DAA LAB REPORT

THAKUR JAIDEEP SINGH
DSAI - 221020455

1.1 BINARY SEARCH

```cpp
// Binary Search (Iterative)
int binary_search_iterative(vector<int>& nums, int target) {
    int left = 0, right = nums.size() - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] == target)
            return mid;
        else if (nums[mid] < target)
            left = mid + 1;
        else
            right = mid - 1;
    }
    return -1;
}

// Binary Search (Recursive)
int binary_search_recursive(vector<int>& nums, int target, int left, int right) {
    if (left <= right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] == target)
            return mid;
        else if (nums[mid] < target)
            return binary_search_recursive(nums, target, mid + 1, right);
        else
            return binary_search_recursive(nums, target, left, mid - 1);
    }
    return -1;
}
```

Output

```
Input Array: [1, 3, 5, 7, 9, 11, 13, 15]
Target: 7

Iterative Binary Search:
Element 7 found at index: 3

Recursive Binary Search:
Element 7 found at index: 3
```

Time Complexity : O(Log(N))

1.2 MERGE SORT

```
void merge(vector<int>& arr, int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;

    vector<int> L(n1), R(n2);

    for (int i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

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

    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = R[j];
```

```
        j++;
        k++;
    }
}

void mergeSort(vector<int>& arr, int l, int r) {
    if (l >= r) return;
    int m = l + (r - l) / 2;
    mergeSort(arr, l, m);
    mergeSort(arr, m + 1, r);
    merge(arr, l, m, r);
}
```

OUTPUT:

```
Original array: 12 11 13 5 6 7
Sorted array: 5 6 7 11 12 13
```

TIME COMPLEXITY = O(NLOG(N))
SPACE COMPLEXITY = O(N)

1.3 QUICK SORT

```
int partition(vector<int>& arr, int low, int high) {
    int pivot = arr[low];
    int i = low;

    for (int j = low + 1; j <= high; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }

    swap(arr[i], arr[low]);
    return i;
}

void quickSort(vector<int>& arr, int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
```

```
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

OUTPUT:

```
Original array:
12 3 6 1 8 10 9 2 7 4 11 5
Sorted array:
1 2 3 4 5 6 7 8 9 10 11 12
```

TIME COMPLEXITY
Best Case - O(NLOG(N))
Worst Case - O(N^2)

1.4 FIND INDEX FOR A[i] = i;

```
int findFixedPoint(const vector<int>& A) {
    int low = 0;
    int high = A.size() - 1;

    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (A[mid] == mid) {
            return mid;
        } else if (A[mid] < mid) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }

    return -1;
}
```

OUTPUT:

```
Given Array:
-10, -5, 0, 3, 7,
Found fixed point at index 3. A[3] = 3
```

TIME COMPLEXITY : O(LOG(N));

3.1 SOLUTION

```
bool hasPairWithSum(vector<int>& arr, int K) {
    // Sort the array in O(n log n) time
    sort(arr.begin(), arr.end());

    // Initialize two pointers
    int left = 0;
    int right = arr.size() - 1;

    // Loop until the pointers meet
    while (left < right) {
        int sum = arr[left] + arr[right];
        if (sum == K) {
            return true; // Pair found
        } else if (sum < K) {
            left++; // Increase the sum
        } else {
            right--; // Decrease the sum
        }
    }

    return false; // Pair not found
}
```

OUTPUT

```
Input Array : 1, 4, 5, 7, 9, 10,
Target : 13
Yes, there exists a pair with sum 13.
```

TIME COMPLEXITY : O(N Log(N))

3.2 SOLUTION

```
bool hasTripletsWithSum(const vector<int>& arr, int K) {
    int n = arr.size();

    // Sort the array in O(n log n) time
    vector<int> sortedArr = arr;
    sort(sortedArr.begin(), sortedArr.end());

    // Fix the first element and use two pointers technique for the
remaining elements
    for (int i = 0; i < n - 2; ++i) {
        int left = i + 1;
        int right = n - 1;
        while (left < right) {
            int sum = sortedArr[i] + sortedArr[left] + sortedArr[right];
            if (sum == K) {
                return true; // Triplet found
            } else if (sum < K) {
                left++; // Increase the sum
            } else {
                right--; // Decrease the sum
            }
        }
    }

    return false; // Triplet not found
}
```

OUTPUT

```
Input array: 1 4 5 7 9 10
Target sum: 22
Yes, there exists a triplet with sum 22.
```

TIME COMPLEXITY : O(N^2)

3.3 SOLUTION

```cpp
bool hasPairWithSum(vector<int>& A, vector<int>& B, int K) {
    // Sort both arrays
    sort(A.begin(), A.end());
    sort(B.begin(), B.end());

    // Initialize pointers
    int i = 0;
    int j = B.size() - 1;

    // Search for pairs
    while (i < A.size() && j >= 0) {
        int sum = A[i] + B[j];
        if (sum == K) {
            cout << "A[" << i << "] = " << A[i] << ", B[" << j << "] = "
<< B[j] << ", Sum = " << sum << endl;
            return true;
        } else if (sum < K) {
            i++;
        } else {
            j--;
        }
    }

    // No such pair found
    return false;
}
```

OUTPUT

```
Array A: 1 4 7 10
Array B: 2 3 9 11
Target Sum: 13
A[1] = 4, B[2] = 9, Sum = 13
There exists a pair in A and B with sum 13
```

TIME COMPLEXITY : O(N Log(N))

3.4 SOLUTION

```
bool hasDuplicate(const vector<int>& arr) {
    // Sort the array
    vector<int> sortedArr = arr;
    sort(sortedArr.begin(), sortedArr.end());

    // Check for duplicates
    for (int i = 0; i < sortedArr.size() - 1; ++i) {
        if (sortedArr[i] == sortedArr[i+1]) {
            return true; // Duplicate found
        }
    }

    return false; // No duplicates found
}
```

OUTPUT

```
Input array: 3 7 2 9 4 3
Duplicate elements exist in the array.
```

Time Complexity : O(N Log(N))

3.5 SOLUTION

```
int findMaxOccurringElement(vector<int>& arr) {
    map<int, int> freqMap;

    // Count the occurrences of each element
    for (int num : arr) {
        freqMap[num]++;
    }

    int maxCount = 0;
    int res = -1;

    // Find the element with maximum occurrences
    for (auto& pair : freqMap) {
```

```
        if (pair.second > maxCount) {
            maxCount = pair.second;
            res = pair.first;
        }
    }

    return res;
}
```

OUTPUT

```
Input Array: 1 2 3 4 5 6 6 6 6 7 8 9 6
Element with maximum occurrences: 6
```

Time Complexity : O(N)

4.1 KNAPSACK PROBLEM

```
float knapsack(vector<float> profits, vector<float> weights, int
max_weight) {
    if (profits.empty() || weights.empty() || profits.size() !=
weights.size() || max_weight <= 0) {
        cout << "Invalid input or empty knapsack!" << endl;
        return 0;
    }

    vector<pair<float, float>> mpp;

    for (int i = 0; i < profits.size(); ++i) {
        mpp.push_back({profits[i] / weights[i], weights[i]});
    }

    sort(mpp.begin(), mpp.end(), [](auto a, auto b) { return a.first >
b.first; });

    float profit = 0;
    for (auto it : mpp) {
        if (max_weight <= 0) break;

        if (it.second > max_weight) {
            profit += it.first * max_weight;
```

```
            break;
        } else {
            profit += it.first * it.second;
            max_weight -= it.second;
        }
    }

    cout << "Max profit is: " << profit << endl;
    return profit;
}
```

OUTPUT

```
Input Profits: 60 100 120
Input Weights: 10 20 30
Max profit is: 240
```

Time Complexity : O(N Log(N))

4.2 JOB SEQUENCING

```
float job_scheduling(vector<float> profits, vector<float> deadlines) {
    if (profits.empty() || deadlines.empty() || profits.size() != deadlines.size()) {
        cout << "Invalid input or empty jobs!" << endl;
        return 0;
    }

    int max_deadline = *max_element(deadlines.begin(), deadlines.end());

    vector<int> job_order(max_deadline, -1);
    vector<int> sequence(max_deadline, -1);

    vector<pair<pair<int, int>, int>> mpp;
    for (int i = 0; i < profits.size(); ++i) {
        mpp.push_back({{profits[i], deadlines[i]}, i + 1});
    }

    sort(mpp.begin(), mpp.end(), [](auto a, auto b) { return a.first.first > b.first.first; });
```

```
    for (auto it : mpp) {
        int deadline = it.first.second - 1;
        while (deadline >= 0 && sequence[deadline] != -1) {
            --deadline;
        }
        if (deadline >= 0) {
            sequence[deadline] = it.first.first;
            job_order[deadline] = it.second;
        }
    }

    float sum = 0;
    for (auto it : sequence) {
        if (it != -1) {
            sum += it;
        }
    }

    cout << "Input Profits: ";
    for (float p : profits) {
        cout << p << " ";
    }
    cout << endl;

    cout << "Input Deadlines: ";
    for (float d : deadlines) {
        cout << d << " ";
    }
    cout << endl;

    cout << "Sequence is: ";
    for (auto it : job_order) {
        if (it != -1) {
            cout << "J" << it << ", ";
        }
    }
    cout << endl;

    cout << "Max Profit is: " << sum << endl;
    return sum;
```

```
}
```

OUTPUT:

```
Input Profits: 100 50 200 120
Input Deadlines: 2 1 2 1
Sequence is: J4, J3,
Max Profit is: 320
```

Time Complexity : O(N^2)

4.3 Prim's Algorithm

```
// Function to find the vertex with the minimum key value,
// from the set of vertices not yet included in MST
int minKey(vector<int>& key, vector<bool>& mstSet, int V) {
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; ++v) {
        if (mstSet[v] == false && key[v] < min) {
            min = key[v];
            min_index = v;
        }
    }

    return min_index;
}

// Function to print the constructed MST stored in parent[]
void printMST(vector<int>& parent, vector<vector<int>>& graph, int V) {
    cout << "Edge \tWeight\n";
    for (int i = 1; i < V; ++i) {
        cout << parent[i] << " - " << i << "\t" << graph[i][parent[i]] <<
endl;
    }
}

// Function to construct and print MST for a graph represented using
adjacency matrix representation
```

```cpp
void primMST(vector<vector<int>>& graph, int V) {
    // Array to store constructed MST
    vector<int> parent(V);

    // Key values used to pick minimum weight edge in cut
    vector<int> key(V, INT_MAX);

    // To represent set of vertices included in MST
    vector<bool> mstSet(V, false);

    // Always include first  vertex in MST.
    // Make key 0 so that this vertex is picked as first vertex
    key[0] = 0;
    parent[0] = -1; // First node is always root of MST

    // The MST will have V vertices
    for (int count = 0; count < V - 1; ++count) {
        // Pick the minimum key vertex from the set of vertices
        // not yet included in MST
        int u = minKey(key, mstSet, V);

        // Add the picked vertex to the MST Set
        mstSet[u] = true;

        // Update key value and parent index of the adjacent vertices
        // of the picked vertex. Consider only those vertices which are
        // not yet included in MST
        for (int v = 0; v < V; ++v) {
            // graph[u][v] is non zero only for adjacent vertices of m
            // mstSet[v] is false for vertices not yet included in MST
            // Update the key only if graph[u][v] is smaller than key[v]
            if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])
{
                parent[v] = u;
                key[v] = graph[u][v];
            }
        }
    }

    // Print the constructed MST
```

```
    printMST(parent, graph, V);
}

// Function to display the adjacency matrix of the graph
void displayGraph(vector<vector<int>>& graph, int V) {
    cout << "Adjacency Matrix:" << endl;
    for (int i = 0; i < V; ++i) {
        for (int j = 0; j < V; ++j) {
            cout << graph[i][j] << " ";
        }
        cout << endl;
    }
}
// Driver code
int main() {
    // Sample adjacency matrix representation of a graph
    vector<vector<int>> graph = {
        {0, 2, 0, 6, 0},
        {2, 0, 3, 8, 5},
        {0, 3, 0, 0, 7},
        {6, 8, 0, 0, 9},
        {0, 5, 7, 9, 0}
    };

    int V = graph.size();

    // Display the input graph (adjacency matrix)
    displayGraph(graph, V);

    // Find and display the minimum spanning tree (MST)
    cout << "\nMinimum Spanning Tree (MST) using Prim's Algorithm:\n";
    primMST(graph, V);

    return 0;
}
```

OUTPUT

```
Adjacency Matrix:
0 2 0 6 0
2 0 3 8 5
0 3 0 0 7
6 8 0 0 9
0 5 7 9 0

Minimum Spanning Tree (MST) using Prim's Algorithm:
Edge    Weight
0 - 1   2
1 - 2   3
0 - 3   6
1 - 4   5
```

Time Complexity : O(E Log(V))

4.4 Krushkal's Algorithm

```
// Data structure to represent an edge
struct Edge {
    int src, dest, weight;
};

// Data structure to represent a disjoint set
struct DisjointSet {
    int parent, rank;
};

// Find operation with path compression
int find(vector<DisjointSet>& subsets, int i) {
    if (subsets[i].parent != i) {
        subsets[i].parent = find(subsets, subsets[i].parent);
    }
    return subsets[i].parent;
}

// Union operation by rank
void Union(vector<DisjointSet>& subsets, int x, int y) {
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);
```

```cpp
    if (subsets[xroot].rank < subsets[yroot].rank) {
        subsets[xroot].parent = yroot;
    } else if (subsets[xroot].rank > subsets[yroot].rank) {
        subsets[yroot].parent = xroot;
    } else {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}

// Compare function for sorting edges by weight
bool compareEdges(Edge a, Edge b) {
    return a.weight < b.weight;
}

// Function to find the minimum spanning tree using Kruskal's algorithm
void kruskalMST(vector<vector<int>>& graph, int V) {
    vector<Edge> edges;
    for (int i = 0; i < V; ++i) {
        for (int j = i + 1; j < V; ++j) {
            if (graph[i][j] != 0) {
                edges.push_back({i, j, graph[i][j]});
            }
        }
    }

    // Sort all the edges in non-decreasing order of their weight
    sort(edges.begin(), edges.end(), compareEdges);

    vector<Edge> result; // Stores the edges of the MST
    vector<DisjointSet> subsets(V);

    // Create V disjoint sets, one for each vertex
    for (int i = 0; i < V; ++i) {
        subsets[i].parent = i;
        subsets[i].rank = 0;
    }

    // Number of edges to be taken is equal to V-1
```

```
    int e = 0, i = 0;
    while (e < V - 1 && i < edges.size()) {
        Edge next_edge = edges[i++];

        int x = find(subsets, next_edge.src);
        int y = find(subsets, next_edge.dest);

        // If including this edge does not cause cycle, include it
        if (x != y) {
            result.push_back(next_edge);
            Union(subsets, x, y);
            e++;
        }
    }

    // Print the edges of the MST
    cout << "Edges of the Minimum Spanning Tree (MST) using Kruskal's
Algorithm:\n";
    for (i = 0; i < result.size(); ++i) {
        cout << result[i].src << " - " << result[i].dest << " : " <<
result[i].weight << endl;
    }
}

// Function to display the adjacency matrix of the graph
void displayGraph(vector<vector<int>>& graph, int V) {
    cout << "Adjacency Matrix:" << endl;
    for (int i = 0; i < V; ++i) {
        for (int j = 0; j < V; ++j) {
            cout << graph[i][j] << " ";
        }
        cout << endl;
    }
}
// Driver code
int main() {
    // Sample adjacency matrix representation of a graph
    vector<vector<int>> graph = {
        {0, 2, 0, 6, 0},
        {2, 0, 3, 8, 5},
```

```
        {0, 3, 0, 0, 7},
        {6, 8, 0, 0, 9},
        {0, 5, 7, 9, 0}
    };

    int V = graph.size();

    // Display the input graph (adjacency matrix)
    displayGraph(graph, V);

    // Find and display the minimum spanning tree (MST)
    cout << "\nMinimum Spanning Tree (MST) using Kruskal's Algorithm:\n";
    kruskalMST(graph, V);

    return 0;
}
```

OUTPUT

```
Adjacency Matrix:
0 2 0 6 0
2 0 3 8 5
0 3 0 0 7
6 8 0 0 9
0 5 7 9 0

Minimum Spanning Tree (MST) using Kruskal's Algorithm:
Edges of the Minimum Spanning Tree (MST) using Kruskal
's Algorithm:
0 - 1 : 2
1 - 2 : 3
1 - 4 : 5
0 - 3 : 6
```

Time Complexity : O(E Log(E))

4.5 Dijkstra's Algorithm

```
// Function to find the vertex with the minimum distance value
int minDistance(vector<int>& dist, vector<bool>& visited, int V) {
    int min = INT_MAX, min_index;
```

```
    for (int v = 0; v < V; ++v) {
        if (!visited[v] && dist[v] <= min) {
            min = dist[v];
            min_index = v;
        }
    }

    return min_index;
}

// Function to print the shortest distances from source to all other
vertices
void printSolution(vector<int>& dist, int V) {
    cout << "Vertex \t Distance from Source" << endl;
    for (int i = 0; i < V; ++i) {
        cout << i << " \t " << dist[i] << endl;
    }
}

// Function to find the shortest paths from a source vertex using
Dijkstra's algorithm
void dijkstra(vector<vector<int>>& graph, int src, int V) {
    vector<int> dist(V, INT_MAX); // Vector to store the shortest distance
from src to i
    vector<bool> visited(V, false); // Boolean array to track visited
vertices

    dist[src] = 0; // Distance from source to itself is 0

    for (int count = 0; count < V - 1; ++count) {
        int u = minDistance(dist, visited, V); // Get the vertex with the
minimum distance value

        visited[u] = true; // Mark the picked vertex as visited

        // Update the distance values of adjacent vertices of the picked
vertex
        for (int v = 0; v < V; ++v) {
```

```cpp
            if (!visited[v] && graph[u][v] && dist[u] != INT_MAX &&
dist[u] + graph[u][v] < dist[v]) {
                dist[v] = dist[u] + graph[u][v];
            }
        }
    }

    // Print the shortest distances from source to all other vertices
    printSolution(dist, V);
}

// Function to display the adjacency matrix of the graph
void displayGraph(vector<vector<int>>& graph, int V) {
    cout << "Adjacency Matrix:" << endl;
    for (int i = 0; i < V; ++i) {
        for (int j = 0; j < V; ++j) {
            cout << graph[i][j] << " ";
        }
        cout << endl;
    }
}

void Run_Code(){
    vector<vector<int>> graph = {
        {0, 4, 0, 0, 0, 0, 0, 8, 0},
        {4, 0, 8, 0, 0, 0, 0, 11, 0},
        {0, 8, 0, 7, 0, 4, 0, 0, 2},
        {0, 0, 7, 0, 9, 14, 0, 0, 0},
        {0, 0, 0, 9, 0, 10, 0, 0, 0},
        {0, 0, 4, 14, 10, 0, 2, 0, 0},
        {0, 0, 0, 0, 0, 2, 0, 1, 6},
        {8, 11, 0, 0, 0, 0, 1, 0, 7},
        {0, 0, 2, 0, 0, 0, 6, 7, 0}
    };

    int V = graph.size();

    // Display the input graph (adjacency matrix)
    displayGraph(graph, V);
```

```
    // Choose a source vertex (e.g., vertex 0)
    int source = 0;

    // Find and display the shortest paths from the source vertex
    cout << "\nShortest Paths from Source Vertex " << source << ":\n";
    dijkstra(graph, source, V);
}
```

OUTPUT:

```
Adjacency Matrix:
0 4 0 0 0 0 0 8 0
4 0 8 0 0 0 0 11 0
0 8 0 7 0 4 0 0 2
0 0 7 0 9 14 0 0 0
0 0 0 9 0 10 0 0 0
0 0 4 14 10 0 2 0 0
0 0 0 0 0 2 0 1 6
8 11 0 0 0 0 1 0 7
0 0 2 0 0 0 6 7 0

Shortest Paths from Source Vertex 0:
Vertex   Distance from Source
0        0
1        4
2        12
3        19
4        21
5        11
6        9
7        8
8        14
```

Time Complexity: O(V^2)

5.1 optimal order of multiplying n matrices

```
#include <iostream>
#include <vector>
#include <climits>

using namespace std;

int matrixChainOrder(const vector<pair<int, int>>& dims) {
    int n = dims.size();
    vector<vector<int>> dp(n, vector<int>(n, 0));

    for (int len = 2; len < n; len++) {
        for (int i = 1; i < n - len + 1; i++) {
            int j = i + len - 1;
            dp[i][j] = INT_MAX;
            for (int k = i; k <= j - 1; k++) {
                int cost = dp[i][k] + dp[k + 1][j] + dims[i - 1].first *
dims[k].second * dims[j].second;
                if (cost < dp[i][j]) {
                    dp[i][j] = cost;
                }
            }
        }
    }
    return dp[1][n - 1];
}

int main() {
    // Example input: dimensions of matrices
    vector<pair<int, int>> dims = {{10, 20}, {20, 30}, {30, 40}, {40,
30}};

    cout << "Input dimensions of matrices:" << endl;
    for (const auto& dim : dims) {
        cout << "(" << dim.first << ", " << dim.second << ")" << endl;
    }

    int minScalarMultiplications = matrixChainOrder(dims);
    cout << "\nMinimum number of scalar multiplications: " <<
minScalarMultiplications << endl;
```

```
    return 0;
}
```

OUTPUT

```
Input dimensions of matrices:
(10, 20)
(20, 30)
(30, 40)
(40, 30)

Minimum number of scalar multiplications: 24000
```

Time Complexity: O(N^3)

5.2 OBST

```
#include <iostream>
#include <vector>
#include <numeric>
#include <climits>

using namespace std;

float optimalBST(vector<float> &keys, vector<float> &freq) {
    int n = keys.size();
    vector<vector<float>> dp(n + 1, vector<float>(n + 1, 0));

    // Initialize diagonal elements with frequencies
    for (int i = 0; i < n; ++i)
        dp[i][i] = freq[i];

    // Fill the dp table
    for (int len = 1; len <= n; ++len) {
        for (int i = 0; i <= n - len; ++i) {
            int j = i + len - 1;
            dp[i][j] = INT_MAX;
```

```
            // Try making all keys in subarray [i, j] as root
            for (int k = i; k <= j; ++k) {
                float cost = ((k > i) ? dp[i][k - 1] : 0) +
                             ((k < j) ? dp[k + 1][j] : 0) +
                             accumulate(freq.begin() + i, freq.begin() + j
+ 1, 0.0);

                if (cost < dp[i][j])
                    dp[i][j] = cost;
            }
        }
    }

    return dp[0][n - 1];
}

int main() {
    // Example input
    vector<float> keys = {10, 20, 30};
    vector<float> freq = {2, 4, 1};

    // Displaying the input
    cout << "Keys: ";
    for (float key : keys)
        cout << key << " ";
    cout << endl;

    cout << "Frequencies: ";
    for (float f : freq)
        cout << f << " ";
    cout << endl;

    // Running the function
    float result = optimalBST(keys, freq);

    // Displaying the result
    cout << "Optimal Cost: " << result << endl;

    return 0;
}
```

OUTPUT

```
Keys: 10 20 30
Frequencies: 2 4 1
Optimal Cost: 10
```

Time Complexity : O(N^3)

5.3 0/1 Knapsack Problem

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int knapsack(int W, vector<int>& weights, vector<int>& values) {
    int n = weights.size();
    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 (weights[i - 1] <= w) {
                dp[i][w] = max(values[i - 1] + dp[i - 1][w - weights[i - 1]], dp[i - 1][w]);
            } else {
                dp[i][w] = dp[i - 1][w];
            }
        }
    }

    return dp[n][W];
}

int main() {
    int W = 50; // Capacity of knapsack
    vector<int> weights = {10, 20, 30, 40, 50, 60, 70}; // Weights of items
    vector<int> values = {60, 100, 120, 150, 200, 220, 250}; // Values of items
```

```
    cout << "Knapsack Capacity: " << W << endl;

    cout << "Weights of items: ";
    for (int weight : weights)
        cout << weight << " ";
    cout << endl;

    cout << "Values of items: ";
    for (int value : values)
        cout << value << " ";
    cout << endl;

    cout << "Maximum value that can be obtained: " << knapsack(W, weights,
values) << endl;

    return 0;
}
```

Output:

```
Knapsack Capacity: 50
Weights of items: 10 20 30 40 50 60 70
Values of items: 60 100 120 150 200 220 250
Maximum value that can be obtained: 220
```

5.4 All Pairs Shortest Path

```
#include <iostream>
#include <vector>

using namespace std;

const int INF = 1e9; // Infinity value

void floydWarshall(vector<vector<int>>& graph) {
    int n = graph.size();

    // Initialize distance matrix with graph values
    vector<vector<int>> dist(graph);
```

```
    // Update distance matrix using Floyd-Warshall algorithm
    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] != INF && dist[k][j] != INF && dist[i][k] +
dist[k][j] < dist[i][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j];
                }
            }
        }
    }

    // Print the shortest distances
    cout << "Shortest distances between all pairs:" << endl;
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            if (dist[i][j] == INF) {
                cout << "INF ";
            } else {
                cout << dist[i][j] << " ";
            }
        }
        cout << endl;
    }
}

int main() {
    // Example input as adjacency matrix
    vector<vector<int>> graph = {
        {0, 5, INF, 10, INF},
        {INF, 0, 3, INF, INF},
        {INF, INF, 0, 1, INF},
        {INF, INF, INF, 0, 2},
        {INF, INF, INF, INF, 0}
    };

    cout << "Input adjacency matrix:" << endl;
    for (const auto& row : graph) {
        for (int val : row) {
```

```
            if (val == INF) {
                cout << "INF ";
            } else {
                cout << val << " ";
            }
        }
        cout << endl;
    }

    floydWarshall(graph);

    return 0;
}
```

Output

```
Input adjacency matrix:
0 5 INF 10 INF
INF 0 3 INF INF
INF INF 0 1 INF
INF INF INF 0 2
INF INF INF INF 0
Shortest distances between all pairs:
0 5 8 9 11
INF 0 3 4 6
INF INF 0 1 3
INF INF INF 0 2
INF INF INF INF 0
```

Time Complexity : O(V^3)

5.5 Travelling Salesman Problem

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;
```

```
const int INF = 1e9; // Infinity value

int tsp(int n, vector<vector<int>>& graph) {
    vector<vector<int>> dp(1 << n, vector<int>(n, INF));
    dp[1][0] = 0; // Base case: starting from node 0

    for (int mask = 1; mask < (1 << n); ++mask) {
        for (int u = 0; u < n; ++u) {
            if (mask & (1 << u)) { // If node u is included in the subset
represented by mask
                for (int v = 0; v < n; ++v) {
                    if (mask != (1 << v) && graph[v][u] != INF) { // If
node v is not included in the subset and there is an edge from v to u
                        dp[mask][u] = min(dp[mask][u], dp[mask ^ (1 <<
u)][v] + graph[v][u]);
                    }
                }
            }
        }
    }

    // Find the minimum cost of reaching back to node 0
    int minCost = INF;
    for (int u = 1; u < n; ++u) {
        if (graph[u][0] != INF) { // If there is an edge from u to 0
            minCost = min(minCost, dp[(1 << n) - 1][u] + graph[u][0]);
        }
    }

    return minCost;
}

int main() {
    // Example input as adjacency matrix
    vector<vector<int>> graph = {
        {0, 10, 15, 20},
        {10, 0, 35, 25},
        {15, 35, 0, 30},
        {20, 25, 30, 0}
    };
```

```
    int n = graph.size(); // Number of nodes

    cout << "Input adjacency matrix:" << endl;
    for (const auto& row : graph) {
        for (int val : row) {
            if (val == INF) {
                cout << "INF ";
            } else {
                cout << val << " ";
            }
        }
        cout << endl;
    }

    int minCost = tsp(n, graph);

    cout << "Minimum cost of visiting all nodes: " << minCost << endl;

    return 0;
}
```

OUTPUT

```
Input adjacency matrix:
0 10 15 20
10 0 35 25
15 35 0 30
20 25 30 0
Minimum cost of visiting all nodes: 80
```

2.1 Strassen's Multiplication

```
#include <iostream>
#include <vector>

using namespace std;
```

```
vector<vector<int>> matrixAdd(const vector<vector<int>>& A, const
vector<vector<int>>& B) {
    int n = A.size();
    vector<vector<int>> result(n, vector<int>(n, 0));
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            result[i][j] = A[i][j] + B[i][j];
        }
    }
    return result;
}

vector<vector<int>> matrixSub(const vector<vector<int>>& A, const
vector<vector<int>>& B) {
    int n = A.size();
    vector<vector<int>> result(n, vector<int>(n, 0));
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            result[i][j] = A[i][j] - B[i][j];
        }
    }
    return result;
}

vector<vector<int>> strassenMultiply(const vector<vector<int>>& A, const
vector<vector<int>>& B) {
    int n = A.size();
    if (n == 1) {
        return {{A[0][0] * B[0][0]}};
    }

    // Divide matrices into submatrices
    int mid = n / 2;
    vector<vector<int>> A11(mid, vector<int>(mid)), A12(mid,
vector<int>(mid)), A21(mid, vector<int>(mid)), A22(mid, vector<int>(mid));
    vector<vector<int>> B11(mid, vector<int>(mid)), B12(mid,
vector<int>(mid)), B21(mid, vector<int>(mid)), B22(mid, vector<int>(mid));

    for (int i = 0; i < mid; ++i) {
        for (int j = 0; j < mid; ++j) {
```

```
            A11[i][j] = A[i][j];
            A12[i][j] = A[i][j + mid];
            A21[i][j] = A[i + mid][j];
            A22[i][j] = A[i + mid][j + mid];

            B11[i][j] = B[i][j];
            B12[i][j] = B[i][j + mid];
            B21[i][j] = B[i + mid][j];
            B22[i][j] = B[i + mid][j + mid];
        }
    }

    // Strassen's algorithm
    vector<vector<int>> S1 = matrixSub(B12, B22);
    vector<vector<int>> S2 = matrixAdd(A11, A12);
    vector<vector<int>> S3 = matrixAdd(A21, A22);
    vector<vector<int>> S4 = matrixSub(B21, B11);
    vector<vector<int>> S5 = matrixAdd(A11, A22);
    vector<vector<int>> S6 = matrixAdd(B11, B22);
    vector<vector<int>> S7 = matrixSub(A12, A22);
    vector<vector<int>> S8 = matrixAdd(B21, B22);
    vector<vector<int>> S9 = matrixSub(A11, A21);
    vector<vector<int>> S10 = matrixAdd(B11, B12);

    vector<vector<int>> P1 = strassenMultiply(A11, S1);
    vector<vector<int>> P2 = strassenMultiply(S2, B22);
    vector<vector<int>> P3 = strassenMultiply(S3, B11);
    vector<vector<int>> P4 = strassenMultiply(A22, S4);
    vector<vector<int>> P5 = strassenMultiply(S5, S6);
    vector<vector<int>> P6 = strassenMultiply(S7, S8);
    vector<vector<int>> P7 = strassenMultiply(S9, S10);

    vector<vector<int>> C11 = matrixAdd(matrixSub(matrixAdd(P5, P4), P2),
P6);
    vector<vector<int>> C12 = matrixAdd(P1, P2);
    vector<vector<int>> C21 = matrixAdd(P3, P4);
    vector<vector<int>> C22 = matrixSub(matrixSub(matrixAdd(P5, P1), P3),
P7);

    vector<vector<int>> result(n, vector<int>(n));
```

```
    for (int i = 0; i < mid; ++i) {
        for (int j = 0; j < mid; ++j) {
            result[i][j] = C11[i][j];
            result[i][j + mid] = C12[i][j];
            result[i + mid][j] = C21[i][j];
            result[i + mid][j + mid] = C22[i][j];
        }
    }
    return result;
}

void printMatrix(const vector<vector<int>>& matrix) {
    for (const auto& row : matrix) {
        for (int val : row) {
            cout << val << " ";
        }
        cout << endl;
    }
}

int main() {
    vector<vector<int>> A = {{1, 2}, {3, 4}};
    vector<vector<int>> B = {{5, 6}, {7, 8}};

    cout << "Matrix A:" << endl;
    printMatrix(A);
    cout << "Matrix B:" << endl;
    printMatrix(B);

    cout << "Result of matrix multiplication using Strassen's algorithm:"
<< endl;
    vector<vector<int>> C = strassenMultiply(A, B);
    printMatrix(C);

    return 0;
}
```

Output

```
Matrix A:
1 2
3 4
Matrix B:
5 6
7 8
Result of matrix multiplication using Strassen's algor
ithm:
19 22
43 50
```

Time complexity : O(N^log(7))