DAA Post Mid Sem Assignment

Devam Desai - IIT2022035 - Section A

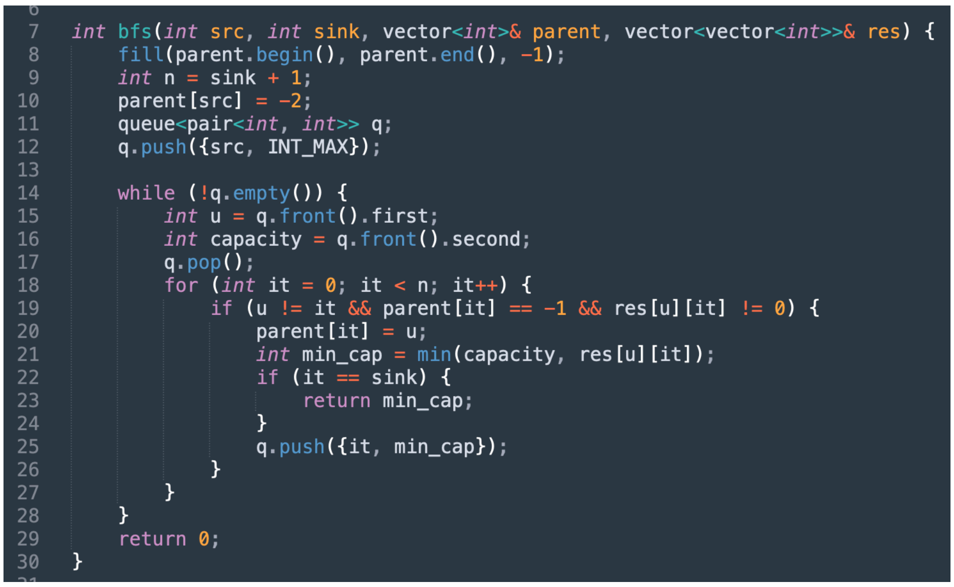
Q1. **Maximum Bipartite Matching:**

Design: We can augment this algorithm as a Maximum Flow problem and solve it using the Edmonds Karp algorithm(Ford Fulkerson implementation).

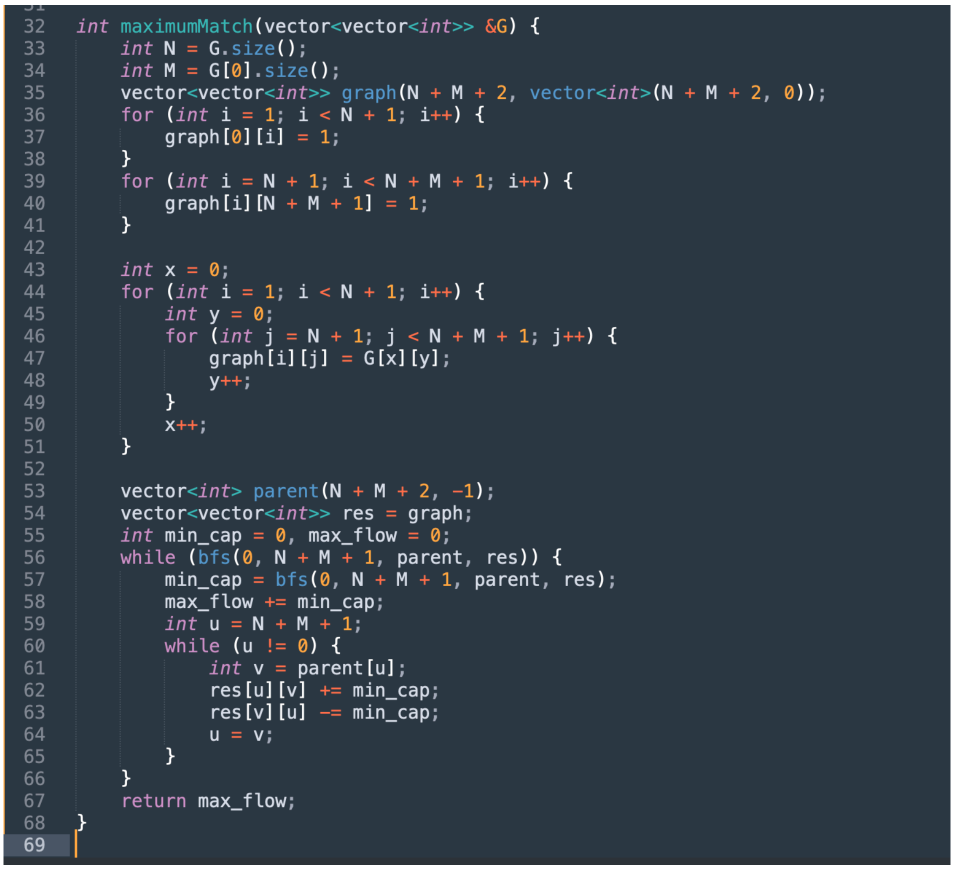
We convert the problem into a max flow problem as follows:
i)Create a source node 's' and connect it to all nodes in the first partition (left side) with edges of capacity 1.
ii)Create a sink node t and connect it to all nodes in the second partition (right side) with edges of capacity 1.
iii)Maintain the original edges' capacities.

Now apply the Edmonds Karp Algorithm

Code:



```
int bfs(int src, int sink, vector<int>& parent, vector<vector<int>>& res) {
    fill(parent.begin(), parent.end(), -1);
    int n = sink + 1;
    parent[src] = -2;
    queue<pair<int, int>> q;
    q.push({src, INT_MAX});

    while (!q.empty()) {
        int u = q.front().first;
        int capacity = q.front().second;
        q.pop();
        for (int it = 0; it < n; it++) {
            if (u != it && parent[it] == -1 && res[u][it] != 0) {
                parent[it] = u;
                int min_cap = min(capacity, res[u][it]);
                if (it == sink) {
                    return min_cap;
                }
                q.push({it, min_cap});
            }
        }
    }
    return 0;
}
```

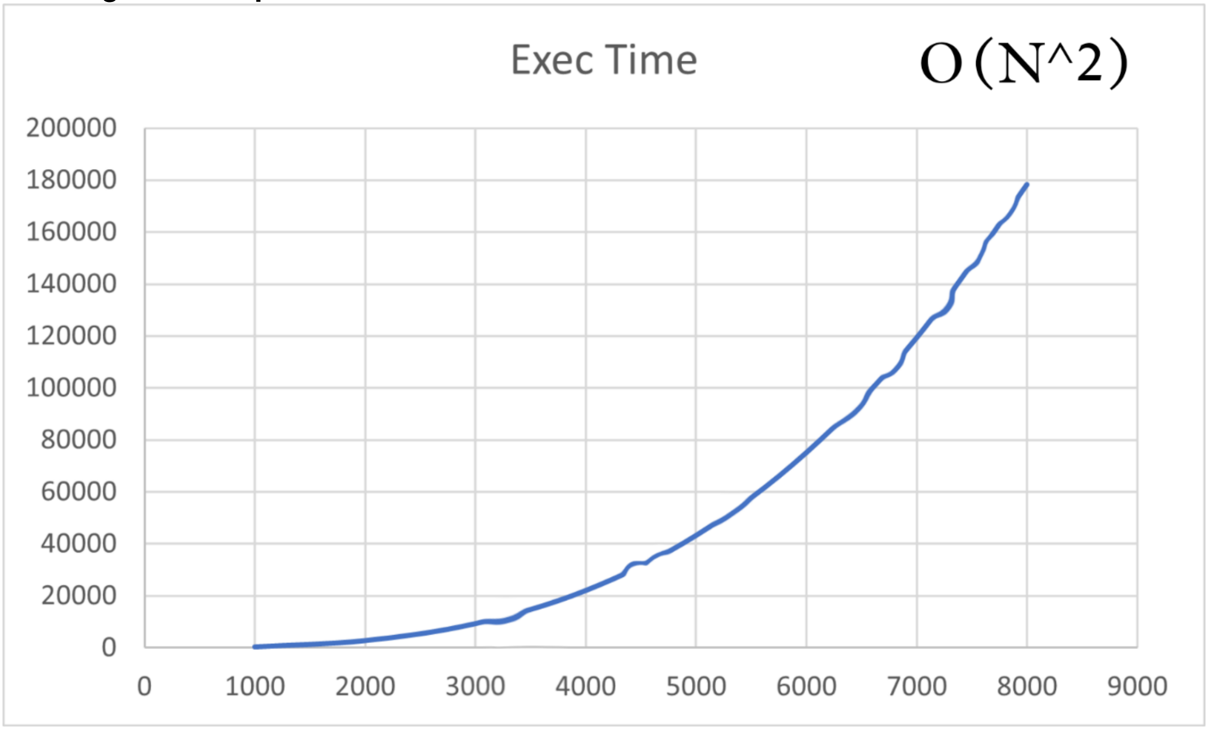


```cpp
int maximumMatch(vector<vector<int>> &G) {
    int N = G.size();
    int M = G[0].size();
    vector<vector<int>> graph(N + M + 2, vector<int>(N + M + 2, 0));
    for (int i = 1; i < N + 1; i++) {
        graph[0][i] = 1;
    }
    for (int i = N + 1; i < N + M + 1; i++) {
        graph[i][N + M + 1] = 1;
    }

    int x = 0;
    for (int i = 1; i < N + 1; i++) {
        int y = 0;
        for (int j = N + 1; j < N + M + 1; j++) {
            graph[i][j] = G[x][y];
            y++;
        }
        x++;
    }

    vector<int> parent(N + M + 2, -1);
    vector<vector<int>> res = graph;
    int min_cap = 0, max_flow = 0;
    while (bfs(0, N + M + 1, parent, res)) {
        min_cap = bfs(0, N + M + 1, parent, res);
        max_flow += min_cap;
        int u = N + M + 1;
        while (u != 0) {
            int v = parent[u];
            res[u][v] += min_cap;
            res[v][u] -= min_cap;
            u = v;
        }
    }
    return max_flow;
}

```

Analysis:

Let V be the number of vertices and E be the number of edges in the bipartite graph.
Converting Bipartite Graph to Flow Network:
Creating source and sink nodes: O(1)
Connecting source to left side and sink to right side:O(V) Total time complexity:O(V)
Edmonds-Karp algorithm:
Each iteration of the algorithm involves finding an augmenting path and updating the flow.
Each augmenting path can be found in O(E) using BFS or DFS.
Maximum number of iterations can be O(VE) (worst-case scenario).
Updating the flow takes O(E).
Total time complexity: O(VE^2) using BFS or O(V^2E) using DFS (with unit capacities).
Finding Maximum Flow: After running the algorithm, the maximum flow is known in O(1) time.
Recovering the Matching:Determining the matching from the flow can be done in O(E) time.
Overall Time Complexity:O(V) (Convert) + O(VE^2) (FF with BFS) orO(V^2E) (FF with DFS) + O(E) (Recover) = **O(VE^2) or O(V^2E).**

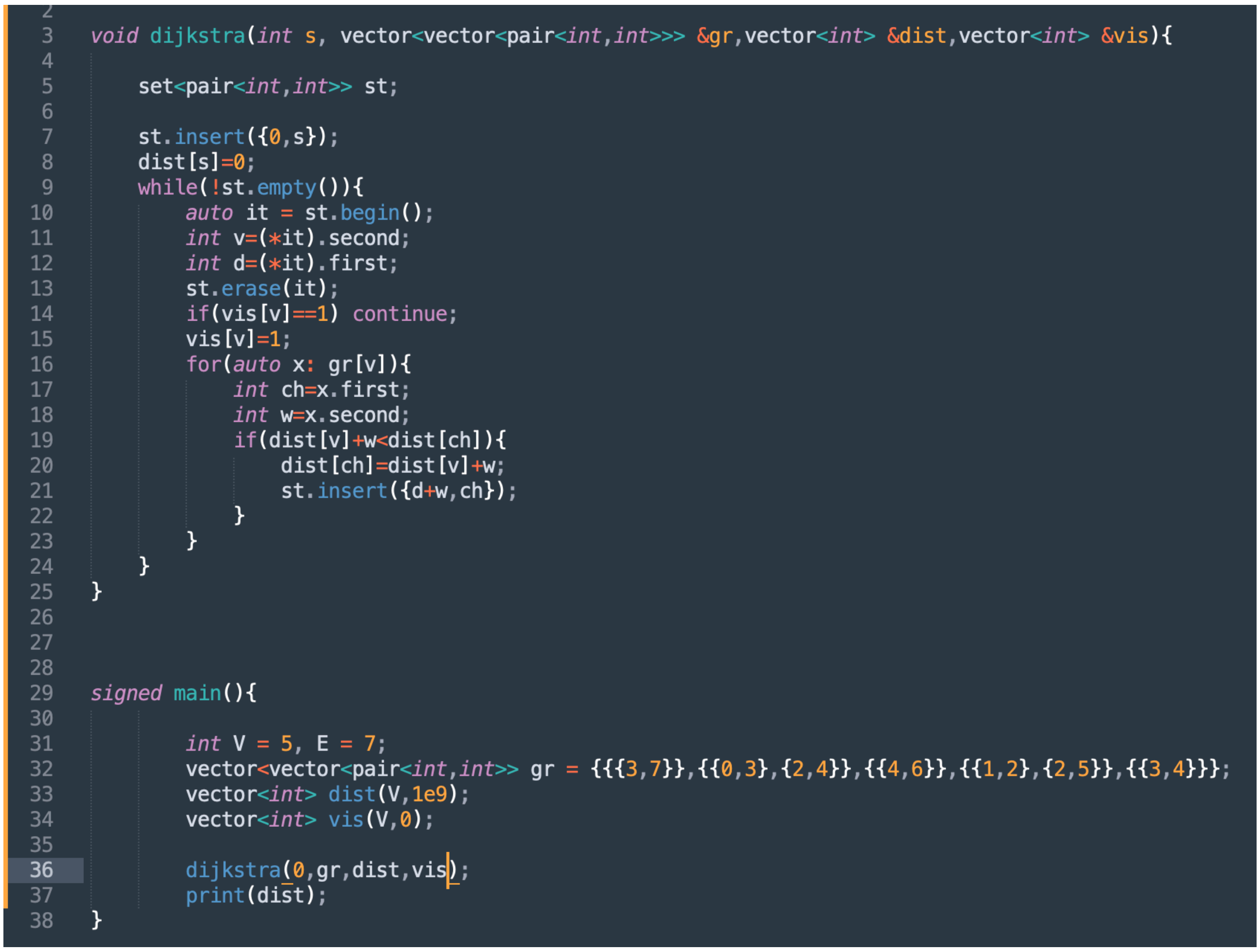
Running Time Graph:



Q2. **Dijkstra's Algorithm**

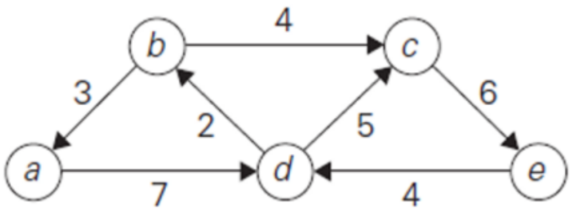
Design:
Apply Dijkstra's Single Source Shortest Path Algorithm as there are no negative weights.

Code:



```
void dijkstra(int s, vector<vector<pair<int,int>>> &gr,vector<int> &dist,vector<int> &vis){

    set<pair<int,int>> st;

    st.insert({0,s});
    dist[s]=0;
    while(!st.empty()){
        auto it = st.begin();
        int v=(*it).second;
        int d=(*it).first;
        st.erase(it);
        if(vis[v]==1) continue;
        vis[v]=1;
        for(auto x: gr[v]){
            int ch=x.first;
            int w=x.second;
            if(dist[v]+w<dist[ch]){
                dist[ch]=dist[v]+w;
                st.insert({d+w,ch});
            }
        }
    }
}



signed main(){

        int V = 5, E = 7;
        vector<vector<pair<int,int>> gr = {{{3,7}},{{0,3},{2,4}},{{4,6}},{{1,2},{2,5}},{{3,4}}};
        vector<int> dist(V,1e9);
        vector<int> vis(V,0);

        dijkstra(0,gr,dist,vis);
        print(dist);
}
```

Given Example:



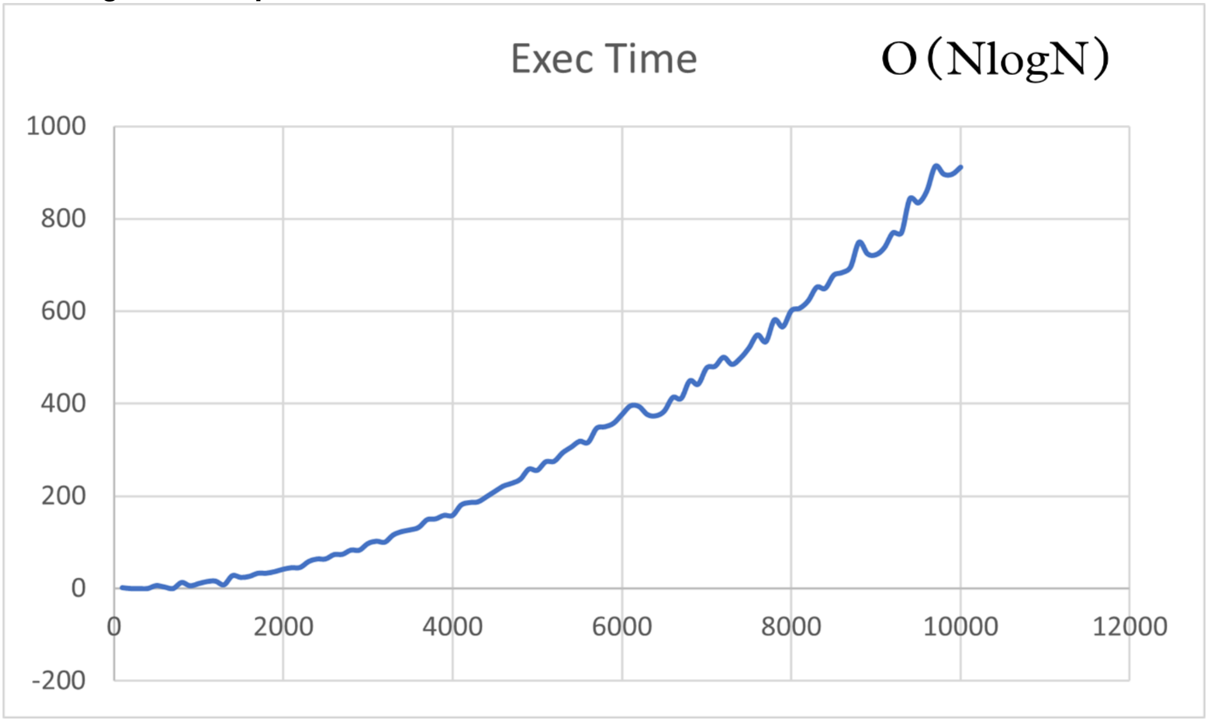
Answer:

Dist = {0, 9, 12, 7, 18}

Analysis:

Let V be the number of vertices and E be the number of edges in the graph.
Initialization: Initializing the distance array: O(V).
Priority Queue Operations:Inserting and extracting the minimum element from the priority queue: O(logV) per operation.
Total complexity for all operations: O(ElogV) (since each vertex can be inserted and extracted at most once).
Main Loop:The main loop runs E times (once for each edge).
Relaxation:For each edge, relaxation takes constant time.
Overall Time Complexity:
The overall time complexity of Dijkstra's algorithm is dominated by the priority queue operations, resulting in **O(V+ElogV) complexity.**
Space Complexity:The space complexity is **O(V+E)** to store the graph (adjacency list) and the priority queue.

Running Time Graph:

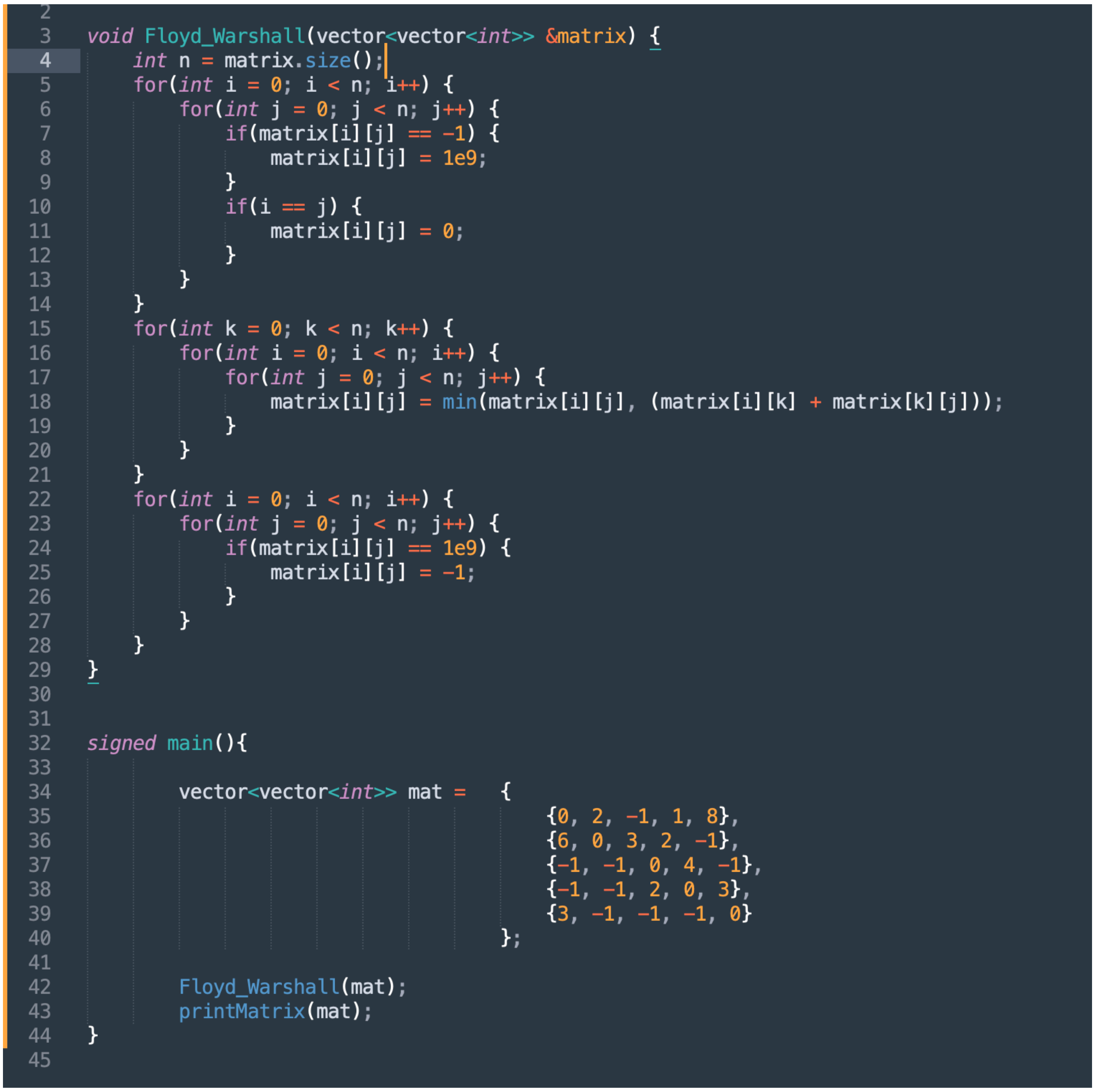


Q3. Floyd Warshall Algorithm

Design:

For each vertex k from 1 to n, representing an intermediate vertex: For each pair of vertices i and j, update dist[i][j] as the minimum of:The current value of dist[i][j], the sum of the distances from vertex i to vertex k and from vertex k to vertex j.

Code:



```cpp

void Floyd_Warshall(vector<vector<int>> &matrix) {
    int n = matrix.size();
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
            if(matrix[i][j] == -1) {
                matrix[i][j] = 1e9;
            }
            if(i == j) {
                matrix[i][j] = 0;
            }
        }
    }
    for(int k = 0; k < n; k++) {
        for(int i = 0; i < n; i++) {
            for(int j = 0; j < n; j++) {
                matrix[i][j] = min(matrix[i][j], (matrix[i][k] + matrix[k][j]));
            }
        }
    }
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
            if(matrix[i][j] == 1e9) {
                matrix[i][j] = -1;
            }
        }
    }
}


signed main(){

        vector<vector<int>> mat =   {
                                        {0, 2, -1, 1, 8},
                                        {6, 0, 3, 2, -1},
                                        {-1, -1, 0, 4, -1},
                                        {-1, -1, 2, 0, 3},
                                        {3, -1, -1, -1, 0}
                                    };

        Floyd_Warshall(mat);
        printMatrix(mat);
}

```

Given Example:

$$\begin{bmatrix} 0 & 2 & \infty & 1 & 8 \\ 6 & 0 & 3 & 2 & \infty \\ \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 3 & \infty & \infty & \infty & 0 \end{bmatrix}$$

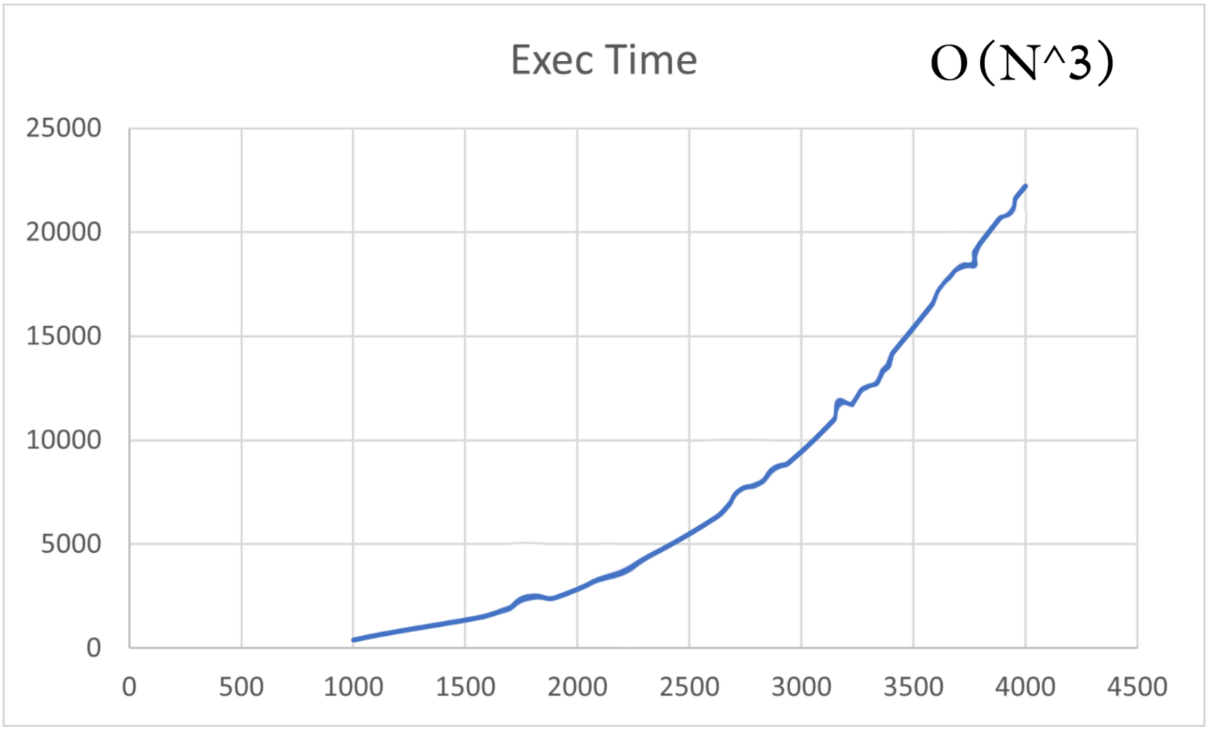
Answer:

```
{0   2    3   1   4}
{6    0   3   2   5}
{10 12 0    4   7}
{6    8   2   0   3}
{3    5   6   4   0}
```

Analysis:

Let n be the number of vertices in the graph.
1)**Initialization**:Initializing the distance matrix: O(n^2).
2)**Floyd-Warshall Algorithm:**
The algorithm consists of three nested loops, each running n times.
The time complexity of updating each entry in the distance matrix is O(1) since it involves simple comparisons and additions.
Total complexity:O(n^3).
Overall Time Complexity:O(n^2) (Initialization) +O(n^3) (Floyd-Warshall Algorithm) = O(n^3).
Space Complexity:The space complexity is O(n^2) to store the distance matrix.

Running Time Graph:

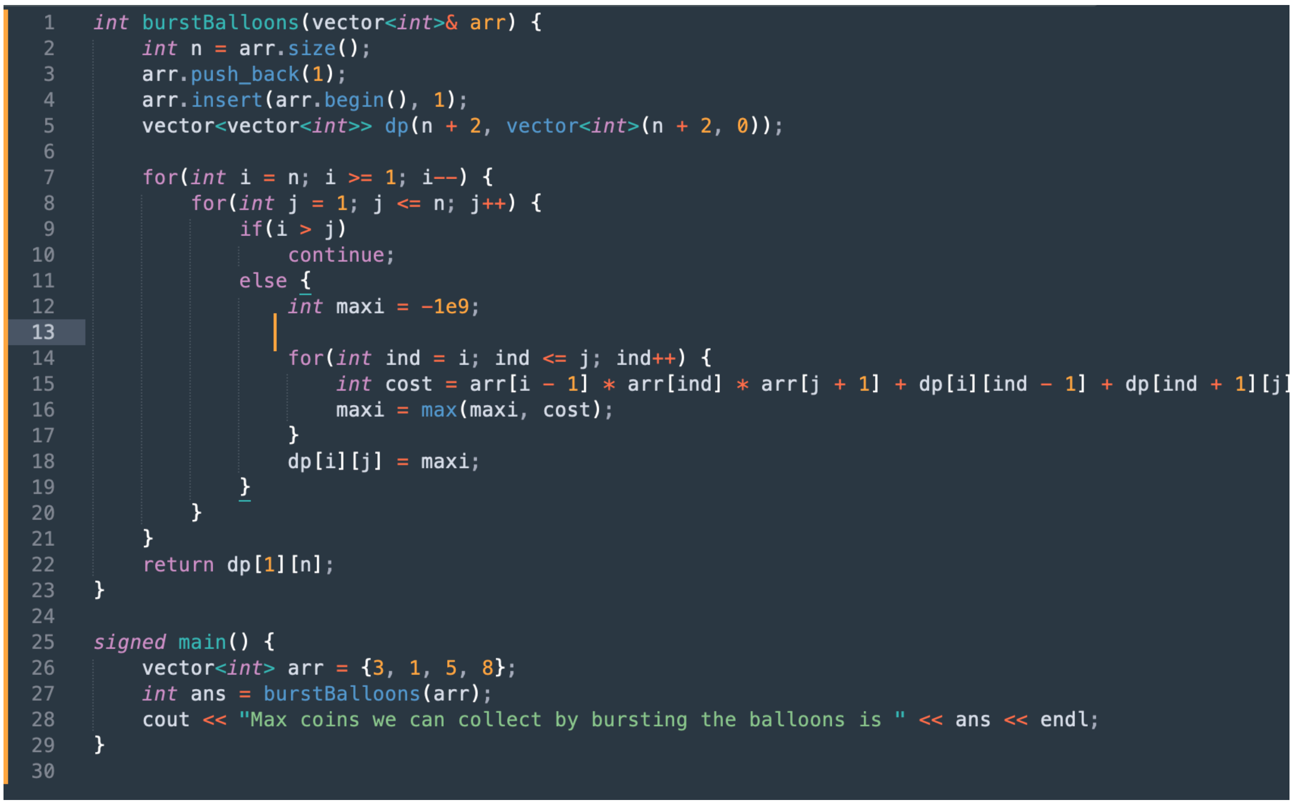


Q4. **Burst Balloons**

Design:

Algorithm: **Matrix Chain Multiplication Variant**
Algo:
Define a recursive function solve(i, j, v, memo) that takes the start index i, end index j, the list of balloon values v, and a memoization table memo.
1) Base Cases: If i>j, return 0 (no balloons left to burst).
If memo[i][j] is not equal to -1 (indicating the subproblem has been solved before), return the value stored in memo[i][j].
2) Initialize a variable temp to store the maximum score obtained by bursting balloons between indices i and j, initially set to INT_MIN.
3) Iterate from index i to index j: Burst the balloon at index k.
Recursively calculate the scores for bursting balloons to the left of k and to the right of k. Update temp with the maximum of its current value and the score obtained by bursting the current balloon plus the scores obtained by bursting balloons to the left and right of k.
4) Memoize the result by storing it in memo[i][j]. Return temp, which represents the maximum score obtained by bursting balloons between indices i and j.
5) In the burstballons function:
i)Preprocess the input vector v by adding 1 at the
beginning and end to simulate balloons that cannot be burst.
ii) Initialize a memoization table memo of size (n+2)×(n+2) and fill it with -1.
iii) Call the solve function with the appropriate parameters. Return the maximum
score obtained.

Code:



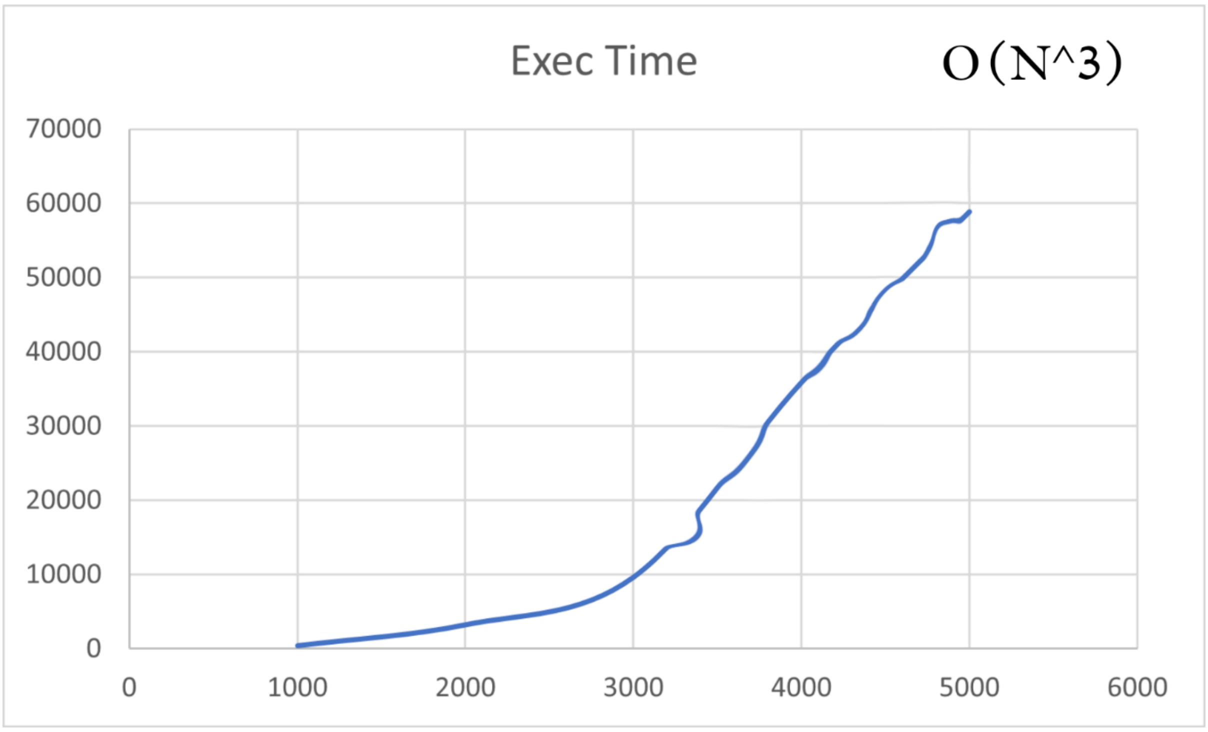
```cpp
int burstBalloons(vector<int>& arr) {
    int n = arr.size();
    arr.push_back(1);
    arr.insert(arr.begin(), 1);
    vector<vector<int>> dp(n + 2, vector<int>(n + 2, 0));

    for(int i = n; i >= 1; i--) {
        for(int j = 1; j <= n; j++) {
            if(i > j)
                continue;
            else {
                int maxi = -1e9;

                for(int ind = i; ind <= j; ind++) {
                    int cost = arr[i - 1] * arr[ind] * arr[j + 1] + dp[i][ind - 1] + dp[ind + 1][j]
                    maxi = max(maxi, cost);
                }
                dp[i][j] = maxi;
            }
        }
    }
    return dp[1][n];
}

signed main() {
    vector<int> arr = {3, 1, 5, 8};
    int ans = burstBalloons(arr);
    cout << "Max coins we can collect by bursting the balloons is " << ans << endl;
}
```

Given Example:
Example :
Input: arr = [3,1,5,8] , Output: 167
Input: arr = [1,5]   , Output: 10

Analysis:

Let n be the number of balloons.
1)Preprocessing: Adding 1 at the beginning and end of the input vector v: O(1).
2)Memoization Table Initialization: Initializing a 2D memoization table of size (n+2)×(n+2) with -1: O(n^2).
3)Recursive Function solve:
The function is called for each subproblem once.
The time complexity for each subproblem calculation is O(n) due to the loop from i to j.
Total time complexity: O(n^3).
Overall Time Complexity:O(n) (Preprocessing) + O(n^2) (Memoization Table Initialization) + O(n^3) (Recursive Function) = **O(n^3).**
Space Complexity:The space complexity is **O(n^2)** to store the memoization table.

Running Time Graph:



Q5. **Trapping Rainwater**

Design:

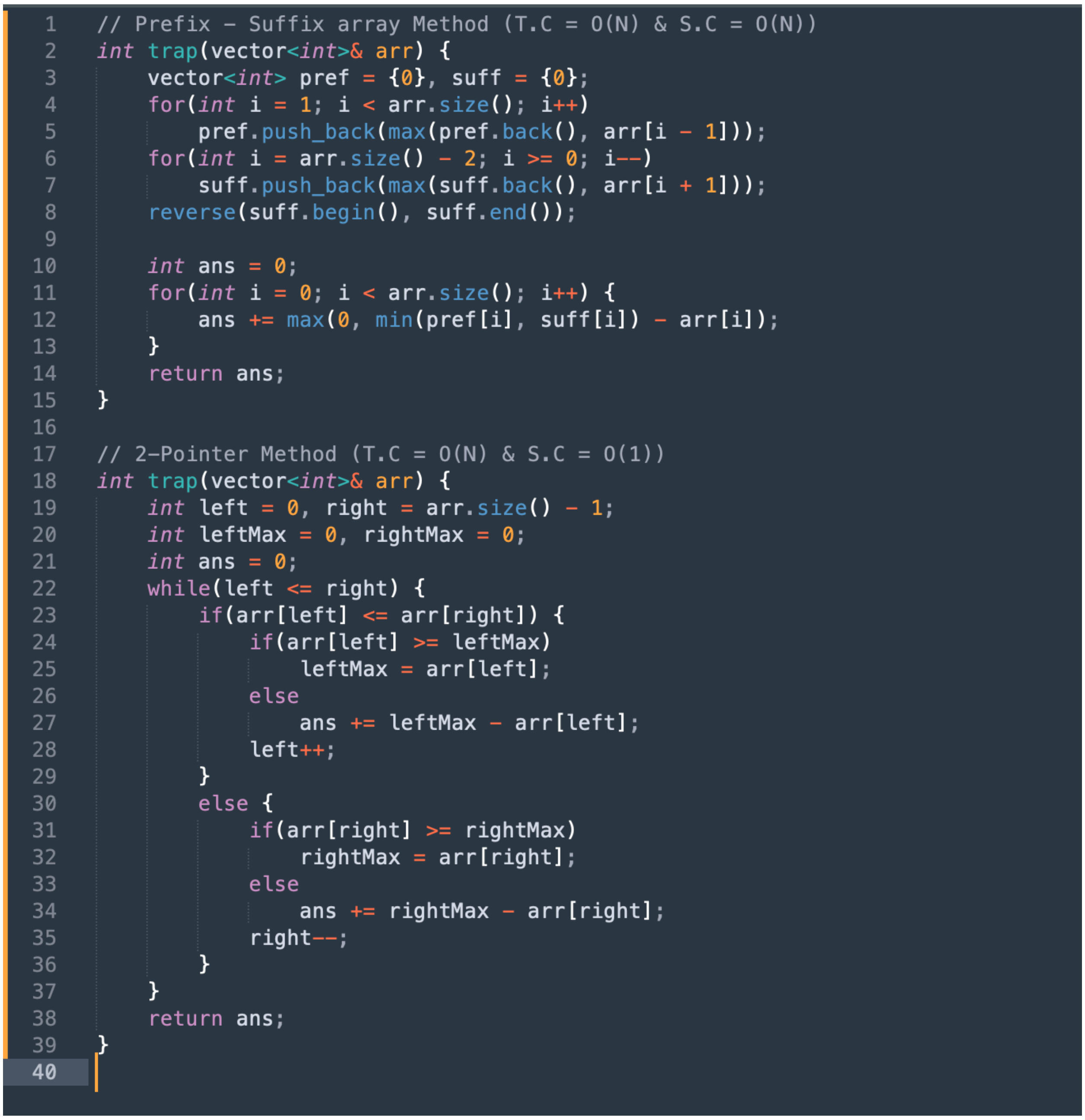
Algorithm:
SubOptimal(Using Prefix and Suffix Arrays):
1)Initialize two arrays lh (left heights) and rh (right heights) of size n, where n is the size of the input array heights.
2)Traverse the heights array from left to right and compute the maximum height encountered so far. Store this maximum height at each index in the lh array.
3)Traverse the heights array from right to left and compute the maximum height encountered so far. Store this maximum height at each index in the rh array.
4)Initialize a variable res to store the total trapped water, initially set to 0.

5)Traverse the heights array:
i)For each index i, compute the minimum of lh[i] and rh[i]. ii)Subtract the height of the current bar from this minimum. iii)Add this value to res.
iv)Return res, which represents the total trapped water.

Optimal(Using 2 pointer approach)(Decreases Space Complexity):

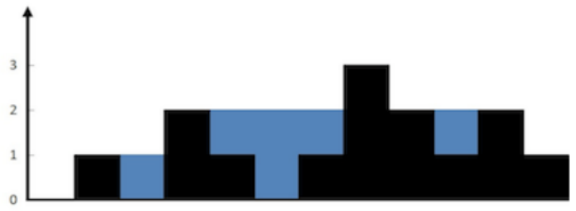
1. Take 2 pointers l(left pointer) and r(right pointer) pointing to 0th and (n-1)th index respectively.
2. Take two variables leftMax and rightMax and initialize them to 0. If height[l] is less than or equal to height[r] then if leftMax is less than height[l] update leftMax to height[l] else add leftMax-height[l] to your final answer and move the l pointer to the right i.e l++. If height[r] is less than height[l], then now we are dealing with the right block. If height[r] is greater than rightMax, then update rightMax to height[r] else add rightMax-height[r] to the final answer. Now move r to the left. Repeat these steps till l and r crosses each other.

Code:



```cpp
// Prefix - Suffix array Method (T.C = O(N) & S.C = O(N))
int trap(vector<int>& arr) {
    vector<int> pref = {0}, suff = {0};
    for(int i = 1; i < arr.size(); i++)
        pref.push_back(max(pref.back(), arr[i - 1]));
    for(int i = arr.size() - 2; i >= 0; i--)
        suff.push_back(max(suff.back(), arr[i + 1]));
    reverse(suff.begin(), suff.end());

    int ans = 0;
    for(int i = 0; i < arr.size(); i++) {
        ans += max(0, min(pref[i], suff[i]) - arr[i]);
    }
    return ans;
}

// 2-Pointer Method (T.C = O(N) & S.C = O(1))
int trap(vector<int>& arr) {
    int left = 0, right = arr.size() - 1;
    int leftMax = 0, rightMax = 0;
    int ans = 0;
    while(left <= right) {
        if(arr[left] <= arr[right]) {
            if(arr[left] >= leftMax)
                leftMax = arr[left];
            else
                ans += leftMax - arr[left];
            left++;
        }
        else {
            if(arr[right] >= rightMax)
                rightMax = arr[right];
            else
                ans += rightMax - arr[right];
            right--;
        }
    }
    return ans;
}
```

Given Example:



Input: arr = [0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1],
Output: 6

Input: arr = [4, 2, 0, 3, 2, 5]
Output: 9

Analysis:
Prefix-Suffix Method:
Time Complexity:
- O(N) for both prefix and suffix array construction (one loop each iterating N times).
- O(N) for water calculation (one loop iterating N times).
- The total time complexity is dominated by the linear loops, resulting in O(N).

Space Complexity:
- O(N) for prefix array (`pref`).
- O(N) for suffix array (`suff`).
- The total space complexity is O(N) due to the additional arrays used.

2-Pointer Approach:

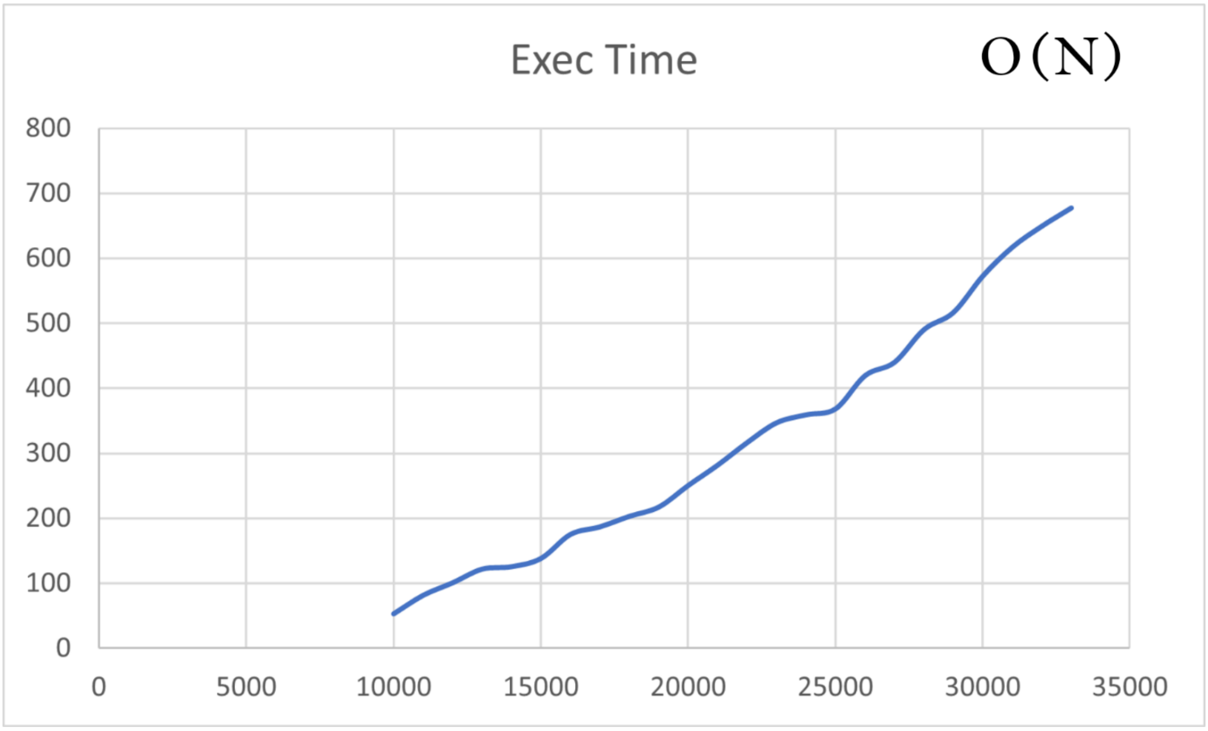
Time Complexity:
- It uses a single loop that iterates through the array at most twice (once from left to right and potentially another from right to left).

Hence Time Complexity: O(N)

Space Complexity:
- O(1) - It only uses constant extra space for variables like `left`, `right`, `leftMax`, `rightMax`, and `ans`.

Hence Space Complexity: O(1)

Running Time Graph:


Exec Time
O(N)