EXPERIMENT 3 **DATE –**

DYNAMIC PROGRAMMING

1. Introduction to Dynamic Programming

Dynamic Programming (DP) is an algorithmic paradigm that solves complex problems by breaking them into smaller overlapping subproblems and storing their solutions to avoid redundant computations. It is particularly useful for optimization problems where the solution can be derived from optimal solutions to subproblems.

Key Characteristics of DP:

- **Optimal Substructure**: The optimal solution to a problem can be constructed from optimal solutions to its subproblems.
- **Overlapping Subproblems**: The problem can be divided into smaller subproblems that are reused multiple times.
- **Memoization/Tabulation**: Storing intermediate results to avoid recomputation.

2. Principle of Optimality

The **Principle of Optimality** (Bellman's Principle) states that:

"An optimal solution to a problem contains optimal solutions to its subproblems."

Example: Shortest Path Problem

- Suppose we need the shortest path from vertex **i** to **j**.
- If the optimal path is $\mathbf{i \rightarrow i_1 \rightarrow i_2 \rightarrow \dots \rightarrow i_k \rightarrow j}$, then the subpath $\mathbf{i_1 \rightarrow i_2 \rightarrow \dots \rightarrow j}$ must also be the shortest path from $\mathbf{i_1}$ to **j**.
- If not, we could replace it with a shorter path, contradicting optimality.

3. Dynamic Programming vs. Greedy Method

Feature	Greedy Method	Dynamic Programming
Decision Making	Makes locally optimal choices at each step.	Considers all possible decisions and selects the best one.
Optimality	Works if the problem has the **greedy choice property**.	Works if the problem has **optimal substructure** and **overlapping subproblems**.
Time Complexity	Often **O(n log n)** or **O(n)**.	Typically **polynomial ($O(n^2)$, $O(n^3)$, etc.)**.

Feature	Greedy Method	Dynamic Programming
Example Problems	Dijkstra's Algorithm, Kruskal's MST.	Knapsack, Longest Common Subsequence (LCS), Matrix Chain Multiplication.

4. Approaches in Dynamic Programming

A. Top-Down (Memoization)

- **Recursive approach** with stored results (caching).
- Uses **recursion + memoization** (e.g., Fibonacci with memoization).

B. Bottom-Up (Tabulation)

- **Iterative approach**, solves subproblems first and builds up to the main problem.
- Uses **tables (arrays/matrices)** to store intermediate results (e.g., DP tables in the Knapsack problem).

5. Steps to Solve DP Problems

1. **Define the Problem in Terms of Subproblems**
    - Identify how the problem can be broken down.
    - Example:
        - Fibonacci: F(n) = F(n-1) + F(n-2)
        - LCS: LCS(X, Y, m, n) = LCS(X, Y, m-1, n-1) + 1 if X[m] == Y[n]
2. **Formulate the Recurrence Relation**
    - Express the problem recursively.
    - Example (Knapsack):

```
if (wt[i-1] <= W)
  dp[i][W] = max(val[i-1] + dp[i-1][W-wt[i-1]], dp[i-1][W])
else
  dp[i][W] = dp[i-1][W]
```

3. **Decide on Memoization or Tabulation**
    - **Memoization**: Store computed values in a hash table or array.
    - **Tabulation**: Fill a DP table iteratively.
4. **Implement the Solution**
    - Write the recursive (top-down) or iterative (bottom-up) solution.
5. **Optimize Space (if possible)**
    - Some DP problems can be optimized to use **O(1) or O(n) space** instead of **O(n²)**.

Date -

MULTISTAGE GRAPHS (FORWARD AND BACKWARD APPROACH)

AIM – Write a C program to calculate shortest path for traversing multistage stage graph using forward and backward approach

Statement – Given a multistage graph with k stages ,n vertices and given edges ,implement multistage graph approach to determine shortest path for traversing the graph .

Input - Number of stages = 5 , number of vertices = 14 and all the edges of graph are inputed

Output - i} Shortest path for traversing the graph from 1st stage to 5th stage

ii} Shortest path for traversing the graph from 5th stage to 1st stage

ALGORITHM –

I] Algorithm FGraph(G,k,n,p)

```
// The input is a k-stage graph G=(V,E) with n vertices
// indexed in order of stages. E is a set of edges and c[i,j]
// is the cost of <i,j>. p[1:k] is a minimum-cost path.
{
  cost[n] := 0.0;
  for j := n-1 to 1 step -1 do
  { // Compute cost[j].
    Let r be a vertex such that <j,r> is an edge
    of G and c[j,r] + cost[r] is minimum;
    cost[j] := c[j,r] + cost[r];
    d[j] := r;
  }
  // Find a minimum-cost path.
  p[1] := 1; p[k] := n;
  for j := 2 to k-1 do p[j] := d[p[j-1]];
}
```

Recurrence Relation

Let n be the number of vertices and E be the total number of edges.

T(n)=O(E)

Time Complexity

I] Best Case:

O(E)

- Even in the best scenario, the algorithm must check all outgoing edges from each vertex to find the minimum-cost edge.
- So, the time depends on the total number of edges in the graph.

II] Average Case:

O(E)

- On average, each vertex has a few outgoing edges, and each is still checked once.
- The number of edge checks remains proportional to E, hence still linear in E.

III] Worst Case:

O(E)

- In the worst case, the graph is dense, and each vertex has many outgoing edges.
- All edges are processed once, making time complexity O(E).

Space Complexity

I] Best Case:

O(n)

- Only arrays cost[n], d[n], and p[k] are used.
- k (number of stages) is always ≤ n, so total space is linear.

II] Average Case:

O(n)

- No extra space is used per edge or any recursion.
- Space used depends only on number of vertices, not edge density.

III] Worst Case:

O(n)

- Even if the graph is fully connected, space usage remains the same.

- No additional space is used for edges or graphs beyond the input.

II]Algorithm BGraph(G,k,n,p)

// Same function as FGraph

```
bcost[1] := 0.0;
for j := 2 to n do
{ // Compute bcost[j].
  Let r be such that <r,j> is an edge of
  G and bcost[r] + c[r,j] is minimum;
  bcost[j] := bcost[r] + c[r,j];
  d[j] := r;
}
// Find a minimum-cost path.
p[1] := 1; p[k] := n;
for j := k - 1 to 2 do p[j] := d[p[j+1]];
}
```

Recurrence Relation

Let T(n) be the time to compute all bcost[j] for j = 2 to n.

Each bcost[j] is computed by checking all incoming edges <r, j>, so assuming average E/n incoming edges per node:

T(n)=O(E)

⏱ **Time Complexity**

I] Best Case: O(E)

- Even in the best case, all incoming edges to each vertex must be checked to find the minimum-cost path.
- Every edge is processed once, hence time is proportional to number of edges.

II] Average Case: O(E)

- For each node j, average number of incoming edges is E/n.
- Loop runs for n-1 iterations, each doing O(E/n) work → total O(E).

III] Worst Case: O(E)

- In a fully connected graph, each node may have up to n incoming edges.
- All edges still processed once, so worst-case time is still O(E).

💾 **Space Complexity**

I] Best Case: O(n)

- Arrays bcost[n], d[n], and path array p[k] are used.
- No dynamic allocation based on edges → space depends on n.

II] Average Case: O(n)

- Space remains linear in number of vertices, even with varying edge counts.
- Edge data is assumed to be part of input, not affecting auxiliary space.

III] Worst Case: O(n)

- Dense or sparse, space used for arrays doesn't change.
- No additional memory for recursive calls or per-edge storage.

PROGRAM –

```
#include <stdio.h>

#include <limits.h>

#include <stdlib.h>

#include <time.h>


#define MAXV 100

#define MAXS 20

#define MAXE 1000


typedef struct {

   int from;

   int to;

   int cost;

} Edge;


void print_forward_calculations(Edge edges[], int edge_count, int k, int n, int fcost[], int d[]) {

   printf("\nForward Approach - Detailed Calculations:\n");

   for (int j = n-1; j >= 1; j--) {

      int stage = k - ((j-1)/(n/k));

      printf("Stage %d, Vertex %d:\n", stage, j);

      printf("fcost(%d,%d) = min{", stage, j);


      int first = 1;

      for (int e = 0; e < edge_count; e++) {

         if (edges[e].from == j) {

            if (!first) printf(", ");

            printf("c(%d,%d)+fcost(%d,%d)=%d+%d=%d",

                   j, edges[e].to,
stage+1, edges[e].to,
edges[e].cost, fcost[edges[e].to],

                   edges[e].cost + fcost[edges[e].to]);

            first = 0;

         }

      }

      printf("} = %d\n", fcost[j]);

      printf("d(%d,%d) = %d\n\n", stage, j, d[j]);

   }

}


void print_backward_calculations(Edge edges[], int edge_count, int k, int n, int bcost[], int d[]) {

   printf("\nBackward Approach - Detailed Calculations:\n");

   for (int j = 2; j <= n; j++) {

      int stage = ((j-1)/(n/k)) + 1;

      printf("Stage %d, Vertex %d:\n", stage, j);

      printf("bcost(%d,%d) = min{", stage, j);


      int first = 1;

      for (int e = 0; e < edge_count; e++) {

         if (edges[e].to == j) {

            if (!first) printf(", ");

            printf("bcost(%d,%d)+c(%d,%d)=%d+%d=%d",

                   stage-1,
edges[e].from, edges[e].from, j,

                   bcost[edges[e].from],
edges[e].cost,

                   bcost[edges[e].from] + edges[e].cost);

            first = 0;

         }

      }

      printf("} = %d\n", bcost[j]);

      printf("d(%d,%d) = %d\n\n", stage, j, d[j]);

   }

}


void forward(Edge edges[], int edge_count, int k, int n, int p[], int show_steps) {

   clock_t start = clock();

   int fcost[MAXV], d[MAXV];

   fcost[n] = 0;

   d[n] = -1;


   for (int j = 1; j < n; j++) fcost[j] = INT_MAX;


   for (int j = n-1; j >= 1; j--) {

      for (int e = 0; e < edge_count; e++) {

         if (edges[e].from == j && edges[e].cost + fcost[edges[e].to] < fcost[j]) {

            fcost[j] = edges[e].cost + fcost[edges[e].to];

            d[j] = edges[e].to;

         }

      }

   }
```

```
    if (show_steps)
print_forward_calculations(edges,
edge_count, k, n, fcost, d);


    p[1] = 1;

    p[k] = n;

    for (int j = 2; j <= k-1; j++) p[j] =
d[p[j-1]];


    clock_t end = clock();

    double time_taken =
((double)(end - start)) /
CLOCKS_PER_SEC;


    printf("\nForward Approach
Results:\n");

    printf("Minimum cost: %d\n",
fcost[1]);

    printf("Shortest path: ");

    for (int i = 1; i <= k; i++)
printf("%d ", p[i]);

    printf("\nPath details:\n");

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

        int edge_cost = 0;

        for (int e = 0; e < edge_count;
e++) {

            if (edges[e].from == p[i] &&
edges[e].to == p[i+1]) {

                edge_cost =
edges[e].cost;

                break;

            }

        }

        printf("Stage %d to %d: %d-
>%d (cost: %d)\n",

            i, i+1, p[i], p[i+1],
edge_cost);

    }

    printf("Execution time: %f
seconds\n", time_taken);

}
```

```
void backward(Edge edges[], int
edge_count, int k, int n, int p[], int
show_steps) {

    clock_t start = clock();

    int bcost[MAXV], d[MAXV];

    bcost[1] = 0;

    d[1] = -1;


    for (int j = 2; j <= n; j++) bcost[j]
= INT_MAX;


    for (int j = 2; j <= n; j++) {

        for (int e = 0; e < edge_count;
e++) {

            if (edges[e].to == j &&
bcost[edges[e].from] +
edges[e].cost < bcost[j]) {

                bcost[j] =
bcost[edges[e].from] +
edges[e].cost;

                d[j] = edges[e].from;

            }

        }

    }


    if (show_steps)
print_backward_calculations(edge
s, edge_count, k, n, bcost, d);


    p[1] = 1;

    p[k] = n;

    for (int j = k-1; j >= 2; j--) p[j] =
d[p[j+1]];


    clock_t end = clock();

    double time_taken =
((double)(end - start)) /
CLOCKS_PER_SEC;


    printf("\nBackward Approach
Results:\n");

    printf("Minimum cost: %d\n",
bcost[n]);

    printf("Shortest path: ");

    for (int i = k; i >= 1; i--)
printf("%d ", p[i]);

    printf("\nPath details:\n");

    for (int i = k-1; i >= 1; i--) {

        int edge_cost = 0;

        for (int e = 0; e < edge_count;
e++) {

            if (edges[e].from == p[i] &&
edges[e].to == p[i+1]) {

                edge_cost =
edges[e].cost;

                break;

            }

        }

        printf("Stage %d to %d: %d-
>%d (cost: %d)\n",

            i, i+1, p[i], p[i+1],
edge_cost);

    }

    printf("Execution time: %f
seconds\n", time_taken);

}


void show_edges(Edge edges[], int
edge_count) {

    printf("\nEdge List:\n");

    printf("From To Cost\n");

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

        printf("%4d %4d %4d\n",
edges[i].from, edges[i].to,
edges[i].cost);

    }

}
```

```
int main() {

    printf
("**************************
**************************"
);

  printf ("\n Roll number: 23B-CO-
010\n");

  printf (" PR Number -
202311390\n");

  printf("*******************
**************************
*****\n\n\n");

  int choice, k, n, p[MAXS],
show_steps;

  Edge edges[MAXE];

  int edge_count = 0;


  printf("Multistage Graph Path
Finder\n");


  while (1) {

    printf("\nMenu:\n");

    printf("1. Input Graph\n");

    printf("2. Run Forward
Approach\n");

    printf("3. Run Backward
Approach\n");

    printf("4. Show Edges\n");

    printf("5. Exit\n");

    printf("Enter choice: ");

    scanf("%d", &choice);


    switch (choice) {

      case 1: {

        printf("Number of stages
(k): ");

        scanf("%d", &k);

        printf("Number of
vertices (n): ");

        scanf("%d", &n);

        printf("Enter edges (from
to cost), -1 -1 -1 to finish:\n");

        edge_count = 0;

        while (1) {

          printf("Edge %d: ",
edge_count+1);

          scanf("%d %d %d",
&edges[edge_count].from,
&edges[edge_count].to,
&edges[edge_count].cost);


          if
(edges[edge_count].from == -1
&& edges[edge_count].to == -1
&& edges[edge_count].cost == -1)
{

            break;

          }


          edge_count++;

          if (edge_count >=
MAXE) {

            printf("Maximum
edges reached!\n");

            break;

          }

        }

        break;

      }


      case 2:

        if (edge_count == 0)
printf("Please input graph
first!\n");

        else {

          printf("Show detailed
calculations? (1=Yes, 0=No): ");

          scanf("%d",
&show_steps);

          forward(edges,
edge_count, k, n, p, show_steps);

        }

        break;


      case 3:

        if (edge_count == 0)
printf("Please input graph
first!\n");

        else {

          printf("Show detailed
calculations? (1=Yes, 0=No): ");

          scanf("%d",
&show_steps);

          backward(edges,
edge_count, k, n, p, show_steps);

        }

        break;


      case 4:

        if (edge_count == 0)
printf("Please input graph
first!\n");

        else show_edges(edges,
edge_count);

        break;


      case 5:

        printf("Exiting...\n");

        exit(0);


      default:

        printf("Invalid
choice!\n");

    }

  }


  return 0;

}
```

INPUT –

```
Enter choice: 1
Number of stages (k): 5
Number of vertices (n): 14
Enter edges (from to cost), -1 -1 -1 to finish:
Edge 1: 1 2 9
Edge 2: 1 3 8
Edge 3: 1 4 6
Edge 4: 1 5 7
Edge 5: 2 6 10
Edge 6: 2 7 11
Edge 7: 2 8 12
Edge 8: 3 6 15
Edge 9: 3 8 14
Edge 10: 3 9 10
Edge 11: 4 7 10
Edge 12: 4 8 11
Edge 13: 4 9 12
Edge 14: 5 6 8
Edge 15: 5 7 9
Edge 16: 5 9 10
Edge 17: 6 10 5
Edge 18: 6 11 6
Edge 19: 6 13 7
Edge 20: 7 10 8
Edge 21: 7 11 9
Edge 22: 7 12 7
Edge 23: 8 11 6
Edge 24: 8 12 7
Edge 25: 8 13 8
Edge 26: 9 10 4
Edge 27: 9 12 8
Edge 28: 9 13 11
Edge 29: 10 14 8
Edge 30: 11 14 9
Edge 31: 12 14 8
Edge 32: 13 14 7
Edge 33: -1 -1 -1
```

OUTPUT – I] Forward Approach

```
Menu:
1. Input Graph
2. Run Forward Approach
3. Run Backward Approach
4. Show Edges
5. Exit
Enter choice: 2
Show detailed calculations? (1=Yes, 0=No): 1

Forward Approach - Detailed Calculations:

--- STAGE 4 ---
Vertex 10:
fcost(4,10) = min{c(10,14)+fcost(5,14)=8+0=8} = 8
d(4,10) = 14

Vertex 11:
fcost(4,11) = min{c(11,14)+fcost(5,14)=9+0=9} = 9
d(4,11) = 14

Vertex 12:
fcost(4,12) = min{c(12,14)+fcost(5,14)=8+0=8} = 8
d(4,12) = 14

Vertex 13:
fcost(4,13) = min{c(13,14)+fcost(5,14)=7+0=7} = 7
d(4,13) = 14


--- STAGE 3 ---
Vertex 6:
fcost(3,6) = min{c(6,10)+fcost(4,10)=5+8=13, c(6,11)+fcost(4,11)=6+9=15, c(6,13)+fcost(4,13)=7+7=14} = 13
d(3,6) = 10

Vertex 7:
fcost(3,7) = min{c(7,10)+fcost(4,10)=8+8=16, c(7,11)+fcost(4,11)=9+9=18, c(7,12)+fcost(4,12)=7+8=15} = 15
d(3,7) = 12

Vertex 8:
fcost(3,8) = min{c(8,11)+fcost(4,11)=6+9=15, c(8,12)+fcost(4,12)=7+8=15, c(8,13)+fcost(4,13)=8+7=15} = 15
d(3,8) = 11

Vertex 9:
fcost(3,9) = min{c(9,10)+fcost(4,10)=4+8=12, c(9,12)+fcost(4,12)=8+8=16, c(9,13)+fcost(4,13)=11+7=18} = 12
d(3,9) = 10


--- STAGE 2 ---
Vertex 2:
fcost(2,2) = min{c(2,6)+fcost(3,6)=10+13=23, c(2,7)+fcost(3,7)=11+15=26, c(2,8)+fcost(3,8)=12+15=27} = 23
d(2,2) = 6

Vertex 3:
fcost(2,3) = min{c(3,6)+fcost(3,6)=15+13=28, c(3,8)+fcost(3,8)=14+15=29, c(3,9)+fcost(3,9)=10+12=22} = 22
d(2,3) = 9

Vertex 4:
fcost(2,4) = min{c(4,7)+fcost(3,7)=10+15=25, c(4,8)+fcost(3,8)=11+15=26, c(4,9)+fcost(3,9)=12+12=24} = 24
d(2,4) = 9

Vertex 5:
fcost(2,5) = min{c(5,6)+fcost(3,6)=8+13=21, c(5,7)+fcost(3,7)=9+15=24, c(5,9)+fcost(3,9)=10+12=22} = 21
d(2,5) = 6


--- STAGE 1 ---
Vertex 1:
fcost(1,1) = min{c(1,2)+fcost(2,2)=9+23=32, c(1,3)+fcost(2,3)=8+22=30, c(1,4)+fcost(2,4)=6+24=30, c(1,5)+fcost(2,5)=7+21=28} = 28
d(1,1) = 5
```

```
Forward Approach Results:
Minimum cost: 28
Shortest path: 1 5 6 10 14
Path details:
Stage 1 to 2: 1->5 (cost: 7)
Stage 2 to 3: 5->6 (cost: 8)
Stage 3 to 4: 6->10 (cost: 5)
Stage 4 to 5: 10->14 (cost: 8)
```

TIME TAKEN –

```
Execution time: 0.007000 seconds
```

II] BACKWARD APPROACH

```
Menu:
1. Input Graph
2. Run Forward Approach
3. Run Backward Approach
4. Show Edges
5. Exit
Enter choice: 3
Show detailed calculations? (1=Yes, 0=No): 1

Backward Approach - Detailed Calculations:

--- STAGE 2 ---
Vertex 2:
bcost(2,2) = min{bcost(1,1)+c(1,2)=0+9=9} = 9
d(2,2) = 1

Vertex 3:
bcost(2,3) = min{bcost(1,1)+c(1,3)=0+8=8} = 8
d(2,3) = 1

Vertex 4:
bcost(2,4) = min{bcost(1,1)+c(1,4)=0+6=6} = 6
d(2,4) = 1

Vertex 5:
bcost(2,5) = min{bcost(1,1)+c(1,5)=0+7=7} = 7
d(2,5) = 1


--- STAGE 3 ---
Vertex 6:
bcost(3,6) = min{bcost(2,2)+c(2,6)=9+10=19, bcost(2,3)+c(3,6)=8+15=23, bcost(2,5)+c(5,6)=7+8=15} = 15
d(3,6) = 5

Vertex 7:
bcost(3,7) = min{bcost(2,2)+c(2,7)=9+11=20, bcost(2,4)+c(4,7)=6+10=16, bcost(2,5)+c(5,7)=7+9=16} = 16
d(3,7) = 4

Vertex 8:
bcost(3,8) = min{bcost(2,2)+c(2,8)=9+12=21, bcost(2,3)+c(3,8)=8+14=22, bcost(2,4)+c(4,8)=6+11=17} = 17
d(3,8) = 4

Vertex 9:
bcost(3,9) = min{bcost(2,3)+c(3,9)=8+10=18, bcost(2,4)+c(4,9)=6+12=18, bcost(2,5)+c(5,9)=7+10=17} = 17
d(3,9) = 5


--- STAGE 4 ---
Vertex 10:
bcost(4,10) = min{bcost(3,6)+c(6,10)=15+5=20, bcost(3,7)+c(7,10)=16+8=24, bcost(3,9)+c(9,10)=17+4=21} = 20
d(4,10) = 6

Vertex 11:
bcost(4,11) = min{bcost(3,6)+c(6,11)=15+6=21, bcost(3,7)+c(7,11)=16+9=25, bcost(3,8)+c(8,11)=17+6=23} = 21
d(4,11) = 6

Vertex 12:
bcost(4,12) = min{bcost(3,7)+c(7,12)=16+7=23, bcost(3,8)+c(8,12)=17+7=24, bcost(3,9)+c(9,12)=17+8=25} = 23
d(4,12) = 7

Vertex 13:
bcost(4,13) = min{bcost(3,6)+c(6,13)=15+7=22, bcost(3,8)+c(8,13)=17+8=25, bcost(3,9)+c(9,13)=17+11=28} = 22
d(4,13) = 6


--- STAGE 5 ---
Vertex 14:
bcost(5,14) = min{bcost(4,10)+c(10,14)=20+8=28, bcost(4,11)+c(11,14)=21+9=30, bcost(4,12)+c(12,14)=23+8=31, bcost(4,13)+c(13,14)=22+7=29} = 28
d(5,14) = 10
```

```
Backward Approach Results:
Minimum cost: 28
Shortest path: 14 10 6 5 1
Path details:
Stage 4 to 5: 10->14 (cost: 8)
Stage 3 to 4: 6->10 (cost: 5)
Stage 2 to 3: 5->6 (cost: 8)
Stage 1 to 2: 1->5 (cost: 7)
```

TIME TAKEN

```
Execution time: 0.005000 seconds
```

CONCLUSION – Shortest path and minimum cost of the path of the given k stage graph was successfully calculated using forward and backward Multistage graph approach

Date –

ALL PAIRS SHORTEST PATH

AIM – Write a C program to calculate shortest path from a vertex to all the vertexes of a given graph using all paths algorithm

Problem statement – Given a cost adjacency matrix of a graph apply all paths algorithm to calculate shortest path from a vertex to all other vertex in the graph

Input – The cost adjacency matrix A is inputed

Output - Shortest path from a vertex to all vertexes of the graph .

ALGORITHM

Algorithm AllPaths(cost,A,n)

```
// cost[1:n,1:n] is the cost adjacency matrix of a graph with
// n vertices; A[i,j] is the cost of a shortest path from vertex
// i to vertex j. cost[i,i]=0.0, for 1≤i≤n.
{
  for i := 1 to n do
    for j := 1 to n do
      A[i,j] := cost[i,j]; // Copy cost into A.

  for k := 1 to n do
    for i := 1 to n do
      for j := 1 to n do
        A[i,j] := min(A[i,j], A[i,k] + A[k,j]);
}
```

Recurrence Relation

This algorithm uses **3 nested loops** over n, so:

$T(n)=O(n^3)$

Each A[i][j] is updated by checking all possible intermediate vertices k, leading to cubic time.

Time Complexity

I] Best Case: $O(n^3)$

- Even if no updates are needed (e.g., all shortest paths already known), all n^3 comparisons are still performed.
- Hence, no improvement in best-case time.

II] Average Case: $O(n^3)$

- For typical graphs, each triplet (i, j, k) is processed once.
- The algorithm doesn't adapt to graph sparsity and always performs cubic operations.

III] Worst Case: $O(n^3)$

- For large or dense graphs, each path update may occur, and all vertex combinations are checked.
- Still n^3 steps regardless of the graph's edge count.

Space Complexity

I] Best Case: $O(n^2)$

- Matrix A[n][n] is used to store shortest path costs.
- Even if the graph is sparse, the full matrix is maintained.

II] Average Case: $O(n^2)$

- Space does not vary based on edge density.
- The algorithm always stores a full n x n matrix.

III] Worst Case: $O(n^2)$

- Even with complete graphs, no extra space is needed beyond A[n][n].
- No recursion or per-path memory allocation beyond matrix.

PROGRAM –

```
#include <stdio.h>
#include <limits.h>
#include <time.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>

#define MAX 100
#define INF 99999
#define INFINITY INF

int p[MAX][MAX];

void printMatrixWithChanges(int n, int A[MAX][MAX], int prev[MAX][MAX], char* title) {
    printf("%s\n", title);

    printf("%-3s| ", title);
    for (int j = 1; j <= n; j++) {
        printf("%-3d ", j);
    }
    printf("\n");

    printf("---|");
    for (int j = 1; j <= n; j++) {
        printf("----");
    }
    printf("\n");

    for (int i = 1; i <= n; i++) {
        printf("%-3d| ", i);
        for (int j = 1; j <= n; j++) {
            if (A[i][j] >= INF) {
                printf("INF ");
            }
            else if (prev != NULL && A[i][j] != prev[i][j]) {
                printf("[%-2d] ", A[i][j]);
            }
            else {
                printf("%-3d ", A[i][j]);
            }
        }
        printf("\n");
    }
    printf("\n");
}

void printPMatrix(int n, int p[MAX][MAX], int k) {
    char title[10];
    sprintf(title, "P%d", k);

    printf("%s\n", title);

    printf("%-3s| ", title);
    for (int j = 1; j <= n; j++) {
        printf("%-3d ", j);
    }
    printf("\n");

    printf("---|");
    for (int j = 1; j <= n; j++) {
        printf("----");
    }
    printf("\n");

    for (int i = 1; i <= n; i++) {
        printf("%-3d| ", i);
        for (int j = 1; j <= n; j++) {
            if (p[i][j] == -1) {
                printf("-   ");
            } else {
                printf("%-3d ", p[i][j]);
            }
        }
        printf("\n");
    }
    printf("\n");
}

void allPaths(int n, int cost[MAX][MAX], int A[MAX][MAX], int p[MAX][MAX]) {
    int prev[MAX][MAX];

    clock_t start, end;
    double cpu_time_used;

    start = clock();

    // Initialize A^0 with the cost matrix
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            A[i][j] = cost[i][j];
            if (i == j) {
                p[i][j] = -1;
            } else if (cost[i][j] < INF) {
                p[i][j] = i;
            } else {
                p[i][j] = -1;
            }
        }
```

```
    }


    printMatrixWithChanges(n, A,
NULL, "A0");
    printPMatrix(n, p, 0);


    // Floyd-Warshall algorithm
    for (int k = 1; k <= n; k++) {
        // Save previous matrix for
highlighting changes
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= n; j++) {
                prev[i][j] = A[i][j];
            }
        }


        // Update distances
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= n; j++) {
                if (A[i][k] < INF && A[k][j]
< INF) {
                    int newDist = A[i][k] +
A[k][j];
                    if (newDist < A[i][j]) {
                        A[i][j] = newDist;
                        p[i][j] = p[k][j];
                    }
                }
            }
        }


        // Print matrix with changes
highlighted
        char title[10];
        sprintf(title, "A%d", k);
        printMatrixWithChanges(n, A,
prev, title);
        printPMatrix(n, p, k);
    }


    end = clock();
    cpu_time_used = ((double) (end
- start)) / CLOCKS_PER_SEC;
    printf("\nAllPaths algorithm
execution time: %f seconds\n",
cpu_time_used);
}



char *getPath(int p[MAX][MAX],
int i, int j) {
    if (i == j) {
        char *path = (char
*)malloc(10 * sizeof(char));
        sprintf(path, "%d", i);
        return path;
    }
    if (p[i][j] == -1) {
        char *path = (char
*)malloc(10 * sizeof(char));
        sprintf(path, "No path");
        return path;
    }
    char *intermediatePath =
getPath(p, i, p[i][j]);
    char *path = (char
*)malloc((strlen(intermediatePath
) + 10) * sizeof(char));
    sprintf(path, "%s->%d",
intermediatePath, j);
    free(intermediatePath);
    return path;
}


// Function definition for AllPair
algorithm
void AllPair(int cost[MAX][MAX],
int A[MAX][MAX], int
p[MAX][MAX], int n) {
    // Initialize p matrix
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            if (i == j || cost[i][j] == INF)
                p[i][j] = -1;
            else
                p[i][j] = i;
        }
    }


    // Floyd-Warshall algorithm
    for (int k = 1; k <= n; k++) {
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= n; j++) {
                if (A[i][k] < INF && A[k][j]
< INF && A[i][k] + A[k][j] < A[i][j]) {
                    A[i][j] = A[i][k] + A[k][j];
                    p[i][j] = p[k][j];
                }
            }
        }
    }
}


void runAllPair(int n, int
cost[MAX][MAX]) {
    int A[MAX][MAX],
p[MAX][MAX];


    clock_t start, end;
    double cpu_time_used;


    start = clock();
```

```
    // Initialize A with the cost matrix
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            A[i][j] = cost[i][j];
        }
    }

    AllPair(cost, A, p, n);

    end = clock();
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
    printf("\nAllPair algorithm execution time: %f seconds\n", cpu_time_used);
}

void inputGraph(int *n, int cost[MAX][MAX]) {
    printf("Enter number of vertices (n, max %d): ", MAX-1);
    scanf("%d", n);

    printf("\nEnter the cost matrix (%d for infinity):\n", INF);

    for (int i = 1; i <= *n; i++) {
        for (int j = 1; j <= *n; j++) {
            printf("cost[%d][%d]: ", i, j);
            scanf("%d", &cost[i][j]);
            if (i == j) cost[i][j] = 0;  // Diagonal is 0
        }
    }
}

void displayMenu() {
    printf("\n===== AllPaths Algorithm Menu =====\n");
    printf("1. Input Graph\n");
    printf("2. Run AllPaths Algorithm\n");
    printf("3. Exit\n");
    printf("Enter your choice: ");
}

void printHeader() {
    printf("**********************************************************\n");
    printf(" Roll number: 23B-CO-010\n");
    printf(" PR Number - 202311390\n");
    printf("**********************************************************\n\n");
    printf(" All-Pairs Shortest Path\n");
    printf("==========================================\n");
}

int main() {
    int n = 0;
    int cost[MAX][MAX];
    int A[MAX][MAX];
    int p[MAX][MAX];
    int choice;
    int graphEntered = 0;

    printHeader();

    do {
        displayMenu();
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                inputGraph(&n, cost);
                graphEntered = 1;
                break;
            case 2:
                if (graphEntered) {
                    printf("\nRunning AllPaths algorithm...\n\n");
                    allPaths(n, cost, A, p);

                    printf("Final Shortest Paths:\n");
                    printf("+---------+---------+---------+------------------+\n");
                    printf("| Source  | Dest    | Length  | Path             |\n");
                    printf("+---------+---------+---------+------------------+\n");
                    for (int i = 1; i <= n; i++) {
                        for (int j = 1; j <= n; j++) {
                            char *path = getPath(p, i, j);
                            if (A[i][j] == INFINITY)
                                printf("| %-7d | %-7d | %-7s | %-16s |\n", i, j, "∞", path);
                            else
                                printf("| %-7d | %-7d | %-7d | %-16s |\n", i, j, A[i][j], path);
                            free(path);
                        }
                    }
                    printf("+---------+---------+---------+------------------+\n");
```

```
            } else {
                printf("\nPlease input a graph first (Option 1).\n");
            }
            break;
        case 3:
            printf("\nExiting program.\n");
            break;
        default:
            printf("\nInvalid choice. Please try again.\n");
    }
} while (choice != 3);

return 0;
}
```

INPUT –

```
===== AllPaths Algorithm Menu =====
1. Input Graph
2. Run AllPaths Algorithm
3. Exit
Enter your choice: 1
Enter number of vertices (n, max 99): 5

Enter the cost matrix (99999 for infinity):
cost[1][1]: 0
cost[1][2]: 8
cost[1][3]: 3
cost[1][4]: 99999
cost[1][5]: -4
cost[2][1]: 2
cost[2][2]: 0
cost[2][3]: -2
cost[2][4]: 4
cost[2][5]: 7
cost[3][1]: 99999
cost[3][2]: 2
cost[3][3]: 0
cost[3][4]: 6
cost[3][5]: 99999
cost[4][1]: -2
cost[4][2]: 99999
cost[4][3]: -5
cost[4][4]: 0
cost[4][5]: -2
cost[5][1]: 6
cost[5][2]: 99999
cost[5][3]: 99999
cost[5][4]: 6
cost[5][5]: 0
```

OUTPUT –

```
Running AllPaths algorithm...

A0
A0 | 1   2   3   4   5
---|--------------------
1  | 0   8   3   INF -4
2  | 2   0   -2  4   7
3  | INF 2   0   6   INF
4  | -2  INF -5  0   -2
5  | 6   INF INF 6   0

P0
P0 | 1   2   3   4   5
---|--------------------
1  | -   1   1   -   1
2  | 2   -   2   2   2
3  | -   3   -   3   -
4  | 4   -   4   -   4
5  | 5   -   -   5   -

A1
A1 | 1   2   3   4   5
---|--------------------
1  | 0   8   3   INF -4
2  | 2   0   -2  4   [-2]
3  | INF 2   0   6   INF
4  | -2  [6 ] -5  0   [-6]
5  | 6   [14] [9 ] 6   0

P1
P1 | 1   2   3   4   5
---|--------------------
1  | -   1   1   -   1
2  | 2   -   2   2   1
3  | -   3   -   3   -
4  | 4   1   4   -   1
5  | 5   1   1   5   -

A2
A2 | 1   2   3   4   5
---|--------------------
1  | 0   8   3   [12] -4
2  | 2   0   -2  4   -2
3  | [4 ] 2   0   6   [0 ]
4  | -2  6   -5  0   -6
5  | 6   14  9   6   0

P2
P2 | 1   2   3   4   5
---|--------------------
1  | -   1   1   2   1
2  | 2   -   2   2   1
3  | 2   3   -   3   1
4  | 4   1   4   -   1
5  | 5   1   1   5   -

A3
A3 | 1   2   3   4   5
---|--------------------
1  | 0   [5 ] 3   [9 ] -4
2  | 2   0   -2  4   -2
3  | 4   2   0   6   0
4  | -2  [-3] -5  0   -6
5  | 6   [11] 9   6   0

P3
P3 | 1   2   3   4   5
---|--------------------
1  | -   3   1   3   1
2  | 2   -   2   2   1
3  | 2   3   -   3   1
4  | 4   3   4   -   1
5  | 5   3   1   5   -

A4
A4 | 1   2   3   4   5
---|--------------------
1  | 0   5   3   9   -4
2  | 2   0   -2  4   -2
3  | 4   2   0   6   0
4  | -2  -3  -5  0   -6
5  | [4 ] [3 ] [1 ] 6   0

P4
P4 | 1   2   3   4   5
---|--------------------
1  | -   3   1   3   1
2  | 2   -   2   2   1
3  | 2   3   -   3   1
4  | 4   3   4   -   1
5  | 4   3   4   5   -
```

```
A5
A5 | 1   2   3   4   5
---|--------------------
1  | 0   [-1] [-3] [2 ] -4
2  | 2   0   -2  4   -2
3  | 4   2   0   6   0
4  | -2  -3  -5  0   -6
5  | 4   3   1   6   0

P5
P5 | 1   2   3   4   5
---|--------------------
1  | -   3   4   5   1
2  | 2   -   2   2   1
3  | 2   3   -   3   1
4  | 4   3   4   -   1
5  | 4   3   4   5   -
```

```
AllPaths algorithm execution time: 0.045000 seconds
Final Shortest Paths:
+---------+--------+---------+------------------+
| Source  | Dest   | Length  | Path             |
+---------+--------+---------+------------------+
| 1       | 1      | 0       | 1                |
| 1       | 2      | -1      | 1->5->4->3->2    |
| 1       | 3      | -3      | 1->5->4->3       |
| 1       | 4      | 2       | 1->5->4          |
| 1       | 5      | -4      | 1->5             |
| 2       | 1      | 2       | 2->1             |
| 2       | 2      | 0       | 2                |
| 2       | 3      | -2      | 2->3             |
| 2       | 4      | 4       | 2->4             |
| 2       | 5      | -2      | 2->1->5          |
| 3       | 1      | 4       | 3->2->1          |
| 3       | 2      | 2       | 3->2             |
| 3       | 3      | 0       | 3                |
| 3       | 4      | 6       | 3->4             |
| 3       | 5      | 0       | 3->2->1->5       |
| 4       | 1      | -2      | 4->1             |
| 4       | 2      | -3      | 4->3->2          |
| 4       | 3      | -5      | 4->3             |
| 4       | 4      | 0       | 4                |
| 4       | 5      | -6      | 4->1->5          |
| 5       | 1      | 4       | 5->4->1          |
| 5       | 2      | 3       | 5->4->3->2       |
| 5       | 3      | 1       | 5->4->3          |
| 5       | 4      | 6       | 5->4             |
| 5       | 5      | 0       | 5                |
+---------+--------+---------+------------------+
```

TIME TAKEN –

```
AllPaths algorithm execution time: 0.016000 seconds
```

CONCLUSION – All paths algorithm was successfully implemented to calculate shortest distance from a vertex to all the vertexes in the graph .

Date –

BELLMAN FORD ALGORITHM

AIM – Write a C program to calculate shortest path from a source vertex to all the vertexes of a given graph using bellman ford algorithm

Problem statement – Given a cost adjacency matrix of a graph apply bellman ford algorithm to calculate shortest path from a given vertex to all other vertex in the graph

Input – The cost adjacency matrix A is inputed

Output - Shortest path from a given vertex to all vertexes of the graph .

ALGORITHM

Algorithm BellmanFord(v, cost, dist, n)

```
// Single-source/all-destinations shortest paths with negative edge costs
{
   for i := 1 to n do            // Initialize dist.
      dist[i] := cost[v, i];

   for k := 2 to n − 1 do         // Relax edges repeatedly
      for each u such that u ≠ v and u has at least one incoming edge do
         for each (i, u) in the graph do
            if dist[u] > dist[i] + cost[i, u] then
               dist[u] := dist[i] + cost[i, u];
}
```

Recurrence Relation

Let n be the number of vertices, and E be the number of edges.

The key operation (edge relaxation) is repeated n - 2 times, across all edges:

$T(n,E) = O(n \cdot E)$

Time Complexity

I] Best Case: O(E)

- If no updates are required after the first iteration (i.e., distances are already optimal), the algorithm can theoretically terminate early.
- But in this version, early stopping is **not implemented**, so we still go through all iterations → $O(n \cdot E)$ even in best case.
- *However*, in **practical optimizations**, early stopping could reduce best case to **O(E)**.

II] Average Case: O(n·E)

- On average, all edges are relaxed n-2 times.
- The outer loop runs n-2 times and the inner loop processes all edges, hence $O(n \cdot E)$.

III] Worst Case: O(n·E)

- In the worst case (e.g., paths improve with each iteration), all edges are relaxed for each of the n-2 passes.
- So, total operations: (n-2) * E → $O(n \cdot E)$

Space Complexity

I] Best Case: O(n)

- Only the dist[n] array is needed for tracking shortest distances.
- Edge list or matrix is assumed to be part of input; no extra space used.

II] Average Case: O(n)

- Space does not depend on edge count, only on number of vertices.
- No recursive calls or per-path memory.

III] Worst Case: O(n)

- Regardless of edge count or graph density, memory use for dist array stays the same.

PROGRAM –

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <limits.h>
#include <time.h>

#define N 100
#define INF 99999

void printPath(int parent[], int j);

void displayIterationDistances(int n, int dist[N][N]) {
   printf("\nDistance table for each iteration (k):\n");

   printf("%5s", "k");
   for (int i = 1; i <= n; i++) {
      printf("%4d", i);
   }
   printf("\n");

   for (int i = 0; i <= n; i++) printf("----");
   printf("\n");

   for (int k = 0; k <= n; k++) {
      printf("%4d |", k);
      for (int j = 1; j <= n; j++) {
         if (dist[k][j] == INF)
            printf("  INF ");
         else
            printf("%3d ", dist[k][j]);
      }
      printf("\n");
   }
}

void displayFinalDistances(int n, int dist[N], int parent[N], int source) {
   printf("\nFinal shortest distances from source vertex %d:\n", source);
   printf("Vertex\tDistance\tPath\n");

   for (int i = 1; i <= n; i++) {
      if (i != source) {
         printf("%d\t", i);

         if (dist[i] == INF)
            printf("∞\t\tUnreachable\n");
         else {
            printf("%d\t\t%d", dist[i], source);
            printPath(parent, i);
            printf("\n");
         }
      }
   }
}

void printPath(int parent[], int j) {
   if (parent[j] == -1)
      return;

   printPath(parent, parent[j]);
   printf(" -> %d", j);
}

void initializeGraph(int n, int graph[][N]) {
   for (int i = 1; i <= n; i++) {
      for (int j = 1; j <= n; j++) {
         if (i == j)
            graph[i][j] = 0;
         else
            graph[i][j] = INF;
      }
   }
}

void displayGraph(int n, int graph[][N]) {
   printf("\nAdjacency Matrix:\n");
   printf("   ");
   for (int i = 1; i <= n; i++)
      printf("%4d", i);
   printf("\n");

   for (int i = 1; i <= n; i++) {
      printf("%2d ", i);
      for (int j = 1; j <= n; j++) {
         if (graph[i][j] == INF)
            printf(" INF");
         else
            printf("%4d", graph[i][j]);
      }
      printf("\n");
   }
}
```

```
void inputGraph(int n, int
graph[][N]) {
   printf("\nEnter the cost matrix
(%d for infinity):\n", INF);

   for (int i = 1; i <= n; i++) {
      for (int j = 1; j <= n; j++) {
         if (i == j) {
            graph[i][j] = 0;
            continue;
         }
         printf("cost[%d][%d]: ", i, j);
         scanf("%d", &graph[i][j]);
         if (graph[i][j] == INF) {
            printf("  (infinity)\n");
         }
      }
   }
}

void runBellmanFord(int n, int
graph[][N], int source) {
   int dist[N][N];
   int finalDist[N];
   int parent[N];
   clock_t start, end;
   double cpu_time_used;
   bool has_negative_cycle = false;

   for (int i = 1; i <= n; i++)
      parent[i] = -1;

   start = clock();

   for (int i = 1; i <= n; i++) {
      if (i == source) {
         dist[0][i] = 0;
      } else {
         dist[0][i] = graph[source][i];
         if (graph[source][i] != INF)
            parent[i] = source;
      }
   }

   for (int k = 1; k <= n; k++) {

      for (int i = 1; i <= n; i++)
         dist[k][i] = dist[k-1][i];

      // Relax edges
      for (int v = 1; v <= n; v++) {
         if (v != source) {
            for (int u = 1; u <= n; u++)
{
               if (graph[u][v] != INF
&& dist[k-1][u] != INF &&
                  dist[k-1][u] +
graph[u][v] < dist[k][v]) {
                  dist[k][v] = dist[k-
1][u] + graph[u][v];
                  parent[v] = u;
               }
            }
         }
      }
   }

   for (int i = 1; i <= n; i++)
      finalDist[i] = dist[n][i];

   for (int u = 1; u <= n; u++) {
      for (int v = 1; v <= n; v++) {
         if (graph[u][v] != INF &&
dist[n][u] != INF &&
            dist[n][u] + graph[u][v] <
dist[n][v]) {
            has_negative_cycle =
true;
            break;
         }
      }
      if (has_negative_cycle) break;
   }

   end = clock();
   cpu_time_used = ((double) (end
- start)) / CLOCKS_PER_SEC;

   printf("\nExecution time: %.6f
seconds\n", cpu_time_used);

   displayIterationDistances(n,
dist);

   if (has_negative_cycle) {
      printf("\nWARNING: Graph
contains negative weight
cycle!\n");
   } else {

      displayFinalDistances(n,
finalDist, parent, source);
   }
}
```

```
int main() {

    printf("**********
**********
*******\n");

  printf(" Roll number: 23B-CO-
010\n");

  printf(" PR Number -
202311390\n");

  printf("**********
**********
*****\n\n");

  int choice, n, source;

  int graph[N][N];

  bool graphCreated = false;



  while (1) {

    printf("\n===== Bellman Ford
Algorithm Menu =====\n");

    printf("1. Create Graph\n");

    printf("2. Run Bellman-Ford
Algorithm\n");

    printf("3. Display Graph\n");

    printf("4. Exit\n");

    printf("Enter your choice: ");

    scanf("%d", &choice);



    switch (choice) {

      case 1:

        printf("Enter number of
vertices (max %d): ", N-1);

        scanf("%d", &n);

        if (n <= 0 || n >= N) {

          printf("Invalid number
of vertices!\n");

          break;

        }



        initializeGraph(n, graph);

        inputGraph(n, graph);

        graphCreated = true;

        break;



      case 2:

        if (!graphCreated) {

          printf("Please create a
graph first!\n");

          break;

        }



        printf("Enter source
vertex (1 to %d): ", n);

        scanf("%d", &source);



        if (source < 1 || source >
n) {

          printf("Invalid source
vertex!\n");

          break;

        }

        runBellmanFord(n, graph,
source);

        break;



      case 3:

        if (!graphCreated) {

          printf("Please create a
graph first!\n");

          break;

        }



        displayGraph(n, graph);

        break;



      case 4:

        printf("Exiting
program...\n");

        return 0;



      default:

        printf("Invalid choice!
Please try again.\n");

    }

  }



  return 0;

}
```

INPUT –

```
*********************************************************
 Roll number: 23B-CO-010
 PR Number - 202311390
*********************************************************


===== Bellman Ford Algorithm Menu =====
1. Create Graph
2. Run Bellman-Ford Algorithm
3. Display Graph
4. Exit
Enter your choice: 1
Enter number of vertices (max 99): 7

Enter the cost matrix (99999 for infinity):
cost[1][2]: 6
cost[1][3]: 5
cost[1][4]: 5
cost[1][5]: 99999
  (infinity)
cost[1][6]: 99999
  (infinity)
cost[1][7]: 99999
  (infinity)
cost[2][1]: 99999
  (infinity)
cost[2][3]: 3
cost[2][4]: 99999
  (infinity)
cost[2][5]: -1
cost[2][6]: 2
cost[2][7]: 3
cost[3][1]: 99999
  (infinity)
cost[3][2]: -2
cost[3][4]: 99999
  (infinity)
cost[3][5]: 1
cost[3][6]: 99999
  (infinity)
cost[3][7]: 99999
  (infinity)
cost[4][1]: -2
cost[4][2]: 3
cost[4][3]: -2
cost[4][5]: 99999
  (infinity)
cost[4][6]: -1
cost[4][7]: 99999
  (infinity)
cost[5][1]: 1
cost[5][2]: 2
cost[5][3]: 99999
  (infinity)
cost[5][4]: 99999
  (infinity)
cost[5][6]: 5
cost[5][7]: 3
cost[6][1]: 4
cost[6][2]: 1
cost[6][3]: 99999
  (infinity)
cost[6][4]: 99999
  (infinity)
cost[6][5]: 99999
  (infinity)
cost[6][7]: 3
cost[7][1]: -2
cost[7][2]: -1
cost[7][3]: 3
cost[7][4]: 99999
  (infinity)
cost[7][5]: 99999
  (infinity)
cost[7][6]: -2
```

OUTPUT –

```
Distance table for each iteration (k):
   k   1   2   3   4   5   6   7
--------------------------------
   0 |  0   6   5   5  INF  INF  INF
   1 |  0   3   3   5   5   4   9
   2 |  0   1   3   5   2   4   6
   3 |  0   1   3   5   0   3   4
   4 |  0   1   3   5   0   2   3
   5 |  0   1   3   5   0   1   3
   6 |  0   1   3   5   0   1   3
   7 |  0   1   3   5   0   1   3

Final shortest distances from source vertex 1:
Vertex  Distance        Path
2       1               1 -> 4 -> 3 -> 2
3       3               1 -> 4 -> 3
4       5               1 -> 4
5       0               1 -> 4 -> 3 -> 2 -> 5
6       1               1 -> 4 -> 3 -> 2 -> 5 -> 7 -> 6
7       3               1 -> 4 -> 3 -> 2 -> 5 -> 7
```

TIME TAKEN –

```
Execution time: 0.000000 seconds
```

CONCLUSION – Bellman ford algorithm was successfully implemented to calculate the minimum cost shortest path from a source vertex to all the given vertices of the graph .

Date –

OPTIMAL BINARY SEARCH TREE

AIM – Write a C program to generate optimal binary search tree for a set of item whose probabilities for successful and unsuccessful search are provided

Problem statement – Given a set of n items whose probability set of successful and unsuccessful search is provided ,generate a binary search tree

Input – $[a_1 \ldots a_5]$ = { auto,case ,extern,static,void }

$[p_1 \ldots\ldots p_5]$ = { 6,7,4,5,6}

$[q_0 \ldots . q_5]$ = { 4,3,7,5,4,3}

Output - Generate binary search tree .

ALGORITHM

I] Algorithm Find(c, r, i, j)

```
{
   min := ∞;
   for m := r[i,j-1] to r[i+1,j] do {
      if (c[i,m-1] + c[m,j]) < min then {
         min := c[i,m-1] + c[m,j];
         l := m;
      }
   }
   return l;
}
```

Recurrence Relation

Let r[i, j] and c[i, j] be DP matrices. The function loops from r[i, j-1] to r[i+1, j.

Let s = r[i+1, j] - r[i, j-1] + 1

Then,

$T(i,j)=O(s)$(in worst case, s can be up to $j-i+1$)

In the standard OBST without optimization, this would be:

$T(i,j)=O(j-i+1)T(i, j) = O(j - i + 1)T(i,j)=O(j-i+1)$

Time Complexity

I] Best Case: O(1)

- If r[i, j-1] == r[i+1, j], only one iteration of the loop runs.
- Only one possible root to consider, so function returns in constant time.

II] Average Case: O(log n) to O(n)

- Thanks to Knuth's optimization, the number of root candidates shrinks.
- For large n, average number of candidates per call is sublinear.

III] Worst Case: O(n)

- Without optimization, the loop runs from i to j, i.e., up to n iterations.
- Even with optimization, worst-case gap between r[i,j-1] and r[i+1,j] can be large.

Space Complexity

I] Best Case: O(1)

- The function only uses a few scalar variables (min, l, m).
- No additional storage beyond the call.

II] Average Case: O(1)

- Space doesn't scale with input size; function modifies no large structures.
- It uses input matrices but doesn't allocate new memory.

III] Worst Case: O(1)

- Still constant space usage, regardless of how many iterations the loop executes.

```
Algorithm OBST(p, q, n)
// Computes optimal binary search tree for identifiers a_1 < ... < a_n
// p[i]: probabilities for identifiers (1 ≤ i ≤ n)
// q[i]: probabilities for dummy identifiers (0 ≤ i ≤ n)
// Outputs cost c[i,j], weight w[i,j], and root r[i,j] for subtrees
{
   // Initialization
   for i := 0 to n-1 do {
      w[i,i] := q[i];
      r[i,i] := 0;
      c[i,i] := 0.0;

      // Trees with one node
      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];
   }
   w[n,n] := q[n];
   r[n,n] := 0;
   c[n,n] := 0.0;

   for m := 2 to n do {
      for i := 0 to n-m do {
         j := i + m;
         w[i,j] := w[i,j-1] + p[j] + q[j];
         k := Find(c, r, i, j);
         c[i,j] := w[i,j] + c[i,k-1] + c[k,j];
         r[i,j] := k;
      }
}
   return (c[0,n], w[0,n], r[0,n]);}
```

Recurrence Relation

Let c[i][j] be the minimum cost of a subtree spanning a_(i+1) to a_j

$$c[i][j] = \min_{k \in [r[i,j-1], r[i+1,j]]} (c[i][k-1] + c[k][j] + w[i][j])$$

Where:

- w[i][j] is the sum of probabilities:

$$w[i][j] = \sum_{t=i+1}^{j} p[t] + \sum_{t=i}^{j} q[t]$$

TIME COMPLEXITY ANALYSIS

I] Best Case: $O(n^2)$

- Knuth's optimization ensures Find() checks only a narrow range [r[i,j−1],r[i+1,j]][r[i,j-1], r[i+1,j]][r[i,j−1],r[i+1,j]] instead of the full range.
- k is quickly found — leading to fast iteration over each subproblem.
- Matrix traversal still requires $O(n^2)$ operations, even if root selection is very fast.

II] Average Case: $O(n^2)$

- On average, the root selection in Find() doesn't degenerate to full search
- Every (i,j) pair is visited once and handled in nearly constant amortized time.
- Hence, overall runtime remains $O(n^2)$.

III] Worst Case: $O(n^2)$

- Even if root ranges expand slightly, the optimization still prevents full $O(n^3)$ behavior.
- Number of operations per entry remains bounded.
- Therefore, the worst-case time complexity is **$O(n^2)$**.

SPACE COMPLEXITY ANALYSIS

I] Best Case: $O(n^2)$

- Arrays c[i][j], w[i][j], and r[i][j] of size (n+1) × (n+1) are always initialized.
- No reduction in memory even for best-case inputs.

II] Average Case: $O(n^2)$

- Regardless of probability distribution, complete matrices are filled to support DP computations.

III] Worst Case: $O(n^2)$

- In all cases, full matrices are required and used.
- Memory use does not depend on input complexity.

PROGRAM –

```
#include <stdio.h>
#include <stdlib.h>
#include <float.h>
#include <string.h>
#include <time.h>

#define MAX_KEYS 100
#define MAX_HEIGHT 100
#define MAX_WIDTH 255

struct node {
   int key;
   char* identifier;
   struct node *left, *right;
};

char treePrint[MAX_HEIGHT][MAX_WIDTH];
char* identifiers[MAX_KEYS];
clock_t start_time, end_time;

void startTimer() {
   start_time = clock();
}

void stopTimer() {
   end_time = clock();
   double time_us = ((double)(end_time - start_time) * 1000000.0) / CLOCKS_PER_SEC;
   printf("\nTime taken: %.0fµs\n", time_us);
}

void clearTreePrint() {
   for (int i = 0; i < MAX_HEIGHT; i++)
      for (int j = 0; j < MAX_WIDTH; j++)
         treePrint[i][j] = ' ';
}

void drawTree(struct node *root, int row, int col, int spacing) {
   if (root == NULL) return;

   char buf[50];
   sprintf(buf, "%s(%d)", root->identifier, root->key);
   int len = strlen(buf);
   for (int i = 0; i < len; i++)
      treePrint[row][col + i] = buf[i];

   if (root->left) {
      int i;
      for (i = 1; i < spacing; i++) {
         treePrint[row + i][col - i] = '/';
      }
      drawTree(root->left, row + spacing, col - spacing * 2, spacing);
   }

   if (root->right) {
      int i;
      for (i = 1; i < spacing; i++) {
         treePrint[row + i][col + len + i - 1] = '\\';
      }
      drawTree(root->right, row + spacing, col + len + spacing, spacing);
   }
}

void displayOBSTTopDown(struct node *root) {
   clearTreePrint();
   drawTree(root, 0, MAX_WIDTH / 2, 3);

   for (int i = 0; i < MAX_HEIGHT; i++) {
      int line_has_char = 0;
      for (int j = 0; j < MAX_WIDTH; j++) {
         if (treePrint[i][j] != ' ') line_has_char = 1;
      }
      if (line_has_char) {
         for (int j = 0; j < MAX_WIDTH; j++)
            putchar(treePrint[i][j]);
         putchar('\n');
      }
   }
}

struct node* buildTree(int r[][MAX_KEYS], int i, int j) {
   if (i >= j || r[i][j] == 0) return NULL;

   struct node* newNode = (struct node*)malloc(sizeof(struct node));
   int rootKey = r[i][j];
   newNode->key = rootKey;
   newNode->identifier = identifiers[rootKey];
```

```
    newNode->left = buildTree(r, i, rootKey - 1);

    newNode->right = buildTree(r, rootKey, j);


    return newNode;

}


// Function to display detailed step-by-step calculations

void displayStepByStepCalculations(float p[], float q[], int n, float w[][MAX_KEYS], float c[][MAX_KEYS], int r[][MAX_KEYS]) {

    printf("\n----- DETAILED STEP-BY-STEP CALCULATIONS -----\n");


    // Initialize base cases

    printf("\n--- Base Cases Initialization ---\n");

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

        printf("w(%d,%d) = q(%d) = %.0f\n", i, i, i, q[i]);

        printf("c(%d,%d) = 0\n", i, i);

        printf("r(%d,%d) = 0\n", i, i);

    }


    // Calculate for length 1

    printf("\n--- Length 1 Calculations ---\n");

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

        int j = i + 1;

        printf("w(%d,%d) = p(%d) + q(%d) + w(%d,%d) = %.0f\n", i, j, j, j, i, i, w[i][j]);

        printf("c(%d,%d) = w(%d,%d) + min{c(%d,%d) + c(%d,%d)} = %.0f\n", i, j, i, j, i, j-1, j, j, c[i][j]);

        printf("r(%d,%d) = %d\n", i, j, r[i][j]);

    }


    // Calculate for lengths 2 and greater

    for (int m = 2; m <= n; m++) {

        printf("\n--- Length %d Calculations ---\n", m);

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

            int j = i + m;

            printf("w(%d,%d) = p(%d) + q(%d) + w(%d,%d) = %.0f\n", i, j, j, j, i, j-1, w[i][j]);


            printf("c(%d,%d) = w(%d,%d) + min {", i, j, i, j);


            // Display all potential k values and their costs

            for (int k = r[i][j-1]; k <= r[i+1][j]; k++) {

                float cost = c[i][k-1] + c[k][j];

                printf("c(%d,%d) + c(%d,%d) = %.0f", i, k-1, k, j, cost);

                if (k < r[i+1][j])

                    printf(", ");

            }


            printf("} = %.0f\n", c[i][j] - w[i][j]);

            printf("Total c(%d,%d) = %.0f\n", i, j, c[i][j]);

            printf("r(%d,%d) = %d\n", i, j, r[i][j]);

        }

    }

}


void calculateOBST(float p[], float q[], int n, float w[][MAX_KEYS], float c[][MAX_KEYS], int r[][MAX_KEYS]) {

    int i, j, k, m;

    float t;


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

        w[i][i] = q[i];

        c[i][i] = 0;

        r[i][i] = 0;

    }


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

        j = i + 1;

        w[i][j] = q[i] + p[j] + q[j];

        c[i][j] = q[i] + p[j] + q[j];

        r[i][j] = j;

    }


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

    printf("| i       | w[i][i] | c[i][i] | r[i][i] |\n");

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

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

        printf("| %d       | %.0f      | %.0f      | %d      |\n", i, w[i][i], c[i][i], r[i][i]);

    }

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


    for (m = 2; m <= n; m++) {

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

            j = i + m;

            w[i][j] = w[i][j-1] + p[j] + q[j];
```

```
        float min_cost = FLT_MAX;
        int min_k = i + 1;

        for (k = r[i][j-1]; k <= r[i+1][j]; k++) {
            t = c[i][k-1] + c[k][j];
            if (t < min_cost) {
                min_cost = t;
                min_k = k;
            }
        }

        c[i][j] = w[i][j] + min_cost;
        r[i][j] = min_k;
    }
  }

  printf("\n+-----------");
  for (int i = 0; i <= n; i++) {
    printf("+-----------");
  }
  printf("+\n|          ");
  for (int i = 0; i <= n; i++) {
    printf("| %-9d ", i);
  }
  printf("|\n+-----------");
  for (int i = 0; i <= n; i++) {
    printf("+-----------");
  }
  printf("+\n");

  for (int d = 0; d <= n; d++) {
    printf("| j-i=%-5d ", d);
    j = 0;
    printf("| w:%6.0f   ", w[j][d]);
    j++;
    for (int i = d + 1; i <= n; i++) {
      printf("| w:%6.0f   ", w[j][i]);
      j++;
    }
    printf("|\n|          ");
    j = 0;
    printf("| c:%6.0f   ", c[j][d]);
    j++;
    for (int i = d + 1; i <= n; i++) {
      printf("| c:%6.0f   ", c[j][i]);
      j++;
    }
    printf("|\n|          ");
    j = 0;
    printf("| r:%6d   ", r[j][d]);
    j++;
    for (int i = d + 1; i <= n; i++) {
      printf("| r:%6d   ", r[j][i]);
      j++;
    }
    printf("|\n+-----------");
    for (int i = 0; i <= n; i++) {
      printf("+-----------");
    }
    printf("+\n");
  }

  printf("\nCost of Optimal Binary Search Tree: %.0f\n", c[0][n]);
  printf("\nRoot of Optimal Binary Search Tree: %d\n", r[0][n]);
}

void freeTree(struct node* root) {
  if (root == NULL) return;

  freeTree(root->left);
  freeTree(root->right);
  free(root);
}

int main() {
  int n;
  float p[MAX_KEYS];
  float q[MAX_KEYS];
  float w[MAX_KEYS][MAX_KEYS];
  float c[MAX_KEYS][MAX_KEYS];
  int r[MAX_KEYS][MAX_KEYS];
  struct node* root = NULL;
  int choice;

  printf("**********************************************************************\n");
  printf(" Roll number: 23B-CO-010\n");
  printf(" PR Number - 202311390\n");
  printf("**********************************************************************\n\n");

  while(1) {
    printf("\n----- OPTIMAL BINARY SEARCH TREE MENU -----\n");
    printf("1. Create a new OBST\n");
    printf("2. Display tables\n");
    printf("3. Display tree visualization\n");
    printf("4. Display detailed step-by-step calculations\n");
    printf("5. Exit\n");
```

```
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch(choice) {
            case 1:
                printf("\nEnter the number of keys: ");
                scanf("%d", &n);

                printf("Enter the keys:\n");
                for (int i = 1; i <= n; i++) {
                    identifiers[i] = (char*)malloc(MAX_KEYS * sizeof(char));
                    printf("Key %d: ", i);
                    scanf("%s", identifiers[i]);
                }

                printf("Enter the probability of successful search:\n");
                for (int i = 1; i <= n; i++) {
                    printf("p[%d] = ", i);
                    scanf("%f", &p[i]);
                }

                printf("Enter the probability of unsuccessful search:\n");
                for (int i = 0; i <= n; i++) {
                    printf("q[%d] = ", i);
                    scanf("%f", &q[i]);
                }

                p[0] = 0.0;

                startTimer();
                calculateOBST(p, q, n, w, c, r);
                stopTimer();

                if (root != NULL) {
                    freeTree(root);
                }
                root = buildTree(r, 0, n);
                break;

            case 2:
                if (root == NULL) {
                    printf("\nPlease create an OBST first.\n");
                } else {
                    printf("\n----- OPTIMAL BST TABLES -----\n");
                    calculateOBST(p, q, n, w, c, r);
                }
                break;

            case 3:
                if (root == NULL) {
                    printf("\nPlease create an OBST first.\n");
                } else {
                    printf("\n----- GRAPHICAL REPRESENTATION OF OPTIMAL BST -----\n");
                    displayOBSTTopDown(root);
                }
                break;

            case 4:
                if (root == NULL) {
                    printf("\nPlease create an OBST first.\n");
                } else {
                    displayStepByStepCalculations(p, q, n, w, c, r);
                }
                break;

            case 5:
                if (root != NULL) {
                    freeTree(root);
                }
                for (int i = 1; i <= n; i++) {
                    if (identifiers[i] != NULL) {
                        free(identifiers[i]);
                    }
                }
                printf("\nThank you for using OBST program!\n");
                return 0;

            default:
                printf("\nInvalid choice. Please try again.\n");
        }
    }

    return 0;
}
```

INPUT –

```
**************************************************
 Roll number: 23B-CO-010
 PR Number - 202311390
**************************************************


----- OPTIMAL BINARY SEARCH TREE MENU -----
1. Create a new OBST
2. Display tables
3. Display tree visualization
4. Display detailed step-by-step calculations
5. Exit
Enter your choice: 1

Enter the number of keys: 5
Enter the keys:
Key 1: AUTO
Key 2: CASE
Key 3: EXTERN
Key 4: STATIC
Key 5: VOID
Enter the probability of successful search:
p[1] = 6
p[2] = 7
p[3] = 4
p[4] = 5
p[5] = 6
Enter the probability of unsuccessful search:
q[0] = 4
q[1] = 3
q[2] = 7
q[3] = 5
q[4] = 4
q[5] = 3
```

OUTPUT –

```
----- OPTIMAL BINARY SEARCH TREE MENU -----
1. Create a new OBST
2. Display tables
3. Display tree visualization
4. Display detailed step-by-step calculations
5. Exit
Enter your choice: 4

----- DETAILED STEP-BY-STEP CALCULATIONS -----

--- Base Cases Initialization ---
w(0,0) = q(0) = 4
c(0,0) = 0
r(0,0) = 0
w(1,1) = q(1) = 3
c(1,1) = 0
r(1,1) = 0
w(2,2) = q(2) = 7
c(2,2) = 0
r(2,2) = 0
w(3,3) = q(3) = 5
c(3,3) = 0
r(3,3) = 0
w(4,4) = q(4) = 4
c(4,4) = 0
r(4,4) = 0
w(5,5) = q(5) = 3
c(5,5) = 0
r(5,5) = 0

--- Length 1 Calculations ---
w(0,1) = p(1) + q(1) + w(0,0) = 13
c(0,1) = w(0,1) + min{c(0,0) + c(1,1)} = 13
r(0,1) = 1
w(1,2) = p(2) + q(2) + w(1,1) = 17
c(1,2) = w(1,2) + min{c(1,1) + c(2,2)} = 17
r(1,2) = 2
w(2,3) = p(3) + q(3) + w(2,2) = 16
c(2,3) = w(2,3) + min{c(2,2) + c(3,3)} = 16
r(2,3) = 3
w(3,4) = p(4) + q(4) + w(3,3) = 14
c(3,4) = w(3,4) + min{c(3,3) + c(4,4)} = 14
r(3,4) = 4
w(4,5) = p(5) + q(5) + w(4,4) = 13
c(4,5) = w(4,5) + min{c(4,4) + c(5,5)} = 13
r(4,5) = 5

--- Length 2 Calculations ---
w(0,2) = p(2) + q(2) + w(0,1) = 27
c(0,2) = w(0,2) + min {c(0,0) + c(1,2) = 17, c(0,1) + c(2,2) = 13} = 13
Total c(0,2) = 40
r(0,2) = 2
w(1,3) = p(3) + q(3) + w(1,2) = 26
c(1,3) = w(1,3) + min {c(1,1) + c(2,3) = 16, c(1,2) + c(3,3) = 17} = 16
Total c(1,3) = 42
r(1,3) = 2
w(2,4) = p(4) + q(4) + w(2,3) = 25
c(2,4) = w(2,4) + min {c(2,2) + c(3,4) = 14, c(2,3) + c(4,4) = 16} = 14
Total c(2,4) = 39
r(2,4) = 3
w(3,5) = p(5) + q(5) + w(3,4) = 23
c(3,5) = w(3,5) + min {c(3,3) + c(4,5) = 13, c(3,4) + c(5,5) = 14} = 13
Total c(3,5) = 36
r(3,5) = 4
```

```
--- Length 3 Calculations ---
w(0,3) = p(3) + q(3) + w(0,2) = 36
c(0,3) = w(0,3) + min {c(0,1) + c(2,3) = 29} = 29
Total c(0,3) = 65
r(0,3) = 2
w(1,4) = p(4) + q(4) + w(1,3) = 35
c(1,4) = w(1,4) + min {c(1,1) + c(2,4) = 39, c(1,2) + c(3,4) = 31} = 31
Total c(1,4) = 66
r(1,4) = 3
w(2,5) = p(5) + q(5) + w(2,4) = 34
c(2,5) = w(2,5) + min {c(2,2) + c(3,5) = 36, c(2,3) + c(4,5) = 29} = 29
Total c(2,5) = 63
r(2,5) = 4

--- Length 4 Calculations ---
w(0,4) = p(4) + q(4) + w(0,3) = 45
c(0,4) = w(0,4) + min {c(0,1) + c(2,4) = 52, c(0,2) + c(3,4) = 54} = 52
Total c(0,4) = 97
r(0,4) = 2
w(1,5) = p(5) + q(5) + w(1,4) = 44
c(1,5) = w(1,5) + min {c(1,2) + c(3,5) = 53, c(1,3) + c(4,5) = 55} = 53
Total c(1,5) = 97
r(1,5) = 3

--- Length 5 Calculations ---
w(0,5) = p(5) + q(5) + w(0,4) = 54
c(0,5) = w(0,5) + min {c(0,1) + c(2,5) = 76, c(0,2) + c(3,5) = 76} = 76
Total c(0,5) = 130
r(0,5) = 2
```

----- OPTIMAL BST TABLES -----

i	w[i][i]	c[i][i]	r[i][i]
0	4	0	0
1	3	0	0
2	7	0	0
3	5	0	0
4	4	0	0
5	3	0	0

	0	1	2	3	4	5
j-i=0	w: 4 c: 0 r: 0	w: 3 c: 0 r: 0	w: 7 c: 0 r: 0	w: 5 c: 0 r: 0	w: 4 c: 0 r: 0	w: 3 c: 0 r: 0
j-i=1	w: 13 c: 13 r: 1	w: 17 c: 17 r: 2	w: 16 c: 16 r: 3	w: 14 c: 14 r: 4	w: 13 c: 13 r: 5	
j-i=2	w: 27 c: 48 r: 2	w: 26 c: 42 r: 2	w: 25 c: 39 r: 3	w: 23 c: 36 r: 4		
j-i=3	w: 36 c: 65 r: 2	w: 35 c: 66 r: 3	w: 34 c: 63 r: 4			
j-i=4	w: 45 c: 97 r: 2	w: 44 c: 97 r: 3				
j-i=5	w: 54 c: 130 r: 2					

```
Cost of Optimal Binary Search Tree: 130

Root of Optimal Binary Search Tree: 2
```

```
----- OPTIMAL BINARY SEARCH TREE MENU -----
1. Create a new OBST
2. Display tables
3. Display tree visualization
4. Display detailed step-by-step calculations
5. Exit
Enter your choice: 3

----- GRAPHICAL REPRESENTATION OF OPTIMAL BST -----
```

CONCLUSION – OBST algorithm was successfully implemented to generate optimal binary search tree over the set of sets based on successful and unsuccessful searches.

Date –

Date -

0/1 Knapsack Problem

AIM – Write a C program to solve 0/1 knapsack problem using dynamic programming approach

Problem statement – Consider we are given n objects and knapsack capacity of 'm' ,each object i has weight w_i and profit p_i .The objective is to fill the knapsack that maximizes the profits and since the capacity is m ,the toal weight must be less than or equal to m .

Input – $[p_1.....p_7]$ = { 2,5,6,3,5,4,3}

$[w_1....w_7]$ = { 5,4,2,5,3,2,4}

Output - Generate the list of items that give maximum profit and has the total weight within the knapsack capacity .

ALGORITHM

Algorithm DKnap(p, w, x, n, m)

```
// Solves 0/1 knapsack problem using dynamic programming
// p[1..n]: profit values
// w[1..n]: weight values
// x[1..n]: solution vector (0 or 1)
// m: knapsack capacity
{
   // Initialize
   b[0] := 1;
   pair[1].p := pair[1].w := 0.0;  // S^o
   t := 1; h := 1;              // Start and end of S^o
   b[1] := next := 2;           // Next free spot in pair[]

   for i := 1 to n-1 do {       // Generate S^i
      k := t;
      u := Largest(pair, w, t, h, i, m);

      for j := t to u do {      // Generate S_1^(i-1) and merge
         pp := pair[j].p + p[i];
```

```
        ww := pair[j].w + w[i];  // (pp,ww) is next element in S1^(i-1)

        while (k ≤ h) and (pair[k].w ≤ ww) do {
            pair[next].p := pair[k].p;
            pair[next].w := pair[k].w;
            next := next + 1;
            k := k + 1;
        }

        if (k ≤ h) and (pair[k].w = ww) then {
            if pp < pair[k].p then pp := pair[k].p;
            k := k + 1;
        }

        if pp > pair[next-1].p then {
            pair[next].p := pp;
            pair[next].w := ww;
            next := next + 1;
        }

        while (k ≤ h) and (pair[k].p ≤ pair[next-1].p) do
            k := k + 1;
    }
    while k ≤ h do {
        pair[next].p := pair[k].p;
        pair[next].w := pair[k].w;
        next := next + 1;
        k := k + 1;    }
    t := h + 1;
    h := next - 1;
    b[i+1] := next;}
TraceBack(p, w, pair, x, m, n);}
```

Recurrence Relation:

This algorithm constructs efficient sets Si of non-dominated (profit, weight) pairs.

- No strict table-based recurrence, but roughly:

$T(n,m)=T(n-1,m)+O(k)$

where k is the size of intermediate pair sets, possibly up to O(m) in worst case.

Time Complexity:

I] **Best Case:** O(n log n)

- Only a few non-dominated pairs are generated at each step.
- Pruning removes dominated pairs effectively.
- Merging step becomes faster due to smaller set sizes.

II] **Average Case:** $O(n^2)$

- Number of pairs grows moderately with each item.
- Partial pruning reduces combinations, but not significantly.
- Each set merge and domination check takes increasing time.

III] **Worst Case:** O(nm)

- Maximum number of non-dominated pairs is close to m per stage.
- Pruning fails to reduce pair count significantly.
- Essentially a pseudo-polynomial time similar to classical 0/1 knapsack.

Space Complexity:

I] **Best Case:** O(n)

- At each step, very few state pairs are stored due to aggressive pruning.
- Efficient memory use with minimal storage.

II] **Average Case:** O(n log n) to $O(n^2)$

- Moderate pair growth, depends on profit-weight distribution.
- Storage increases as new states accumulate per item.

III] **Worst Case:** O(nm)

- No pruning occurs; all possible profit-weight states are stored.
- Full table of pairs similar to standard DP.

PROGRAM –

```
#include <stdio.h>

#include <stdlib.h>

#include <time.h>


#define MAX_ITEMS 100

#define MAX_CAPACITY 1000


typedef struct {

   float p;  // profit

   float w;  // weight

} PW;


int b[MAX_ITEMS+1];


clock_t start_timer() {

   return clock();

}


double end_timer(clock_t
start_time) {

   clock_t end_time = clock();

   return ((double)(end_time -
start_time)) / CLOCKS_PER_SEC *
1000; // in milliseconds

}

float dpknap(float p[], float w[],
int n, int m, PW pair[], int x[]) {

   int i, j, k, h, next;

   float pp, ww;


   h = 0;

   pair[0].p = 0;

   pair[0].w = 0;


   b[0] = 0;

   b[1] = 1;


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

   printf("\n| DYNAMIC
PROGRAMMING
APPROACH           ");

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


   // Print initial subset S0

   printf("\n| S%-2d:
(%.0f,%.0f)                    |", 0,
pair[0].p, pair[0].w);


   for (i = 1; i <= n; i++) {

      k = 0;

      next = h + 1;


      for (j = 0; j <= h; j++) {

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

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


         if (ww <= m) {

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

               k = k + 1;

            }


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

               if (pp < pair[k].p) {

                  pp = pair[k].p;

               }

            } else {

               if (pp > pair[next - 1].p)
{

                  pair[next].p = pp;

                  pair[next].w = ww;

                  next = next + 1;

               }

            }


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

               k = k + 1;

            }

         }

      }


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

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

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

         next = next + 1;

         k = k + 1;

      }


      h = next - 1;

      b[i+1] = next;


      // Print full subset Si

      printf("\n| S%-2d: ", i);

      for (j = 0; j < b[i+1]; j++) {

         printf("(%.0f,%.0f) ",
pair[j].p, pair[j].w);

      }


      // Handle spacing for proper
formatting

      int remaining = 38 - 7 -
(b[i+1]) * 8;

      if (remaining < 0) remaining =
0;

      for (int k = 0; k < remaining;
k++) {
```

```
            printf(" ");
        }
        printf("|");
    }
    printf("\n|----------------------------------------");

float max_profit = 0;
float max_weight = 0;
int max_idx = 0;

for (int i = 0; i <= h; i++) {
    if (pair[i].w <= m && (pair[i].p > max_profit ||
        (pair[i].p == max_profit && pair[i].w > max_weight))) {
        max_profit = pair[i].p;
        max_weight = pair[i].w;
        max_idx = i;
    }
}
    printf("\n| Maximum profit: %.0f, Weight: %.0f |", max_profit, max_weight);
    float remaining_weight = pair[max_idx].w;
    float remaining_profit = pair[max_idx].p;

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

        int j;
        for (j = 0; j <= max_idx; j++) {
            if (pair[j].w == remaining_weight - w[i] &&
                pair[j].p == remaining_profit - p[i]) {
                x[i] = 1;
                remaining_weight -= w[i];
                remaining_profit -= p[i];
                break;
            }
        }
    }

    printf("Selected items: ");
    int any_selected = 0;
    for (int i = 1; i <= n; i++) {
        if (x[i] == 1) {
            printf("%d ", i);
            any_selected = 1;
        }
    }
    if (!any_selected) {
        printf("None");
    }
    printf("\n");

    return max_profit;
}
void displayMenu() {
    printf("\n***** KNAPSACK PROBLEM  *****\n");
    printf("1. Enter new problem instance\n");
    printf("2. Solve using DP-Knapsack algorithm\n");
    printf("3. Display solution\n");
    printf("4. Exit\n");
    printf("Enter your choice: ");
}

void printItems(float p[], float w[], int n) {
    printf("\nItem Details:\n");
    printf("Item\tProfit\tWeight\n");
    for (int i = 1; i <= n; i++) {
        printf("%d\t%.2f\t%.2f\n", i, p[i], w[i]);
    }
}

int main() {
    printf("**********************************************************\n");
    printf(" Roll number: 23B-CO-010\n");
    printf(" PR Number - 202311390\n");
    printf("**********************************************************\n\n");
    int n = 0;
    int capacity = 0;
    float profits[MAX_ITEMS+1];
    float weights[MAX_ITEMS+1];
    int solution[MAX_ITEMS+1] = {0};
    PW pairs[MAX_CAPACITY + 1];
    float optimal_value = 0;
    int choice;
    int solved = 0;

    while (1) {
        displayMenu();
        scanf("%d", &choice);

        switch (choice) {
            case 1:
```

```
        printf("\nEnter the number of items: ");
        scanf("%d", &n);

        if (n <= 0 || n > MAX_ITEMS) {
            printf("Invalid number of items. Please enter a value between 1 and %d.\n", MAX_ITEMS);
            break;
        }

        printf("\nEnter the capacity of the knapsack: ");
        scanf("%d", &capacity);

        if (capacity <= 0 || capacity > MAX_CAPACITY) {
            printf("Invalid capacity. Please enter a value between 1 and %d.\n", MAX_CAPACITY);
            break;
        }

        printf("\nEnter the profit and weight for each item:\n");
        for (int i = 1; i <= n; i++) {
            printf("Item %d profit: ", i);
            scanf("%f", &profits[i]);

            printf("Item %d weight: ", i);
            scanf("%f", &weights[i]);

            if (profits[i] < 0 || weights[i] <= 0) {
                printf("Invalid profit or weight. Profit should be non-negative and weight should be positive.\n");
                i--;
            }
        }

        printItems(profits, weights, n);
        solved = 0;
        printf("\nProblem instance entered successfully.\n");
        break;

    case 2:
        if (n <= 0) {
            printf("\nPlease enter a problem instance first (option 1).\n");
            break;
        }

        printf("\nSolving the knapsack problem.\n");
        start_timer();
        optimal_value = dpknap(profits, weights, n, capacity, pairs, solution);
        double time_taken = end_timer(start_timer());
        printf("\nExecution time: %.2f ms\n", time_taken);

        solved = 1;
        printf("\nProblem solved! The optimal value is %.2f\n", optimal_value);
        break;

    case 3:
        if (!solved) {
            printf("\nPlease solve the problem first (option 2).\n");
            break;
        }

        printf("\n----- KNAPSACK SOLUTION -----\n");
        printf("Optimal value: %.2f\n", optimal_value);
        printf("Selected items: ");
        int any_selected = 0;
        for (int i = 1; i <= n; i++) {
            if (solution[i] == 1) {
                printf("%d ", i);
                any_selected = 1;
            }
        }
        if (!any_selected) {
            printf("None");
        }
        printf("\n");

        printf("\nSelected items details:\n");
        printf("Item\tProfit\tWeight\n");
        float total_weight = 0;
        for (int i = 1; i <= n; i++) {
            if (solution[i] == 1) {
                printf("%d\t%.2f\t%.2f\n", i, profits[i], weights[i]);
                total_weight += weights[i];
            }
        }
        printf("\nTotal profit: %.2f\n", optimal_value);
        printf("Total weight: %.2f / %d\n", total_weight, capacity);
```

```
        break;
                                                    default:                                    return 0;
    case 4:                                             printf("\nInvalid choice.           }
                                                Please try again.\n");
        printf("\nExiting ...\n");
                                                    }
        return 0;
                                                }
```

INPUT –

```
************************************************************
 Roll number: 23B-CO-010
 PR Number - 202311390
************************************************************


***** KNAPSACK PROBLEM  *****
1. Enter new problem instance
2. Solve using DP-Knapsack algorithm
3. Display solution
4. Exit
Enter your choice: 1

Enter the number of items: 7

Enter the capacity of the knapsack: 20

Enter the profit and weight for each item:
Item 1 profit: 2
Item 1 weight: 5
Item 2 profit: 5
Item 2 weight: 4
Item 3 profit: 6
Item 3 weight: 2
Item 4 profit: 3
Item 4 weight: 5
Item 5 profit: 5
Item 5 weight: 3
Item 6 profit: 4
Item 6 weight: 2
Item 7 profit: 3
Item 7 weight: 4

Item Details:
Item    Profit  Weight
1       2.00    5.00
2       5.00    4.00
3       6.00    2.00
4       3.00    5.00
5       5.00    3.00
6       4.00    2.00
7       3.00    4.00

Problem instance entered successfully.
```

OUTPUT –

```
Solving the knapsack problem.

|-----------------------------------------
| DYNAMIC PROGRAMMING APPROACH
|-----------------------------------------
| S0 : (0,0)                             |
| S1 : (0,0) (2,5)                       |
| S2 : (0,0) (2,5) (5,4) (7,9)           |
| S3 : (0,0) (2,5) (5,4) (7,9) (8,7) (11,6) (13,11)           |
| S4 : (0,0) (2,5) (5,4) (7,9) (8,7) (11,6) (13,11) (14,11) (16,16)           |
| S5 : (0,0) (2,5) (5,4) (7,9) (8,7) (11,6) (13,11) (14,11) (16,16) (18,14) (19,14) (21,19)           |
| S6 : (0,0) (2,5) (5,4) (7,9) (8,7) (11,6) (13,11) (14,11) (16,16) (18,14) (19,14) (21,19) (22,16) (23,16)           |
| S7 : (0,0) (2,5) (5,4) (7,9) (8,7) (11,6) (13,11) (14,11) (16,16) (18,14) (19,14) (21,19) (22,16) (23,16) (25,20) (26,20)           |
|-----------------------------------------
| Maximum profit: 26, Weight: 20 |Selected items: 2 3 4 5 6 7

Problem solved! The optimal value is 26.00
```

```
----- KNAPSACK SOLUTION -----
Optimal value: 26.00
Selected items: 2 3 4 5 6 7

Selected items details:
Item    Profit  Weight
2       5.00    4.00
3       6.00    2.00
4       3.00    5.00
5       5.00    3.00
6       4.00    2.00
7       3.00    4.00

Total profit: 26.00
Total weight: 20.00 / 20
```

TIME TAKEN -

```
Execution time: 0.00 ms
```

CONCLUSION – 0/1 KNAPSACK PROBLEM WAS SUCCESSFULLY SOLVED USING DYNAMIC PROGRAMMING APPROACH