**III: Dynamic Programing Date:**

**Aim:-**Write algorithm and C program to implement the following problems using Dynamic programing

1. Forward Graph
2. Backward Graph
3. All Pair Shortest Path
4. Belllman Ford
5. Optimal Binary Search Tree
6. 0/1 KnapSack

Dynamic Programming (DP) is a powerful algorithmic technique used to solve optimization problems by breaking them down into simpler subproblems and solving each subproblem only once. It is based on the principle of storing the solutions to subproblems in a table and reusing them whenever they are needed again, thereby avoiding redundant computations.

The DP approach typically consists of the following steps:

1. Define Subproblems: Identify the subproblems that make up the larger problem and determine how they relate to each other.

2. Formulate Recurrence Relations: Express the solution to each subproblem in terms of solutions to smaller subproblems. This is often done using recurrence relations or equations.

3. Memorization or Tabulation:Implement a method to store the solutions to subproblems, either through memorization (top-down approach) or tabulation (bottom-up approach). Memorization involves storing solutions in a data structure like an array or hash table, while tabulation involves filling up a table iteratively.

4. Solve Subproblems: Use a recursive or iterative approach to solve each subproblem based on the memorized or tabulated solutions.

5. Combine Solutions: Combine the solutions to smaller subproblems to obtain the solution to the original problem.

Dynamic Programming is particularly useful when subproblems overlap or exhibit optimal substructure, meaning that the solution to the original problem can be constructed from solutions to its subproblems.

1. **Forward Graph**

**Date:**

#include<stdio.h>

#include<stdlib.h>

#include<limits.h>

void CreateWeightedGraph(int \*\*cost, int n)

{

    for(int i = 0; i < n; i++)

    {

        for(int j = 0; j < n; j++)

        {

            if(i != j)

                cost[i][j] = INT\_MAX;

            else

                cost[i][j] = 0;

        }

    }

    int x,y,weight;

    while(1)

    {

        printf("Enter edge (x, y) Enter (-1 -1) to exit: \n");

        scanf("%d%d",&x,&y);

        if(--x >= n || --y >= n)

        {

            printf("Node exceeds size limit\nEnter again\n");

        }

        else if(x < 0 && y < 0 )

        {

            break;

        }

        else{

            if(cost[x][y] != INT\_MAX)

            {

                printf("An edge already exists at (%d , %d)",x,y);

                continue;

            }

            printf("Enter Weight(cost) of edge : ");

            scanf("%d",&weight);

            cost[x][y] = weight;

        }

    }

}

int FindMinEdge(int \*\*C, int n, int j, int \*cost)

{

    int k = 0;

    int optimalcost = INT\_MAX;

    for(int i=0; i<n; i++)

    {

        if((C[j][i] != INT\_MAX && j != i))

        {

            if(C[j][i] + cost[i] < optimalcost)

            {

                k = i;

                optimalcost= C[j][i] + cost[i];

            }

        }

    }

    return k;

}

void PrintPath(int \*path, int k)

{

    printf("\nShortest Path :\n");

    for(int i=0; i<k; i++)

    {

        if(i==k-1)

            printf(" %d ",path[i]+1);

        else

            printf(" %d ->",path[i]+1);

    }

}

void PrintTable(int\*\*C, int\*cost, int\*path, int \*stages, int \*d, int n, int k)

{

    int l = k-2;

    printf("\nVertex\t\t\tCost\t\t\tMinimum Values Considered\n");

    for (int i = n - 2; i >= 0; i--)

    {

        if (path[l] == i){

            printf("|d[%d,%d]|=|%d|\t\tcost[%d,%d]=%d\t\t\tmin(", stages[i], i + 1, d[i] + 1, stages[i], i + 1, cost[i]);

        }

        else

            printf("d[%d,%d]=%d\t\tcost[%d,%d]=%d\t\t\tmin(", stages[i], i + 1, d[i] + 1, stages[i], i + 1, cost[i]);

        int first = 1; // Flag to handle printing comma

        for (int j = i + 1; j < n; j++)

        {

            if (C[i][j] != INT\_MAX)

            {

                if (!first)

                {

                    printf(", ");

                }

                printf("%d", cost[j]+C[i][j]);

                first = 0;

            }

        }

        printf(")");

        if (path[l] == i)

        {

            printf(" [Path]");

            l--;

        }

        printf("\n");

    }

}

void CalculateSortestPath(int\*\*C, int\*cost, int\*path, int n, int k)

{

    int D[100] = {INT\_MAX};

    int stages[100] = {-1};

    cost[n-1] = 0;

    D[n-1] = n-1;

    stages[n-1] = k;

    for(int j=n-2; j>=0; j--)

    {

        int r = FindMinEdge(C,n,j,cost);

        cost[j] = C[j][r] + cost[r];

        D[j] = r;

        stages[j] = stages[D[j]]-1;

    }

    path[0]=0;

    path[k-1]=n-1;

    for(int j = 1; j<=k-1; j++)

    {

        path[j] = D[path[j-1]];

    }

    PrintTable(C,cost,path,stages,D,n,k);

}

int main()

{

    int \*\*C;

    int \*cost;

    int \*path;

    int n, k;

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

    scanf("%d",&n);

    printf("Enter number of stages : ");

    scanf("%d",&k);

    C = malloc(sizeof(\*C)\*n);

    cost = malloc(sizeof(\*cost)\*n);

    path = malloc(sizeof(\*path)\*k);

    for(int i=0; i<n; i++)

    {

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

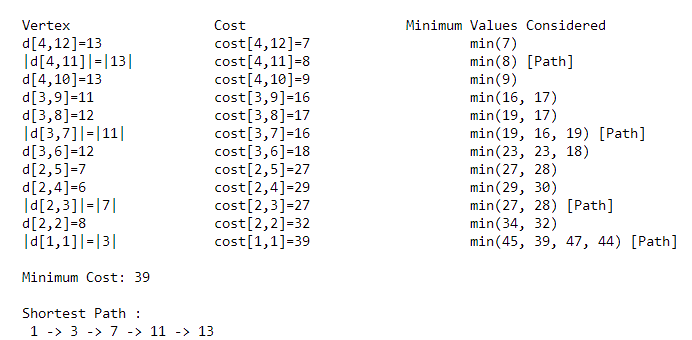
    }

    CreateWeightedGraph(C,n);

    CalculateSortestPath(C,cost,path,n,k);

    PrintPath(path,k);

}

**Output**

1. **Backward Graph**

**Date:**

#include<stdio.h>

#include<stdlib.h>

#include<limits.h>

void CreateWeightedGraph(int \*\*cost, int n)

{

    for(int i = 0; i < n; i++)

    {

        for(int j = 0; j < n; j++)

        {

            if(i != j)

                cost[i][j] = INT\_MAX;

            else

                cost[i][j] = 0;

        }

    }

    int x,y,weight;

    while(1)

    {

        printf("Enter edge (x, y) Enter (-1 -1) to exit: \n");

        scanf("%d%d",&x,&y);

        if(--x >= n || --y >= n)

        {

            printf("Node exceeds size limit\nEnter again\n");

        }

        else if(x < 0 && y < 0 )

        {

            break;

        }

        else{

            if(cost[x][y] != INT\_MAX)

            {

                printf("An edge already exists at (%d , %d)",x,y);

                continue;

            }

            printf("Enter Weight(cost) of edge : ");

            scanf("%d",&weight);

            cost[x][y] = weight;

        }

    }

}

int FindMinEdge(int \*\*C, int \*cost, int n, int j)

{

    int k = 0;

    int min = INT\_MAX;

    for(int i=0; i<n; i++)

    {

        if((C[i][j] != INT\_MAX && j != i) && (C[i][j] + cost[i]) < min)

        {

            min = C[i][j] + cost[i];

            k = i;

        }

    }

    return k;

}

void PrintPath(int \*path, int k)

{

    printf("\nShortest Path :\n");

    for(int i=k-1; i>=0; i--)

    {

        if(i==0)

            printf(" %d ",path[i]+1);

        else

            printf(" %d <-",path[i]+1);

    }

}

void PrintTable(int\*\*C, int\*cost, int\*path, int \*stages, int \*d, int n, int k)

{

    int l = 1;

    printf("\nVertex\t\t\tCost\t\t\tMinimum Values Considered\n");

    for (int i = 1; i <n; i++)

    {

        if (path[l] == i){

            printf("|d[%d,%d]|=|%d|\t\tcost[%d,%d]=%d\t\t\tmin(", stages[i], i + 1, d[i] + 1, stages[i], i + 1, cost[i]);

        }

        else

            printf("d[%d,%d]=%d\t\tcost[%d,%d]=%d\t\t\tmin(", stages[i], i + 1, d[i] + 1, stages[i], i + 1, cost[i]);

        int first = 1;

        for (int j = 0; j < n; j++)

        {

            if (C[j][i] != INT\_MAX)

            {

                if (!first)

                {

                    printf(", ");

                }

                printf("%d", cost[j]+C[j][i]);

                first = 0;

            }

        }

        printf(")");

        if (path[l] == i)

        {

            printf(" [Path]");

            l--;

        }

        printf("\n");

    }

}

void CalculateSortestPath(int\*\*C, int\*cost, int\*path, int n, int k)

{

    int D[100] = {INT\_MAX};

    int stages[100] ={-1};

    cost[0] = 0;

    D[0] = 0;

    stages[0] = 0;

    for(int j=1; j<=n-1; j++)

    {

        int r = FindMinEdge(C,cost,n,j);

        cost[j] = C[r][j] + cost[r];

        D[j] = r;

        stages[j] = stages[D[j]] + 1;

    }

    path[0]=0;

    path[k-1]=n-1;

    for(int j = k-2; j>=1; j--)

    {

        path[j] = D[path[j+1]];

    }

    PrintTable(C,cost,path,stages,D,n,k);

    printf("\nMinimum Cost: %d\n",cost[n-1]);

}

int main()

{

    int \*\*C;

    int \*cost;

    int \*path;

    int n, k;

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

    scanf("%d",&n);

    printf("Enter number of stages : ");

    scanf("%d",&k);

    C = malloc(sizeof(\*C)\*n);

    cost = malloc(sizeof(\*cost)\*n);

    path = malloc(sizeof(\*path)\*k);

    for(int i=0; i<n; i++)

    {

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

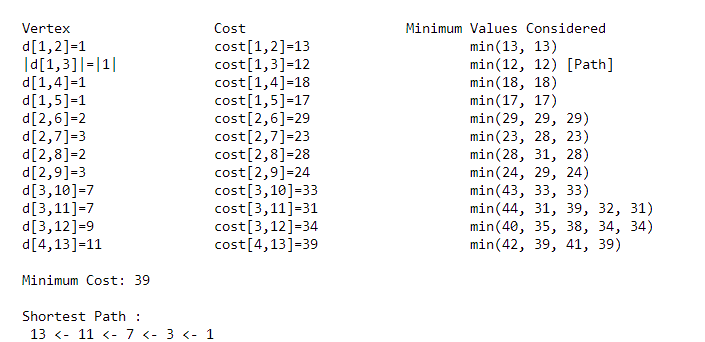
    }

    CreateWeightedGraph(C,n);

    CalculateSortestPath(C,cost,path,n,k);

    PrintPath(path,k);

}

**Output**

1. **All source shortest path**

**Date:**

#include<stdio.h>

#include<stdlib.h>

#include<limits.h>

typedef struct \_Path

{

    int\* list;

    int n;

}path;

void CreateWeightedGraph(int \*\*cost, int n)

{

    for(int i = 0; i < n; i++)

    {

        for(int j = 0; j < n; j++)

        {

            if(i != j)

                cost[i][j] = INT\_MAX;

            else

                cost[i][j] = 0;

        }

    }

    int x,y,weight;

    while(1)

    {

        printf("Enter edge (x, y) Enter (-1 -1) to exit: \n");

        scanf("%d%d",&x,&y);

        if(--x >= n || --y >= n)

        {

            printf("Node exceeds size limit\nEnter again\n");

        }

        else if(x < 0 && y < 0 )

        {

            break;

        }

        else{

            if(cost[x][y] != INT\_MAX)

            {

                printf("An edge already exists at (%d , %d)",x,y);

                continue;

            }

            printf("Enter Weight(cost) of edge : ");

            scanf("%d",&weight);

            cost[x][y] = weight;

        }

    }

}

int min(int a, int b)

{

    return (a < b ? a : b);

}

void Print(int \*\*A, int n, path \*\*P)

{

    printf("sr dist lenght path\n");

    for(int i=0; i<n; i++)

    {

        for(int j=0; j<n; j++)

        {

            if(i!=j)

            {

                printf("%d  %d    %2d     ",i+1,j+1,A[i][j]);

                for(int k=0; k<P[i][j].n; k++)

                {

                    if(k != P[i][j].n-1)

                        printf("%d->",P[i][j].list[k]);

                    else

                        printf("%d\n",P[i][j].list[k]);

                }

            }

        }

        printf("------------------------\n");

    }

}

void EditPath(int i, int j, int k, path \*\*P)

{

    int l=0, m=1;

    P[i][j].n = P[i][k].n + P[k][j].n - 1;

    free(P[i][j].list);

    P[i][j].list = malloc(sizeof(int)\*P[i][j].n);

    while( l<P[i][k].n)

        P[i][j].list[l] = P[i][k].list[l++];

    while( m<P[k][j].n)

        P[i][j].list[l++] = P[k][j].list[m++];

}

void PrintMatrix(int\*\*A, int n, int count)

{

    printf("A%d:\n",count);

    for(int i=0; i<n; i++)

    {

        for(int j=0; j<n; j++)

        {

            if(A[i][j] == INT\_MAX)

                printf(" IN");

            else

                printf(" %2d",A[i][j]);

        }

        printf("\n");

    }

    printf("\n");

}

void AllSourceShortestPath(int \*\*cost, int n, int \*\*A, path \*\*pathMatrix)

{

    int minimun;

    for(int i=0; i<n; i++)

        for(int j=0; j<n; j++){

            A[i][j] = cost[i][j];

            if(A[i][j] != INT\_MAX)

            {

                if(A[i][j] == 0){

                    pathMatrix[i][j].list = malloc(sizeof(int)\*1);

                    pathMatrix[i][j].n = 1;

                    pathMatrix[i][j].list[0] = i+1;

                }

                else{

                    pathMatrix[i][j].list = malloc(sizeof(int)\*2);

                    pathMatrix[i][j].n = 2;

                    pathMatrix[i][j].list[0] = i+1;

                    pathMatrix[i][j].list[1] = j+1;

                }

            }

        }

    PrintMatrix(A,n,0);

    for(int k=0; k<n; k++){

        for(int i=0; i<n; i++)

            for(int j=0; j<n; j++)

            {

                minimun = min(A[i][j], A[i][k] + A[k][j]);

                if(minimun < A[i][j])

                {

                    EditPath(i,j,k,pathMatrix);

                    A[i][j] = minimun;

                }

            }

        PrintMatrix(A,n,k+1);

    }

}

int main()

{

    int n;

    int\*\* cost, \*\*A;

    path \*\*pathMatrix;

    printf("Enter the number of vertices in the graph : ");

    scanf("%d",&n);

    cost = malloc(sizeof(\*cost)\*n);

    A = malloc(sizeof(\*A)\*n);

    pathMatrix = malloc(sizeof(\*pathMatrix)\*n);

    for(int i=0; i<n; i++)

    {

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

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

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

    }

    CreateWeightedGraph(cost,n);

    AllSourceShortestPath(cost,n,A,pathMatrix);

    Print(A,n, pathMatrix);

    printf("\n");

    return 0;

}

**Output**

**d) Bellmen Ford**

**Date:**

#include<stdio.h>

#include<stdlib.h>

#define INT\_MAX 99999

void CreateWeightedGraph(int \*\*cost, int n)

{

    for(int i = 0; i < n; i++)

    {

        for(int j = 0; j < n; j++)

        {

            if(i != j)

                cost[i][j] = INT\_MAX;

            else

                cost[i][j] = 0;

        }

    }

    int x,y,weight;

    while(1)

    {

        printf("Enter edge (x, y) Enter (-1 -1) to exit: \n");

        scanf("%d%d",&x,&y);

        if(--x >= n || --y >= n)

        {

            printf("Node exceeds size limit\nEnter again\n");

        }

        else if(x < 0 && y < 0 )

        {

            break;

        }

        else{

            if(cost[x][y] != INT\_MAX)

            {

                printf("An edge already exists at (%d , %d)",x,y);

                continue;

            }

            printf("Enter Weight(cost) of edge : ");

            scanf("%d",&weight);

            cost[x][y] = weight;

        }

    }

}

void RPrint(int \*pred, int source, int j)

{

    if(j == source)

        printf("%d",source+1);

    else{

        RPrint(pred,source,pred[j]);

        printf("->%d",j+1);

    }

}

void Print(int \*dist, int n, int v, int \*pred)

{

    printf("sr dist lenght path");

        for(int j=0; j<n; j++)

        {

            if(v!=j)

            {

                printf("\n%d  %d    %2d     ",v+1,j+1,dist[j]);

                RPrint(pred,v,j);

            }

        }

}

void BellmenFord(int \*\*cost, int n, int\* dist, int v)

{

    int oldDist[n], pred[n];

    for(int i=0; i<n; i++)

    {

        dist[i] = cost[v][i];

        oldDist[i] = dist[i];

        pred[i] = v;

    }

    printf("Vertex     ");

    for(int i=0; i<n; i++)

    {

        printf(" [%d] ",i+1);

    }

    printf("\nDistance   ");

    for(int i=0; i<n; i++)

    {

        if(dist[i] == INT\_MAX)

                printf(" inf ");

            else

                printf(" %3d ",dist[i]);

    }printf("\nPredcessor ");

    for(int i=0; i<n; i++)

    {

        printf("  %d  ", pred[i]+1);

    }printf("\n");

    for(int k=0; k<n-2; k++){

        for(int i=0; i<n; i++)

        {

            for(int u=0; u<n; u++)

            {

                if(u!=v)

                {

                    if(dist[u] > (oldDist[i] + cost[i][u]))

                    {

                        if(oldDist[i] == INT\_MAX && cost[i][u] == INT\_MAX)

                            dist[u] = INT\_MAX;

                        else{

                            dist[u]=(oldDist[i] + cost[i][u]);

                            pred[u] = i;

                        }

                    }

                }

            }

        }

        //output section

        printf("\nDistance   ");

        for(int j=0; j<n; j++)

        {

            if(dist[j] < oldDist[j])

            {

                printf(" \033[4m%3d\033[24m ",dist[j]);

            }

            else if(dist[j] == INT\_MAX)

                printf(" inf ");

            else

                printf(" %3d ",dist[j]);

        }

        printf("\nPredcessor ");

        for(int j=0; j<n; j++)

        {

            if(dist[j] < oldDist[j])

            {

                printf(" \033[4m[%d]\033[24m ",pred[j]+1);

                oldDist[j] = dist[j];

            }

            else

                printf("  %d  ",pred[j]+1);

        }

        printf("\n");

    }printf("\n");

    Print(dist,n,v,pred);

}

int main()

{

    int n, v;

    int\*\* cost, \*dist;

    printf("Enter the number of vertices in the graph : ");

    scanf("%d",&n);

    cost = malloc(sizeof(\*cost)\*n);

    dist = malloc(sizeof(dist)\*n);

    for(int i=0; i<n; i++)

    {

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

    }

    printf("Enter the starting vertex: ");

    scanf("%d", &v);

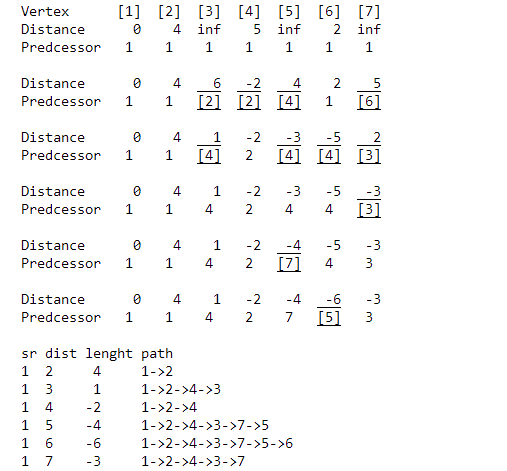
    CreateWeightedGraph(cost,n);

    BellmenFord(cost,n,dist,v-1);

    printf("\n");

    return 0;

}

**Output**

**e) OBST**

**Date:**

#include<stdio.h>

#include<stdlib.h>

#include<limits.h>

typedef struct identifiers

{

    char\* id;

    int num;

} idef;

int count =1;

void PrintHeader(float\* p, float\* q, idef\* ident, int n){

    printf("            ");

    for(int i = 1; i <= n; i++)

        printf("%10s |", &ident[i].id);

    printf("\n");

    printf("            ");

    for(int i = 1; i <= n; i++)

        printf("%10.0f |", p[i]);

    printf("\n");

    for(int i = 0; i <= n; i++)

        printf("%10.0f |", q[i]);

    printf("\n");

    for(int i = 0; i <= 12\*(n+1); i++)

        printf("-");

    printf("\n");

}

void PrintTable(float \*\*w, float \*\*c, int \*\*r, int k, int m){

    for(int i = 0; i <= k; i++)

        printf("w[%d][%d]=%2.0f |", i, i+m, w[i][i+m]);

    printf("\n");

    for(int i = 0; i <= k; i++)

        printf("c[%d][%d]=%2.0f |", i, i+m, c[i][i+m]);

    printf("\n");

    for(int i = 0; i <= k; i++)

        printf("r[%d][%d]=%2d |", i, i+m, r[i][i+m]);

    printf("\n");

    for(int i = 0; i <= 12\*(k+1); i++)

        printf("-");

    printf("\n");

}

void print\_tree(int i, int j, idef \*ident, int \*\*r)

{

    if(i<j)

    {

        if (j - i == 5)

        {

            printf("\n\n     ");

            printf("%s\n    /     \\ \n", &ident[r[i][j]].id);

        }

        print\_tree(i, r[i][j] - 1, ident, r);

        if (j - i == 1 && i == 0 && count == 1)

        {

            printf("%s", &ident[r[i][j]].id);

            count++;

        }

        print\_tree(r[i][j], j, ident, r);

        if (j - i == 3)

        {

            printf("\t  %s\n", &ident[r[i][j]].id);

        }

    }

}

void part2(int i,int j, idef \*ident,int \*\*r)

{

    if(i<j)

    {

        part2(i, r[i][j] - 1,ident,r);

        if(j-i==1&&i==2)

        {

        printf("         /\t\\");

        printf("\n      %s", &ident[r[i][j]]);

        }

        part2(r[i][j], j, ident, r);

        if(j-i==1&&i==4)

        printf("\t%s\n", &ident[r[i][j]]);

    }

}

int Find(float\*\*C, int\*\*R,int i, int j)

{

    float min = INT\_MAX;

    int l;

    for(int m=R[i][j-1]; m<=R[i+1][j]; m++)

    {

        if((C[i][m-1] + C[m][j]) < min)

        {

            min = C[i][m-1] + C[m][j];

            l = m;

        }

    }

    return l;

}

void OBST(float\*\*C,float\*\*W, int\*\*R,float \*p, float\*q, int n)

{

    for(int i=0; i<=n; i++)

    {

        W[i][i] = q[i];

        R[i][i] = 0;

        C[i][i] = 0;

        W[i][i+1] = q[i] + q[i+1] + p[i+1];

        R[i][i+1] = i+1;

        C[i][i+1] = q[i] + q[i+1] + p[i+1];

    }

    PrintTable(W,C,R,n,0);

    for(int m=2; m<=n; m++)

    {

        for(int i=0; i<n; i++)

        {

            int j = i+m;

            W[i][j] = W[i][j-1] + p[j] + q[j];

            int k = Find(C,R,i,j);

            C[i][j] = W[i][j] + C[i][k-1] + C[k][j];

            R[i][j] = k;

        }

        PrintTable(W,C,R,n-m+1,m-1);

    }

    PrintTable(W,C,R,0,n);

    printf("\n");

}

int main()

{

    int n;

    float \*\*cost;

    float \*\*W;

    int \*\*R;

    float \*p,\*q;

    idef \*ident;

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

    scanf("%d",&n);

    cost = malloc(sizeof(\*cost)\*(n+1));

    W = malloc(sizeof(\*W)\*(n+1));

    R = malloc(sizeof(\*R)\*(n+1));

    p = malloc(sizeof(float)\*(n+1));

    q = malloc(sizeof(float)\*(n+1));

    ident = malloc(sizeof(\*ident)\*(n+1));

    for(int i=0; i<=n+1; i++)

    {

        ident[i].id = malloc(sizeof(\*ident[i].id)\*10);

        cost[i]=malloc(sizeof(\*cost[i])\*(n+1));

        W[i]=malloc(sizeof(\*W[i])\*(n+1));

        R[i]=malloc(sizeof(\*R[i])\*(n+1));

    }

     printf("Enter the list of identifiers: \n");

    for(int i = 1; i <= n; i++)

        scanf("%s",&ident[i].id);

    for(int i=1; i<=n; i++)

    {

        printf("Enter P[%d]: ",i);

        scanf("%f",&p[i]);

    }

    for(int i=0; i<=n; i++)

    {

        printf("Enter Q[%d]: ",i);

        scanf("%f",&q[i]);

    }

    PrintHeader(p,q,ident,n);

    OBST(cost,W,R,p,q,n);

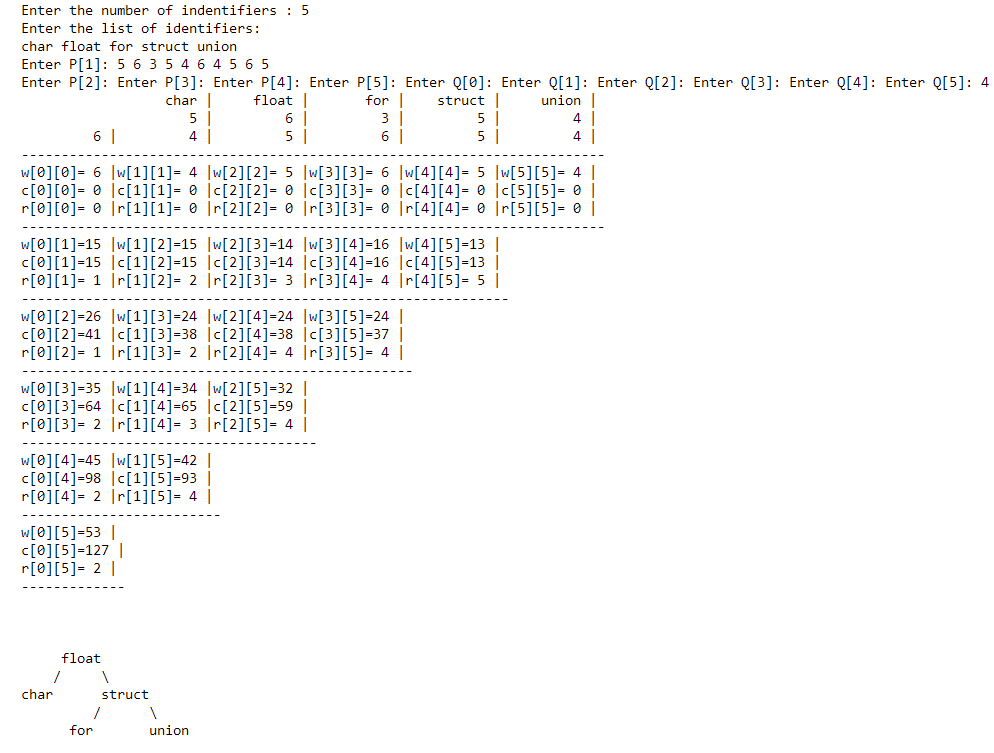
    print\_tree(0,n,ident,R);

    part2(0,n,ident,R);

    printf("\n");

    return 0;

}

**Output**

**d) 0/1 KnapSack**

**Date:**

#include <stdio.h>

#include <stdlib.h>

struct Pair

{

    double p; // Profit

    double w; // Weight

} pair[10000];

int b[101];

int maxm(int a, int b)

{

    return (a > b) ? a : b;

}

int Largest(struct Pair pair[], double w[], int t, int h, int i, int m)

{

    int q, max, mid, u = 0;

    max = 0;

    while (t <= h)

    {

        mid = (t + h) / 2;

        int y = pair[mid].w + w[i];

        if (y > max && y <= m)

        {

            max = pair[mid].w + w[i];

            u = mid;

        }

        if ((pair[mid].w + w[i]) <= m)

            t = mid + 1;

        else

            return u;

    }

    return u;

}

int BinarySearch(struct Pair pair[], int low, int high, double key)

{

    while (low <= high)

    {

        int mid = low + (high - low) / 2;

        if (pair[mid].p == key)

            return mid;

        else if (pair[mid].p < key)

            low = mid + 1;

        else

            high = mid - 1;

    }

    return high; // Return the index of the largest value less than or equal to the key

}

void TraceBack(double p[], double w[], struct Pair pair[], int x[], int to, int n)

{

    double totalProfit = pair[b[n + 1] - 1].p;

    double weight = pair[b[n + 1] - 1].w;

    printf(" \nMAX profit=%1.0f\nTotal weight used=%1.0f\n", totalProfit, pair[b[n + 1] - 1].w);

    double remainingWeight = to;

    for (int i = n - 1; i >= 0; i--)

    {

        int index = BinarySearch(pair, b[i], b[i + 1] - 1, totalProfit);

        if (pair[index].p == totalProfit)

        {

            x[i] = 0;

        }

        else

        {

            totalProfit -= p[i + 1];

            weight -= w[i + 1];

            x[i] = 1;

        }

    }

}

void AlgorithmDKnap(double p[], double w[], int x[], int n, int to)

{

    pair[1].p = 0;

    pair[1].w = 0;

    int t = 1;

    int h = 1;

    int next = 2;

    b[0] = 1;

    b[1] = 2;

    for (int i = 1; i <= n; i++)

    {

        int k = t;

        int u = Largest(pair, w, t, h, i, to);

        for (int j = t; j <= u; j++)

        {

            double pp = pair[j].p + p[i];

            double ww = pair[j].w + w[i];

            while (k <= h && pair[k].w < ww)

            {

                pair[next].p = pair[k].p;

                pair[next].w = pair[k].w;

                next++;

                k++;

            }

            if (k <= h && pair[k].w == ww)

            {

                if (pp < pair[k].p)

                {

                    pp = pair[k].p;

                }

                k++;

            }

            if (pp > pair[next - 1].p && ww <= to)

            {

                pair[next].p = pp;

                pair[next].w = ww;

                next++;

            }

            while (k <= h && pair[k].p <= pair[next - 1].p)

                k++;

        }

        while (k <= h)

        {

            pair[next].p = pair[k].p;

            pair[next].w = pair[k].w;

            next++;

            k++;

        }

        t = h + 1;

        h = next - 1;

        b[i + 1] = next;

    }

    TraceBack(p, w, pair, x, to, n);

    printf("\n\n");

    int cnt = 1;

    for (int i = 0; i < n + 1; i++)

    {

        printf(" S%d: ", i);

        for (int j = b[i]; j < b[i + 1]; j++)

        {

            printf(" (%.0lf,%.0lf ) ", pair[cnt].p, pair[cnt].w);

            cnt++;

        }

        printf("\n");

    }

    printf("\n");

}

int main()

{

    double p[100]; // Profits

    double w[100]; // Weights

    int x[100];    // Solution vector

    int n;         // Number of items

    int to;        // Maximum weight capacity

    // Input

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

    scanf("%d", &n);

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

    for (int i = 1; i <= n; i++)

        scanf("%lf", &p[i]);

    printf("\n");

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

    for (int i = 1; i <= n; i++)

        scanf("%lf", &w[i]);

    printf("\n");

    printf("Profit : ");

    for (int i = 1; i <= n; i++)

        printf("%1.0f ", p[i]);

    printf("\nWeight : ");

     for (int i = 1; i <= n; i++)

        printf("%1.0f ", w[i]);

    printf("\nEnter the maximum weight capacity: ");

    scanf("%d", &to);

    // Solve the knapsack problem

    AlgorithmDKnap(p, w, x, n, to);

    // Output the solution vector

    printf("Solution vector (1 for selected, 0 for not selected):\n");

    for (int i = 0; i < n; i++)

    {

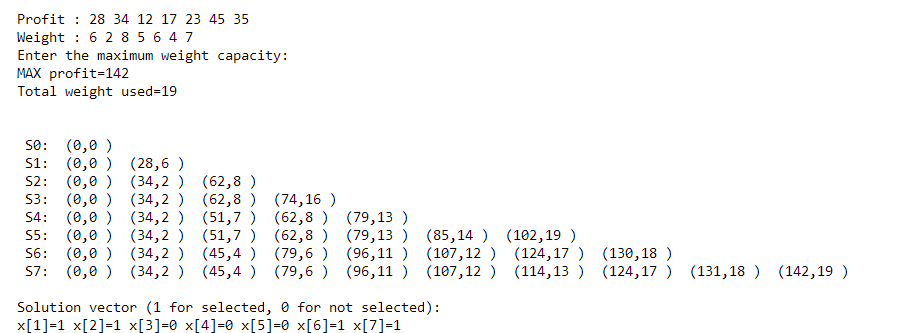
        printf("x[%d]=%d ", i + 1, x[i]);

    }

    printf("\n");

    return 0;

}

**Output**