%d %d", &WeightCost[i][0], &WeightCost[i][1]);

}

knapSack(WeightCost, size, capacity);

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

free(WeightCost[i]);

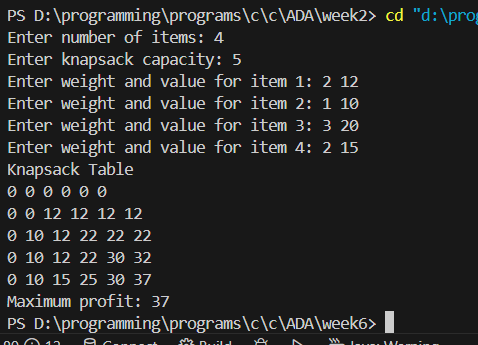
}

free(WeightCost);

return 0;

}

**Screenshot of Output**

****

Implement 0/1 Knapsack problem using dynamic programming.

**Program full details**

LeetCode Program related to Knapsack problem or Dynamic Programming.

**Code**

int fib(int n){

if(n==0 || n==1)

{

return n;

}

int dp[n+1];

dp[0]=0;

dp[1]=1;

for(int i=2;i<=n;i++)

{

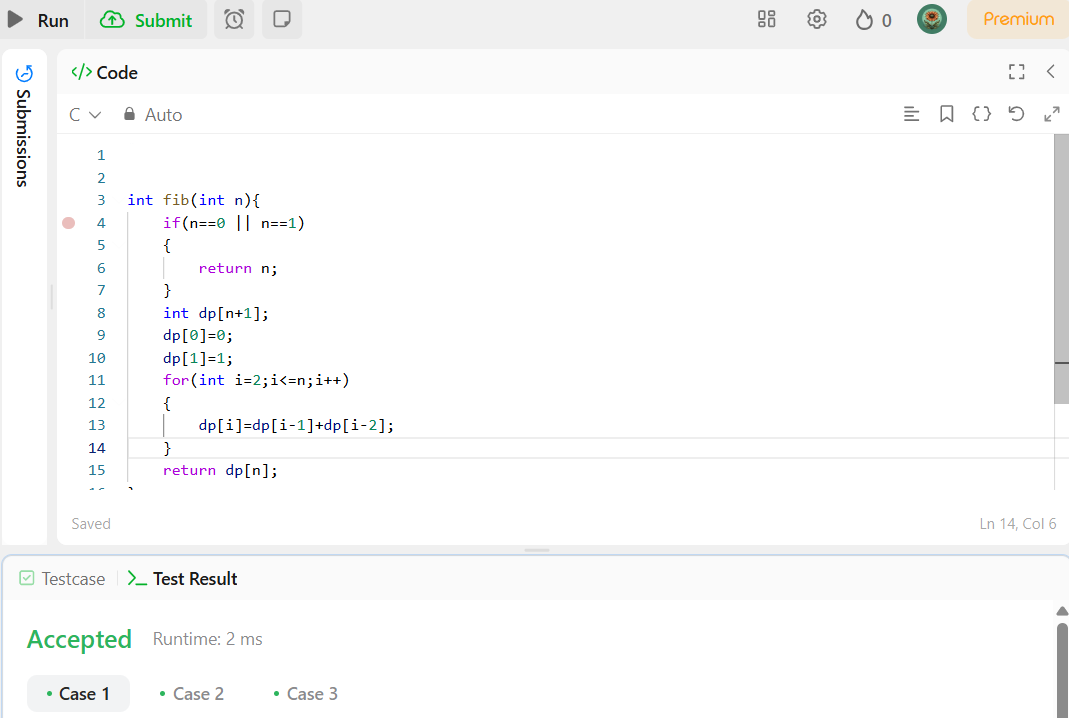
dp[i]=dp[i-1]+dp[i-2];

}

return dp[n];

}

**Screenshot of Output**

****

**Program full details**

Implement All Pair Shortest paths problem using Floyd’s algorithm.

**Code**

//Program to implement  FLoyds Algorithm:

#include<stdio.h>

#include<stdlib.h>

#define V 4

#define INF 999

void floyds(int graph[V][V])

{

    int dist[V][V];

    for(int i=0;i<V;i++)

    {

        for(int j=0;j<V;j++)

        {

            dist[i][j]=graph[i][j];

        }

    }

    for(int i=0;i<V;i++)

    {

        for(int j=0;j<V;j++)

        {

            for(int k=0;k<V;k++)

            {

                if(dist[i][k]+dist[k][j]<dist[i][j])

                {

                    dist[i][j]=dist[i][k]+dist[k][j];

                }

            }

        }

    }

    printf("The min distances are:\n");

    for(int i=0;i<V;i++)

    {

        for(int j=0;j<V;j++)

        {

            if(dist[i][j]==INF)

            {

                printf("INF ");

            }

            else{

                printf("%d ",dist[i][j]);

            }

        }

        printf("\n");

    }

}

int main()

{

    int graph[V][V]={

        {0,5,INF,10},

        {INF,0,3,INF},

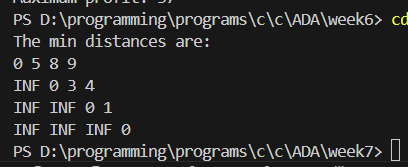
        {INF,INF,0,1},

        {INF,INF,INF,0}

    };

    floyds(graph);

}



LeetCode Program related to shortest distance calculation.

**Code**

#include <stdlib.h>

#include <string.h>

#define MAX 12

typedef struct {

int node;

int mask;

int dist;

} State;

int shortestPathLength(int\*\* graph, int graphSize, int\* graphColSize) {

int finalState = (1 << graphSize) - 1;

int seen[MAX][1 << MAX];

memset(seen, 0, sizeof(seen));

State\* queue = (State\*)malloc(sizeof(State) \* graphSize \* (1 << graphSize));

int front = 0, rear = 0;

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

int mask = (1 << i);

queue[rear++] = (State){i, mask, 0};

seen[i][mask] = 1;

}

while (front < rear) {

State current = queue[front++];

if (current.mask == finalState) {

free(queue);

return current.dist;

}

for (int i = 0; i < graphColSize[current.node]; i++) {

int neighbor = graph[current.node][i];

int nextMask = current.mask | (1 << neighbor);

if (!seen[neighbor][nextMask]) {

seen[neighbor][nextMask] = 1;

queue[rear++] = (State){neighbor, nextMask, current.dist + 1};

}

}

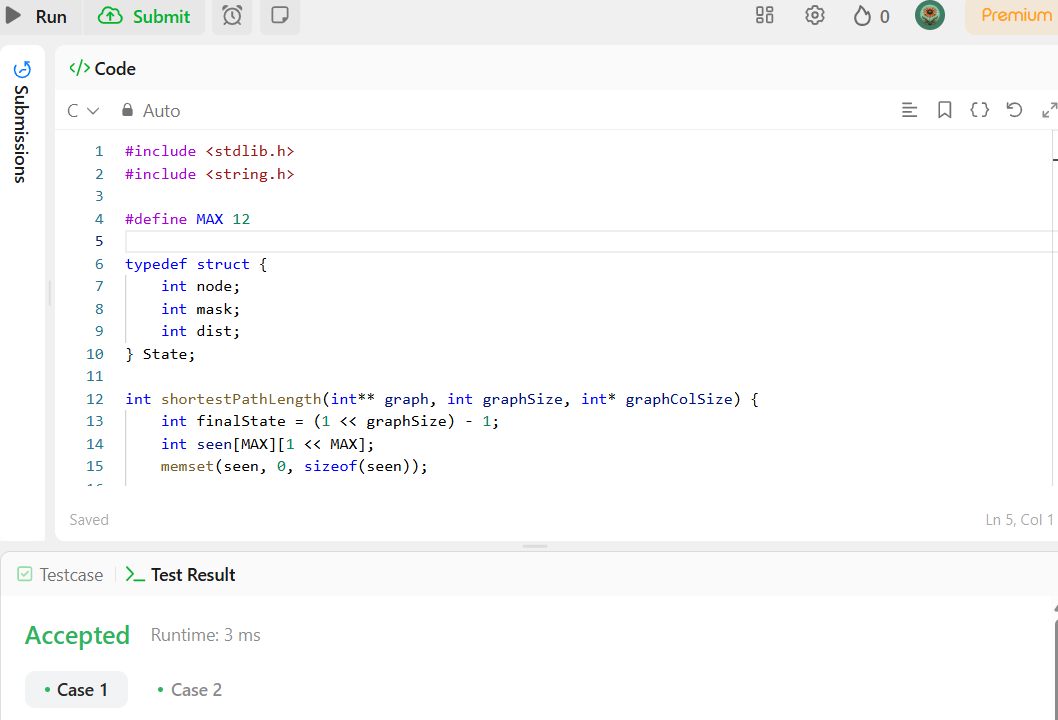
}

free(queue);

return -1; // should never reach here since the graph is connected

}

**Screenshot of Output**

****

**Program full details**

Find Minimum Cost Spanning Tree of a given undirected graph using Prim’s algorithm.

**Code**

//program to implement Prims Algorithm:

#include<stdio.h>

#include<stdlib.h>

#define V 5

#define INF 999

int minkey(int key[],int mstset[])

{

int min=INF, minind=-1;

for(int i=0;i<V;i++)

{

if(!mstset[i] && key[i]<min)

{

min=key[i];

minind=i;

}

}

return minind;

}

void prims(int graph[V][V])

{

int\* key=(int\*)malloc(V\*sizeof(int));

int\* parent=(int\*)malloc(V\*sizeof(int));

int\* mstset=(int\*)calloc(V,sizeof(int));

for(int i=0;i<V;i++)

{

key[i]=INF;

}

key[0]=0;

parent[0]=-1;

for(int i=0;i<V-1;i++)

{

int u=minkey(key,mstset);

mstset[u]=1;

for(int v=0;v<V;v++)

{

if(!mstset[v] && graph[u][v] && key[u]!=INF && graph[u][v]<key[v])

{

parent[v]=u;

key[v]=graph[u][v];

}

}

}

printf("the minimum spanning tree according to prims Algorithm is:\n");

int sum=0;

for (int i = 1; i < V; i++) {

printf("%d - %d \t%d\n", parent[i], i, graph[i][parent[i]]);

sum += graph[i][parent[i]];

}

printf("\nthe total cost %d",sum);

}

int main()

{

int graph[V][V];

printf("Enter the adjacency matrix (%d x %d):\n", V, V);

for (int i = 0; i < V; i++)

{

for (int j = 0; j < V; j++)

{

scanf("%d", &graph[i][j]);

}

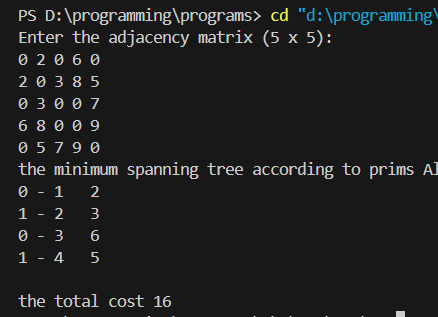
}

prims(graph);

return 0;

}

**Screenshot of Output**



Find Minimum Cost Spanning Tree of a given undirected graph using Kruskal’s algorithm.

**Code**

#include<stdio.h>

#include<stdlib.h>

#define Max 10

typedef struct{

int u,v,w;

}Edge;

int findparent(int\* parent,int i)

{

if(parent[i]==i)

{

return i;

}

return findparent(parent,parent[i]);

}

void unionset(int\* rank,int\* parent,int u,int v)

{

int rootu=findparent(parent,u);

int rootv=findparent(parent,v);

if(rank[rootu]>rank[rootv])

{

parent[rootv]=rootu;

}

else if(rank[rootv]>rank[rootu])

{

parent[rootu]=rootv;

}

else{

parent[rootv]=rootu;

rank[rootu]++;

}

}

void kruskals(Edge edges[],int v,int e)

{

int\* parent=(int\*)malloc(v\*sizeof(int));

int\* rank=(int\*)malloc(v\*sizeof(int));

for(int i=0;i<v;i++)

{

parent[i]=i;

rank[i]=0;

}

for (int i = 0; i < e - 1; i++) {

for (int j = 0; j < e - i - 1; j++) {

if (edges[j].w > edges[j + 1].w) {

Edge temp = edges[j];

edges[j] = edges[j + 1];

edges[j + 1] = temp;

}

}

}

int cost=0;

for(int i=0,count=0;i<e && count<v-1;i++)

{

int u=edges[i].u;

int v=edges[i].v;

int setu=findparent(parent,u);

int setv=findparent(parent,v);

if(setu!=setv)

{

cost+=edges[i].w;

printf("%d-%d\t %d",edges[i].u,edges[i].v,edges[i].w);

unionset(rank,parent,setu,setv);

count++;

}

}

printf("Total cost of MST: %d\n", cost);

free(parent);

free(rank);

}

int main()

{

int v,e;

printf("enter the number of vertices and edges\n");

scanf("%d%d",&v,&e);

Edge\* edges=(Edge\*)malloc(e\*sizeof(Edge));

printf("entter the vertices u v and weight\n");

for(int i=0;i<e;i++)

{

scanf("%d%d%d",&edges[i].u,&edges[i].v,&edges[i].w);

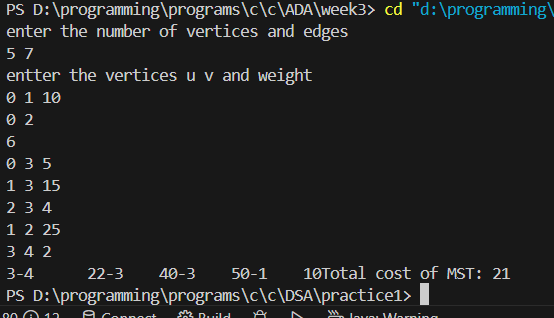
}

kruskals(edges,v,e);

return 0;

}

**Screenshot of Output**

****

**Program full details**

Implement Fractional Knapsack using Greedy technique.

**Code**

#include<stdio.h>

#include<stdlib.h>

void swap(float\* a,float\* b)

{

    float t=\*a;

    \*a=\*b;

    \*b=t;

}

void fractional(float\* weight,float\* value,int n,int capacity)

{

    float\* ratio = (float\*)malloc(n \* sizeof(float));

    float\*\* selected = (float\*\*)malloc(n \* sizeof(float\*));

    float profit = 0.0;

    for(int i=0; i<n; i++)

    {

        selected[i] = (float\*)malloc(2 \* sizeof(float));

        selected[i][0] = 0.0;

        selected[i][1] = 0.0;

    }

    for(int i=0; i<n; i++)

    {

        ratio[i] = value[i] / weight[i];

    }

    // Sort in descending order of ratio

    for(int i=0; i<n-1; i++)

    {

        for(int j=0; j<n-i-1; j++)

        {

            if(ratio[j] < ratio[j+1])

            {

                swap(&ratio[j], &ratio[j+1]);

                swap(&weight[j], &weight[j+1]);

                swap(&value[j], &value[j+1]);

            }

        }

    }

    for(int i=0; i<n && capacity>0; i++)

    {

        if(weight[i] <= capacity)

        {

            selected[i][0] = weight[i];

            selected[i][1] = value[i];

            profit += value[i];

            capacity -= weight[i];

        }

        else

        {

            float fraction = (float)capacity / weight[i];

            selected[i][0] = capacity;

            selected[i][1] = value[i] \* fraction;

            profit += selected[i][1];

            capacity = 0;

        }

    }

    printf("\nThe selected items are:\n");

    for(int i=0; i<n; i++)

    {

        if(selected[i][0] > 0)

            printf("Weight: %.2f Value: %.2f\n", selected[i][0], selected[i][1]);

    }

    printf("Total profit: %.2f\n", profit);

    for(int i=0; i<n; i++)

        free(selected[i]);

    free(selected);

    free(ratio);

}

int main()

{

    int n;

    printf("Enter the number of items:\n");

    scanf("%d", &n);

    int capacity;

    printf("Enter the capacity of the knapsack:\n");

    scanf("%d", &capacity);

    float\* weight = (float\*)malloc(n \* sizeof(float));

    float\* value = (float\*)malloc(n \* sizeof(float));

    printf("Enter the weights and values:\n");

    for(int i=0; i<n; i++)

    {

        printf("Item %d - Weight and Value: ", i+1);

        scanf("%f%f", &weight[i], &value[i]);

    }

    fractional(weight, value, n, capacity);

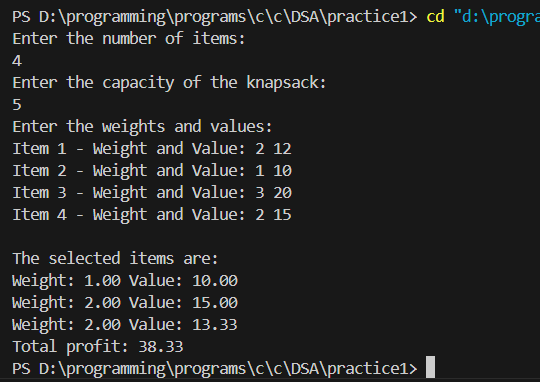
    free(weight);

    free(value);

    return 0;

}

**Screenshot of Output**



LeetCode Program related to Greedy Technique algorithms.

**Code**

#include <string.h>

char\* largestOddNumber(char\* num) {

int len = strlen(num);

for (int i = len - 1; i >= 0; i--) {

if ((num[i] - '0') % 2 != 0) {

num[i + 1] = '\0'; // truncate after the last odd digit

return num;

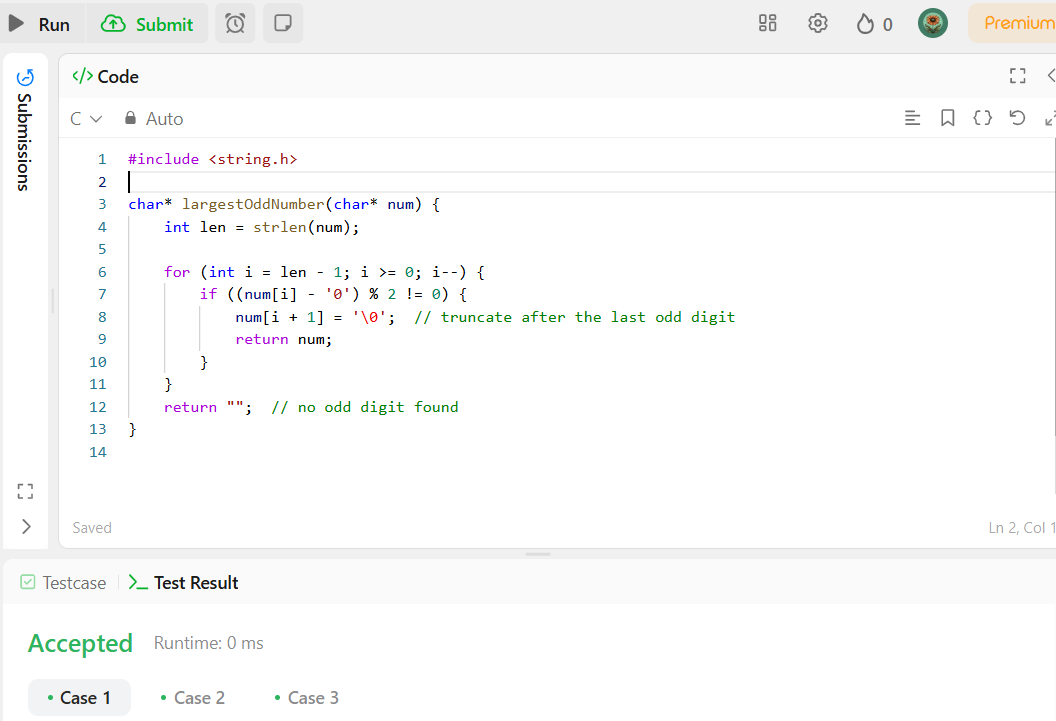
}

}

return ""; // no odd digit found

}

**Screenshot of Output**

****

**Program full details**

From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra’s algorithm.

**Code**

//Program to implement dijkstra's Algorithm:

#include<stdio.h>

#include<stdlib.h>

#define INF 999

int minDistance(int\* dist,int\* visited,int n)

{

int mindist=9999,minind=-1;

for(int i=0;i<n;i++)

{

if(dist[i]<mindist && !visited[i])

{

mindist=dist[i];

minind=i;

}

}

return minind;

}

void dijkstra(int graph[10][10],int n,int src)

{

int\* dist=(int\*)malloc(n\*sizeof(int));

int\* visited=(int\*)calloc(n,sizeof(int));

for(int i=0;i<n;i++)

{

dist[i]=INF;

}

dist[src]=0;

for(int i=0;i<n-1;i++)

{

int u=minDistance(dist,visited,n);

visited[u]=1;

for(int i=0;i<n;i++)

{

if(!visited[i] && graph[u][i] && (dist[u]+graph[u][i])<dist[i])

{

dist[i]=dist[u]+graph[u][i];

}

}

}

printf("the shortest distance from the source:\n");

for(int i=0;i<n;i++)

{

printf("%d ",dist[i]);

}

}

int main() {

int graph[10][10] = {

{0, 4, 0, 0, 0, 0, 0, 8, 0},

{4, 0, 8, 0, 0, 0, 0, 11, 0},

{0, 8, 0, 7, 0, 4, 0, 0, 2},

{0, 0, 7, 0, 9, 14, 0, 0, 0},

{0, 0, 0, 9, 0, 10, 0, 0, 0},

{0, 0, 4, 14, 10, 0, 2, 0, 0},

{0, 0, 0, 0, 0, 2, 0, 1, 6},

{8, 11, 0, 0, 0, 0, 1, 0, 7},

{0, 0, 2, 0, 0, 0, 6, 7, 0}

};

int n = 9; // Number of vertices

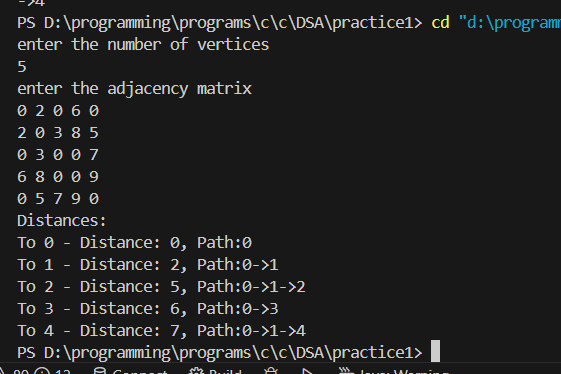
int src = 0;

dijkstra(graph, n, src);

return 0;

}

**Screenshot of Output**

****

**Program full details**

Implement “N-Queens Problem” using Backtracking.

**Code**

#include<stdio.h>

#include<stdlib.h>

int isSafe(int\* position, int row) {

for (int col = 0; col < row; col++) {

// Check same column or diagonal conflicts

if (position[row] == position[col] ||

abs(position[row] - position[col]) == abs(row - col)) {

return 0;

}

}

return 1;

}

void printsol(int\* position,int n)

{

for(int i=0;i<n;i++)

{

printf("%d ",position[i]);

}

printf("\n");

}

void nqueens(int n)

{

int\* position=(int\*)malloc(n\*sizeof(int));

for(int i=0;i<n;i++)

{

position[i]=-1;

}

int row=0;

int count=0;

position[row]=0;

while(row>=0)

{

while(position[row]<n && !isSafe(position,row))

{

position[row]++;

}

if(position[row]<n)

{

if(row==n-1)

{

count++;

printf("Solution %d\n",count);

printsol(position,n);

position[row]++;

}

else{

row++;

position[row]=0;

}

}

else{

position[row]=-1;

row--;

if (row >= 0) {

position[row]++;

}

}

}

printf("\nsolutins: %d",count);

}

int main()

{

int count=0;

nqueens(4);

return 0;

}

**Screenshot of Output**

