**Experiment 1 — 0/1 Knapsack Problem (Dynamic Programming)**

**Aim**

Solve the 0/1 Knapsack problem using dynamic programming and analyze time and space complexity.

**Algorithm (stepwise)**

1. Start and read number of items and bag capacity.
2. Take input of weights and values for each item.
3. Create a table to store the maximum value for each weight limit.
4. For every item, check two options:

   Include the item (if it fits in the bag).

   Exclude the item.

5. Choose the option which gives a higher value.
6. Continue filling the table for all items and capacities.
7. The last value in the table is the maximum value that can be carried.
8. Stop.

**Code (C++):**

```
#include <bits/stdc++.h>

using namespace std;


int main() {
   int n, W;
   cout << "Enter number of items: ";
   cin >> n;
   vector<int> weight(n), value(n);
   cout << "Enter weights:\n";
   for (int i = 0; i < n; i++) cin >> weight[i];
   cout << "Enter values:\n";
   for (int i = 0; i < n; i++) cin >> value[i];
   cout << "Enter capacity of knapsack: ";
   cin >> W;


   vector<vector<int>> dp(n+1, vector<int>(W+1, 0));
   for (int i = 1; i <= n; i++) {
      for (int w = 1; w <= W; w++) {
```

```
            if (weight[i-1] <= w)
                dp[i][w] = max(value[i-1] + dp[i-1][w-weight[i-1]], dp[i-1][w]);
            else
                dp[i][w] = dp[i-1][w];
        }
    }

    cout << "\nMaximum value that can be carried = " << dp[n][W] << endl;
    return 0;
}
```

**Example Input**

n = 3

weights: 2 3 4

values: 3 4 5

W = 5

**Output**

Maximum value that can be carried = 7

**Complexity**

- Time: O(n × W)
- Space: O(n × W) (can be optimized to O(W))

**Conclusion**

DP fills a table of subproblems to avoid recomputation and finds the optimal value efficiently for moderate W.

---

**Experiment 2 — Travelling Sales Person (TSP) using DP (Bitmasking)**

**Aim**

Write a C++ program to implement TSP using dynamic programming (bitmask DP).

**Algorithm (stepwise)**

1. Start and read the number of cities and distance matrix.
2. Choose one city as the starting point.
3. Visit each city exactly once and return to the start.
4. For each city, calculate the cost of visiting all other unvisited cities.
5. Keep track of the minimum total cost of the tour.

6. Compare all possible routes and select the smallest total distance.
7. Display the minimum cost of visiting all cities.
8. Stop.

**Code (C++):**

```
#include <bits/stdc++.h>

using namespace std;

const int INF = 1e9;


int main() {
   int n;
   cout << "Enter number of cities: ";
   cin >> n;
   vector<vector<int>> dist(n, vector<int>(n));
   cout << "Enter distance matrix:\n";
   for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++) cin >> dist[i][j];


   int START = 0;
   int FULL = (1<<n);
   vector<vector<int>> dp(FULL, vector<int>(n, INF));
   dp[1<<START][START] = 0;


   for (int mask = 0; mask < FULL; mask++) {
      for (int u = 0; u < n; u++) {
         if (!(mask & (1<<u))) continue;
         if (dp[mask][u] == INF) continue;
         for (int v = 0; v < n; v++) {
            if (mask & (1<<v)) continue;
            dp[mask | (1<<v)][v] = min(dp[mask | (1<<v)][v], dp[mask][u] + dist[u][v]);
         }
      }
   }
```

```
    int ans = INF;
    for (int i = 0; i < n; i++) {
        if (dp[FULL-1][i] < INF)
            ans = min(ans, dp[FULL-1][i] + dist[i][START]);
    }

    cout << "\nMinimum tour cost = " << ans << endl;
    return 0;
}
```

**Example Input**

For 4 cities:

0 10 15 20

10 0 35 25

15 35 0 30

20 25 30 0

**Output**

Minimum tour cost = 80

**Complexity**

- Time: $O(n^2 \times 2^n)$
- Space: $O(n \times 2^n)$

**Conclusion**

Bitmask DP solves TSP exactly for small n ($n \leq \sim 15$) by storing states of visited sets.

---

**Experiment 3 — All-Pairs Shortest Paths (Floyd–Warshall)**

**Aim**

Compute shortest paths between all pairs of vertices using dynamic programming.

**Algorithm (stepwise)**

1. Start and read the number of vertices and the adjacency matrix.
2. Replace all missing edges with infinity.
3. Take each vertex as an intermediate point one by one.

4. Update the shortest distance between every pair of vertices through this vertex if it gives a smaller path.
5. Repeat this for all vertices.
6. After completing all updates, the matrix will show the shortest distance between every pair of vertices.
7. Print the final matrix.
8. Stop.

**Code (C++):**

```
#include <bits/stdc++.h>

using namespace std;

const int INF = 1e9;


int main() {
   int n;
   cout << "Enter number of vertices: ";
   cin >> n;
   vector<vector<int>> dist(n, vector<int>(n));
   cout << "Enter adjacency matrix (use " << INF << " for INF):\n";
   for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++) cin >> dist[i][j];


   for (int k = 0; k < n; k++)
      for (int i = 0; i < n; i++)
         for (int j = 0; j < n; j++)
            if (dist[i][k] + dist[k][j] < dist[i][j])
               dist[i][j] = dist[i][k] + dist[k][j];


   cout << "\nShortest distance matrix:\n";
   for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
         if (dist[i][j] >= INF/2) cout << "INF ";
         else cout << dist[i][j] << " ";
      }
```

```
        cout << "\n";
    }
    return 0;
}
```

**Example Input**

n = 4

0 3 INF 7

8 0 2 INF

5 INF 0 1

2 INF INF 0

**Output**

0 3 5 6

5 0 2 3

3 6 0 1

2 5 7 0

**Complexity**

- Time: $O(V^3)$
- Space: $O(V^2)$

**Conclusion**

Floyd–Warshall is straightforward and computes all-pairs shortest paths even when edges have negative weights (but no negative cycles).

---

**Experiment 4 — Longest Increasing Subsequence (LIS) using DP**

**Aim**

Compute the length of the Longest Increasing Subsequence and analyze complexities.

**Algorithm (stepwise)**

1. Start and read the number of elements and array values.
2. For each element, find the longest sequence that ends with that element.
3. Compare it with all previous smaller elements to extend the sequence.
4. Keep storing the length of the longest sequence so far.
5. After checking all elements, find the maximum among all lengths.

6. Print that as the length of the longest increasing subsequence.
7. Stop.

**Code (C++):**

```
#include <bits/stdc++.h>
using namespace std;

int main() {
   int n;
   cout << "Enter number of elements: ";
   cin >> n;
   vector<int> arr(n);
   cout << "Enter array elements:\n";
   for (int i = 0; i < n; i++) cin >> arr[i];

   vector<int> lis(n, 1);
   for (int i = 1; i < n; i++)
      for (int j = 0; j < i; j++)
         if (arr[i] > arr[j] && lis[i] < lis[j] + 1)
            lis[i] = lis[j] + 1;

   int ans = *max_element(lis.begin(), lis.end());
   cout << "Length of Longest Increasing Subsequence = " << ans << endl;
   return 0;
}
```

**Example Input**

8

10 22 9 33 21 50 41 60

**Output**

Length of Longest Increasing Subsequence = 5

**Complexity**

- Time: $O(n^2)$

- Space: O(n)

**Conclusion**

DP computes LIS length by building on previous subsequences; can be optimized to O(n log n) with patience sorting method if needed.

---

**Experiment 5 — N-Queens Problem (Backtracking)**

**Aim**

Place N queens on an N×N board so no two attack each other; use backtracking and analyze complexities.

**Algorithm (stepwise)**

1. Start and read the number of queens (N).
2. Place one queen in the first row and first column.
3. Move to the next row and try placing a queen in each column.
4. Before placing, check if it is safe (no other queen attacks it).
5. If safe, place the queen and move to the next row.
6. If not safe, try the next column.
7. If no column works, go back to the previous row (backtrack) and move the queen.
8. Repeat until all queens are placed safely.
9. Display one valid arrangement of queens.
10. Stop.

**Code (C++):**

```
#include <bits/stdc++.h>

using namespace std;


bool isSafe(vector<int>& colForRow, int row, int c) {
   for (int r = 0; r < row; ++r) {
      int cc = colForRow[r];
      if (cc == c || abs(cc - c) == abs(r - row)) return false;
   }
   return true;
}


bool solveNQUtil(int n, vector<int>& colForRow, int row) {
   if (row == n) return true;
   for (int c = 0; c < n; ++c) {
```

```
        if (isSafe(colForRow, row, c)) {
            colForRow[row] = c;
            if (solveNQUtil(n, colForRow, row + 1)) return true;
            colForRow[row] = -1;
        }
    }
    return false;
}

int main() {
    int n;
    cout << "Enter the number of queens: ";
    cin >> n;
    vector<int> colForRow(n, -1);
    if (solveNQUtil(n, colForRow, 0)) {
        cout << "One possible solution (row -> column):\n";
        for (int r = 0; r < n; ++r) cout << r << " -> " << colForRow[r] << "\n";
    } else cout << "No solution exists.\n";
    return 0;
}
```

**Example Input**

n = 4

**Output**

One possible arrangement (columns may vary), e.g.:

0 -> 1

1 -> 3

2 -> 0

3 -> 2

**Complexity**

- Time: O(N!) (upper bound)
- Space: $O(N^2)$ or O(N) for placements + recursion stack

**Conclusion**

Backtracking prunes invalid placements early; feasible for small N (N ≤ 14 generally).

---

**Experiment 6 — Rat in a Maze (Backtracking)**

**Aim**

Find all possible paths for a rat to reach destination cell in a maze using backtracking.

**Algorithm (stepwise)**

1. Start and read the size of the maze and the maze matrix (1 = path, 0 = blocked).
2. Begin from the top-left cell (0,0).
3. Move in all possible directions (Down, Left, Right, Up) one by one.
4. Check if the move is inside the maze and not already visited.
5. If a move leads to the destination (bottom-right cell), record that path.
6. If no move is possible, go back (backtrack) and try another path.
7. Continue until all possible paths are explored.
8. Print all paths that reach the destination.
9. Stop.

**Code (C++):**

```
#include <bits/stdc++.h>

using namespace std;


int n;

bool isSafe(int x, int y, vector<vector<int>>& maze, vector<vector<int>>& vis) {
   return x>=0 && y>=0 && x<n && y<n && maze[x][y]==1 && vis[x][y]==0;
}


void solveMaze(int x, int y, vector<vector<int>>& maze, vector<vector<int>>& vis,
            string path, vector<string>& results) {
   if (x == n-1 && y == n-1) {
      results.push_back(path);
      return;
   }
   vis[x][y] = 1;
   if (isSafe(x+1,y,maze,vis)) solveMaze(x+1,y,maze,vis,path+"D",results);
   if (isSafe(x,y-1,maze,vis)) solveMaze(x,y-1,maze,vis,path+"L",results);
```

```
    if (isSafe(x,y+1,maze,vis)) solveMaze(x,y+1,maze,vis,path+"R",results);
    if (isSafe(x-1,y,maze,vis)) solveMaze(x-1,y,maze,vis,path+"U",results);
    vis[x][y] = 0;
}

int main() {
    cout << "Enter size of maze (n): ";
    cin >> n;
    vector<vector<int>> maze(n, vector<int>(n));
    cout << "Enter maze (0 blocked,1 open):\n";
    for (int i=0;i<n;i++) for (int j=0;j<n;j++) cin >> maze[i][j];

    vector<vector<int>> vis(n, vector<int>(n,0));
    vector<string> results;
    if (maze[0][0] == 1) solveMaze(0,0,maze,vis,"",results);

    if (results.empty()) cout << "No path exists.\n";
    else {
        cout << "All possible paths:\n";
        for (auto &p: results) cout << p << "\n";
    }
    return 0;
}
```

**Example Input**

n = 4

1 0 0 0

1 1 0 1

0 1 0 0

1 1 1 1

**Output**

All possible paths:

DDRDRR

DRDDRR

**Complexity**

- Time: O(4^(n²)) (very large upper bound)
- Space: O(n²) for visited + recursion stack

**Conclusion**

Backtracking explores all possible routes; used in pathfinding examples and simple robot navigation.

---

**Experiment 7 — Hamiltonian Cycle (Backtracking)**

**Aim**

Determine whether a Hamiltonian cycle exists in a graph using backtracking.

**Algorithm (stepwise)**

1. Start and read the number of vertices and adjacency matrix.
2. Choose the first vertex as the starting point.
3. Add one vertex at a time to the current path.
4. Check if the new vertex is connected to the previous one and not already used.
5. If valid, include it in the path and move to the next position.
6. If no vertex can be added, backtrack to the previous step and try a new vertex.
7. When all vertices are included, check if the last vertex connects back to the first.
8. If yes, a Hamiltonian cycle exists.
9. Display the cycle.
10. Stop.

**Code (C++):**

```
#include <bits/stdc++.h>

using namespace std;


bool isSafe(int v, vector<vector<int>>& graph, vector<int>& path, int pos) {

   if (!graph[path[pos-1]][v]) return false;

   for (int i = 0; i < pos; i++) if (path[i] == v) return false;

   return true;

}


bool hamCycleUtil(vector<vector<int>>& graph, vector<int>& path, int pos, int n) {

   if (pos == n) return graph[path[pos-1]][path[0]] == 1;
```

```
    for (int v = 1; v < n; v++) {
        if (isSafe(v, graph, path, pos)) {
            path[pos] = v;
            if (hamCycleUtil(graph, path, pos+1, n)) return true;
            path[pos] = -1;
        }
    }
    return false;
}

int main() {
    int n;
    cout << "Enter number of vertices: ";
    cin >> n;
    vector<vector<int>> graph(n, vector<int>(n));
    cout << "Enter adjacency matrix:\n";
    for (int i=0;i<n;i++) for (int j=0;j<n;j++) cin >> graph[i][j];

    vector<int> path(n, -1);
    path[0] = 0;
    if (hamCycleUtil(graph, path, 1, n)) {
        cout << "Hamiltonian Cycle Exists:\n";
        for (int i=0;i<n;i++) cout << path[i] << " -> ";
        cout << path[0] << "\n";
    } else cout << "No Hamiltonian Cycle exists.\n";
    return 0;
}
```

**Example Input**

```
n = 5
1 0 0 0
1 1 0 1
```

0 1 0 0

1 1 1 1

**Output**

Hamiltonian Cycle Exists:

0 -> 1 -> 2 -> 4 -> 3 -> 0

**Complexity**

- Time: O(n!) (exponential)
- Space: O(n)

**Conclusion**

Backtracking checks permutations and stops early when path fails; works for small graphs.

---

**Experiment 8 — Graph m-Coloring (Backtracking)**

**Aim**

Determine if a graph can be colored with ≤ m colors so adjacent vertices have different colors.

**Algorithm (stepwise)**

1. Start and read the number of vertices, adjacency matrix, and number of colors (m).
2. Assign color 1 to the first vertex.
3. Move to the next vertex and try all colors one by one.
4. Before coloring, check if any connected vertex has the same color.
5. If safe, assign the color and move to the next vertex.
6. If not safe, try another color.
7. If no color works, backtrack to the previous vertex and change its color.
8. Repeat until all vertices are colored.
9. Display the color assigned to each vertex.
10. Stop.

**Code (C++):**

```
#include <bits/stdc++.h>

using namespace std;


bool isSafe(int v, vector<vector<int>>& graph, vector<int>& color, int c, int n) {

   for (int i=0;i<n;i++) if (graph[v][i] && color[i]==c) return false;

   return true;

}
```

```
bool graphColoringUtil(vector<vector<int>>& graph, int m, vector<int>& color, int v, int n) {
    if (v == n) return true;
    for (int c = 1; c <= m; c++) {
        if (isSafe(v, graph, color, c, n)) {
            color[v] = c;
            if (graphColoringUtil(graph, m, color, v+1, n)) return true;
            color[v] = 0;
        }
    }
    return false;
}

int main() {
    int n, m;
    cout << "Enter number of vertices: ";
    cin >> n;
    vector<vector<int>> graph(n, vector<int>(n));
    cout << "Enter adjacency matrix:\n";
    for (int i=0;i<n;i++) for (int j=0;j<n;j++) cin >> graph[i][j];
    cout << "Enter number of colors: ";
    cin >> m;

    vector<int> color(n, 0);
    if (graphColoringUtil(graph, m, color, 0, n)) {
        cout << "Solution Exists: Following are the assigned colors\n";
        for (int i=0;i<n;i++) cout << "Vertex " << i+1 << " -> Color " << color[i] << "\n";
    } else cout << "Solution does not exist.\n";
    return 0;
}
```

**Example Input**

As in previous example with m = 3

**Output**

Solution Exists...

Vertex 1 -> Color 1

...

**Complexity**

- Time: O(m^n) (exponential)
- Space: O(n)

**Conclusion**

Backtracking tries all color assignments; works for small graphs and small m.

---

**Experiment 9 — Weather Data Archiving & Dynamic Filtering**

**Aim**

Create a system to archive weekly weather data and allow dynamic filtering for high-temperature days in the past month.

**Algorithm (stepwise)**

1. Start and create a table for 4 weeks and 7 days.
2. Enter daily temperature values for each week.
3. Enter a threshold temperature value.
4. Go through each week and each day one by one.
5. Compare the temperature with the threshold value.
6. If it is greater, print that day and its temperature.
7. Continue for all weeks and days.
8. If no temperature is above the threshold, print a message accordingly.
9. Stop.

**Code (C++):**

```
#include <iostream>

using namespace std;


int main() {

   const int weeks = 4, days = 7;

   float temp[weeks][days];

   cout << "Enter temperatures for 4 weeks (7 days each):\n";

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

      cout << "Week " << i+1 << ": ";
```

```
    for (int j = 0; j < days; j++) cin >> temp[i][j];
  }
  float threshold;
  cout << "Enter temperature threshold: ";
  cin >> threshold;

  cout << "\nDays with temperature above " << threshold << "°C:\n";
  bool found = false;
  for (int i = 0; i < weeks; i++)
    for (int j = 0; j < days; j++)
      if (temp[i][j] > threshold) {
        cout << "Week " << i+1 << " - Day " << j+1 << ": " << temp[i][j] << "°C\n";
        found = true;
      }
  if (!found) cout << "No days found above the threshold.\n";
  return 0;
}
```

**Example Input**

Enter temperatures for 4 weeks (7 days each):

Week 1: 32 34 33 31 35 36 30

Week 2: 29 31 32 34 33 35 37

Week 3: 30 32 34 36 33 31 29

Week 4: 35 37 38 36 34 32 30

Enter temperature threshold: 35

**Example Output**

Week 1 - Day 6: 36°C

Week 2 - Day 7: 37°C

...

**Complexity**

- Time: O(n) (here n = 28)
- Space: O(n)

**Conclusion**

Simple archival and filtering logic; can be extended to compute averages, monthly highs, or store data in files/DB.

---

**Experiment 10 — Dijkstra's Algorithm (Single-Source Shortest Paths)**

**Aim**

Find shortest paths from a source to all vertices in a weighted graph (non-negative weights).

**Algorithm (stepwise)**

1. Start and read the number of vertices and edges of the graph.
2. Choose a starting vertex.
3. Set the initial distance of all vertices to infinity except the source (set to 0).
4. Pick the vertex with the smallest distance that has not been visited.
5. Update the distances of its neighboring vertices if a shorter path is found.
6. Mark the current vertex as visited.
7. Repeat the process until all vertices are visited.
8. Print the shortest distance from the source to every other vertex.
9. Stop.

**Code (C++ using priority_queue)**

```
#include <bits/stdc++.h>

using namespace std;

const int INF = 1e9;

typedef pair<int,int> pii;


int main() {

   int n;

   cout << "Enter number of vertices: ";

   cin >> n;

   vector<vector<pair<int,int>>> adj(n);

   cout << "Enter number of edges: ";

   int m; cin >> m;

   cout << "Enter edges (u v w) 0-based:\n";

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

      int u,v,w; cin >> u >> v >> w;
```

```
        adj[u].push_back({v,w});
        // If undirected: adj[v].push_back({u,w});
    }
    int src;
    cout << "Enter source vertex: ";
    cin >> src;

    vector<int> dist(n, INF);
    dist[src] = 0;
    priority_queue<pii, vector<pii>, greater<pii>> pq;
    pq.push({0, src});
    while (!pq.empty()) {
        auto [d,u] = pq.top(); pq.pop();
        if (d > dist[u]) continue;
        for (auto &edge : adj[u]) {
            int v = edge.first, w = edge.second;
            if (dist[u] + w < dist[v]) {
                dist[v] = dist[u] + w;
                pq.push({dist[v], v});
            }
        }
    }

    cout << "Vertex  Distance from Source\n";
    for (int i=0;i<n;i++) {
        if (dist[i] >= INF) cout << i << "\tINF\n";
        else cout << i << "\t" << dist[i] << "\n";
    }
    return 0;
}
```

**Example Input**

Graph as earlier converted to edge list; source = 0

**Output**

Vertex  Distance from Source

0  0

1  8

2  9

3  5

4  7

**Complexity**

- Time: O(E log V) using heap (priority queue)
- Space: O(V + E)

**Conclusion**

Dijkstra with priority queue is efficient for sparse graphs; adjacency matrix version is $O(V^2)$.