LAB 8

1)Implement Johnson Trotter algorithm to generate permutations.

Code:

#include <stdio.h>

#include <stdlib.h>

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

{

int temp = \*a;

\*a = \*b;

\*b = temp;

}

void generatePermutations(int arr[], int start, int end)

{

if (start == end)

{

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

{

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

}

printf("\n");

}

else

{

for (int i = start; i <= end; i++)

{

swap(&arr[start], &arr[i]);

generatePermutations(arr, start + 1, end);

swap(&arr[start], &arr[i]);

}

}

}

int main()

{

int n;

printf("Enter the number of elements: ");

scanf("%d", &n);

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

printf("Enter the elements: ");

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

{

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

}

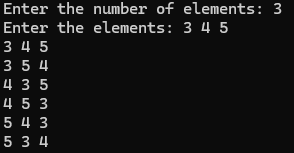
generatePermutations(arr, 0, n - 1);

free(arr);

return 0;

}

OUTPUT:



2) Implement 0/1 Knapsack problem using dynamic programming.

Code:

#include <stdio.h>

int n,m,w[10],p[10],v[10][10];

void knapsack(int,int,int[],int[]);

int max(int,int);

int main()

{

int i,j;

printf("Enter the no. of items:");

scanf("%d",&n);

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

scanf("%d",&m);

printf("Enter weights:");

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

{

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

}

printf("Enter profits:");

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

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

}

knapsack(n,m,w,p);

printf("Optimal Solution:\n");

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

{

for(j=0;j<n;j++)

{

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

}

printf("\n");

}

return 0;

}

void knapsack(int n, int m, int w[],int p[])

{

int i,j;

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

{

for(j=0;j<m;j++)

{

if(i==0 || j==0)

{

v[i][j]=0;

}

else if(w[i]>j)

{

v[i][j]=v[i-1][j];

}

else

{

v[i][j]=max(v[i-1][j],((v[i-1][j-w[i]])+p[i]));

}

}

}

}

int max(int a,int b)

{

if(a>b)

{

return a;

}

else

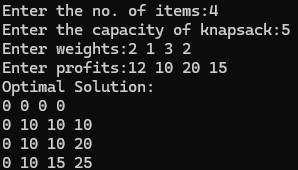
{

return b;

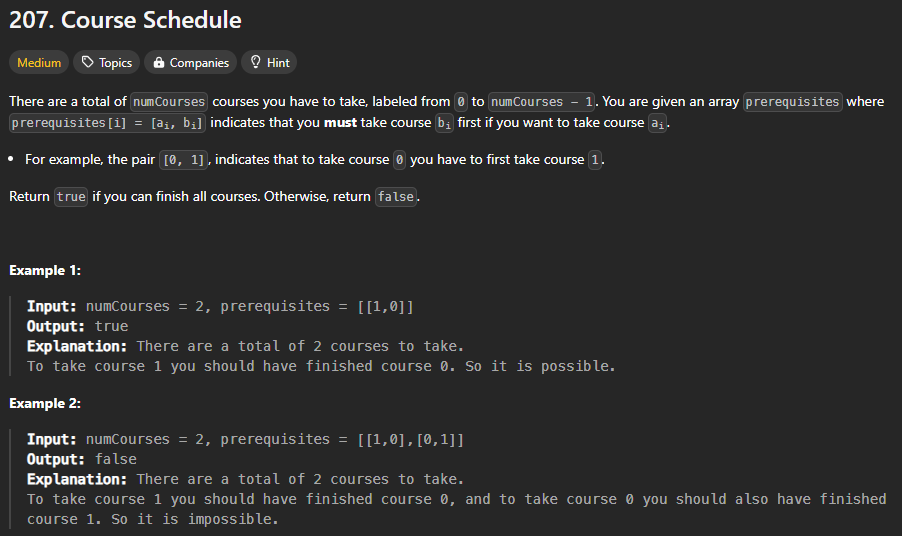
}

}

OUTPUT:



LEETCODE (TOPOLOGICAL SORT TECHNIQUE)



CODE:

#include <stdio.h>

#include <stdlib.h>

#define MAX\_COURSES 10000

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

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

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

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

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

        graph[i] = (int\*)malloc(MAX\_COURSES \* sizeof(int));

    }

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

        int course = prerequisites[i][0];

        int prereq = prerequisites[i][1];

        graph[prereq][graphSize[prereq]++] = course;

        inDegree[course]++;

    }

    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 < graphSize[course]; i++) {

            int neighbor = graph[course][i];

            inDegree[neighbor]--;

            if (inDegree[neighbor] == 0) {

                queue[rear++] = neighbor;

            }

        }

    }

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

        free(graph[i]);

    }

    free(graph);

    free(graphSize);

    free(inDegree);

    free(queue);

    return count == numCourses;

}

OUTPUT:

