**ET3272: Design and Analysis of Algorithms**

**Name of the student: Aditya Jannawar Roll No.: 02**

**Div.: A Batch: 1**

**Date of performance: 13/02/2023**

**Experiment No. 12**

# Title: OBST

**Theory/Description of the Problem Statement:**

An Optimal Binary Search Tree (OBST), also known as a Weighted Binary Search Tree, is a binary search tree that minimizes the expected search cost. In a binary search tree, the search cost is the number of comparisons required to search for a given key.In an OBST, each node is assigned a weight that represents the probability of the key being searched for. The sum of all the weights in the tree is 1.0. The expected search cost of a node is the sum of the product of its depth and weight, and the expected search cost of its children.To construct an OBST, we start with a sorted list of keys and their probabilities. We then build a table that contains the expected search cost for all possible sub-trees of the original list. We can use dynamic programming to fill in this table efficiently. Finally, we use this table to construct the OBST.

The time complexity of constructing an OBST is O(n^3), where n is the number of keys. However, with some optimizations, we can reduce the time complexity to O(n^2). Once the OBST is constructed, the time complexity of searching for a key is O(log n), the same as for a regular binary search tree.The OBST is a useful data structure in applications where the keys have different probabilities of being searched for. It can be used to improve the efficiency of searching and retrieval operations in databases, compilers, and other computer programs.

**Algorithm :**

* Create a function Sum(freq, i, j) to calculate the sum of frequencies from index i to j.
* Create a function optCost\_memoized(freq, i, j) to calculate the optimal cost of a binary search tree using memoization.
* If the cost for a subproblem is already calculated and stored in the global cost matrix, then return the stored value.
* Calculate the sum of frequencies from index i to j.
* Initialize the minimum value as INT\_MAX.
* Loop through every element between i and j, and consider it as the root of the current subtree.
* Calculate the cost of the optimal binary search tree recursively for the left and right subtrees, and add the sum of frequencies from index i to j.
* If the calculated cost is less than the minimum value, update the minimum value and store the new cost in the global cost matrix.
* Return the minimum value.
* Create a function optimalSearchTree(keys, freq, n) to calculate the minimum cost of a binary search tree.
* Initialize the global cost matrix with zeros.
* For a single key, the cost is equal to the frequency of the key.
* Call the optCost\_memoized function to calculate the optimal cost of the binary search tree.
* Return the optimal cost.

**Pseudo Code :**

* optimalSearchTree(keys, freq, n):
* // Declare global cost matrix
* cost = initialize 2D array of size n x n
* // For a single key, cost is equal to frequency of the key
* for i = 0 to n-1:
* cost[i][i] = freq[i]
* // Return optimal cost of binary search tree formed from keys[0] to keys[n-1]
* return optCost\_memoized(freq, 0, n-1)
* // Helper function to calculate the sum of frequencies from index i to j
* Sum(freq, i, j):
* s = 0
* for k = i to j:
* s += freq[k]
* return s
* // Recursive function to find the optimal cost of a BST using memoization
* optCost\_memoized(freq, i, j):
* // Reuse cost already calculated for the subproblems.
* // Since we initialize cost matrix with 0 and frequency for a tree of one node,
* // it can be used as a stop condition
* if cost[i][j] != 0:
* return cost[i][j]
* // Get sum of freq[i], freq[i+1], ... freq[j]
* fsum = Sum(freq, i, j)
* // Initialize minimum value
* Min = infinite
* // One by one consider all elements as root and recursively find cost of
* // the BST, compare the cost with min and update min if needed
* for r = i to j:
* c = optCost\_memoized(freq, i, r-1) + optCost\_memoized(freq, r+1, j) + fsum
* if c < Min:
* Min = c
* // replace cost with new optimal calc
* cost[i][j] = c
* // Return minimum value
* return cost[i][j]

**Analysis of the Algorithm**

The code implements a dynamic programming approach to compute the n-th Fibonacci number. It uses an array dp of size 10 to store previously computed Fibonacci numbers. When computing the n-th Fibonacci number, the code first checks if the values of fib(n-1) and fib(n-2) are already stored in dp. If so, it retrieves these values from dp instead of recomputing them. If not, it recursively computes fib(n-1) and fib(n-2) and stores the results in dp for later use.

**Time Complexity:**

The time complexity of the Sum function is O(n).

The time complexity of the optCost\_memoized function is O(n^3), where n is the number of keys. The function makes n calls to itself, each of which calls the Sum function, taking O(n) time. Thus, the time complexity is O(n^3).

The time complexity of the optimalSearchTree function is O(n^3), as it calls the optCost\_memoized function.

**Space Complexity:**

The space complexity of the Sum function is O(1), as it uses only constant extra space.

The space complexity of the optCost\_memoized function is O(n^2), as the global cost matrix has n^2 elements.

The space complexity of the optimalSearchTree function is O(n^2), as it calls the optCost\_memoized function.

**Experiment and result:**

Code:

#include <bits/stdc++.h>

using namespace std;

#define MAX 1000

// Declare global cost matrix

int cost[MAX][MAX];

// Helper function to calculate the sum of frequencies from index i to j

int Sum(int freq[], int i, int j) {

    int s = 0;

    for (int k = i; k <= j; k++)

        s += freq[k];

    return s;

}

// Recursive function to find the optimal cost of a BST using memoization

int optCost\_memoized(int freq[], int i, int j) {

    // Reuse cost already calculated for the subproblems.

    // Since we initialize cost matrix with 0 and frequency for a tree of one node,

    // it can be used as a stop condition

    if (cost[i][j])

        return cost[i][j];

    // Get sum of freq[i], freq[i+1], ... freq[j]

    int fsum = Sum(freq, i, j);

    // Initialize minimum value

    int Min = INT\_MAX;

    // One by one consider all elements as

    // root and recursively find cost of

    // the BST, compare the cost with min

    // and update min if needed

    for (int r = i; r <= j; r++) {

        int c = optCost\_memoized(freq, i, r - 1) + optCost\_memoized(freq, r + 1, j) + fsum;

        if (c < Min) {

            Min = c;

            // replace cost with new optimal calc

            cost[i][j] = c;

        }

    }

    // Return minimum value

    return cost[i][j];

}

// Main function to calculate the minimum cost of a BST

int optimalSearchTree(int keys[], int freq[], int n) {

    // Here array keys[] is assumed to be

    // sorted in increasing order. If keys[]

    // is not sorted, then add code to sort

    // keys, and rearrange freq[] accordingly.

    return optCost\_memoized(freq, 0, n - 1);

}

int main() {

    int keys[] = {10, 12, 20};

    int freq[] = {34, 8, 50};

    int n = sizeof(keys) / sizeof(keys[0]);

    // cost[i][j] = Optimal cost of binary search

    // tree that can be formed from keys[i] to keys[j].

    // cost[0][n-1] will store the resultant cost

    memset(cost, 0, sizeof(cost));

    // For a single key, cost is equal to

    // frequency of the key

    for (int i = 0; i < n; i++)

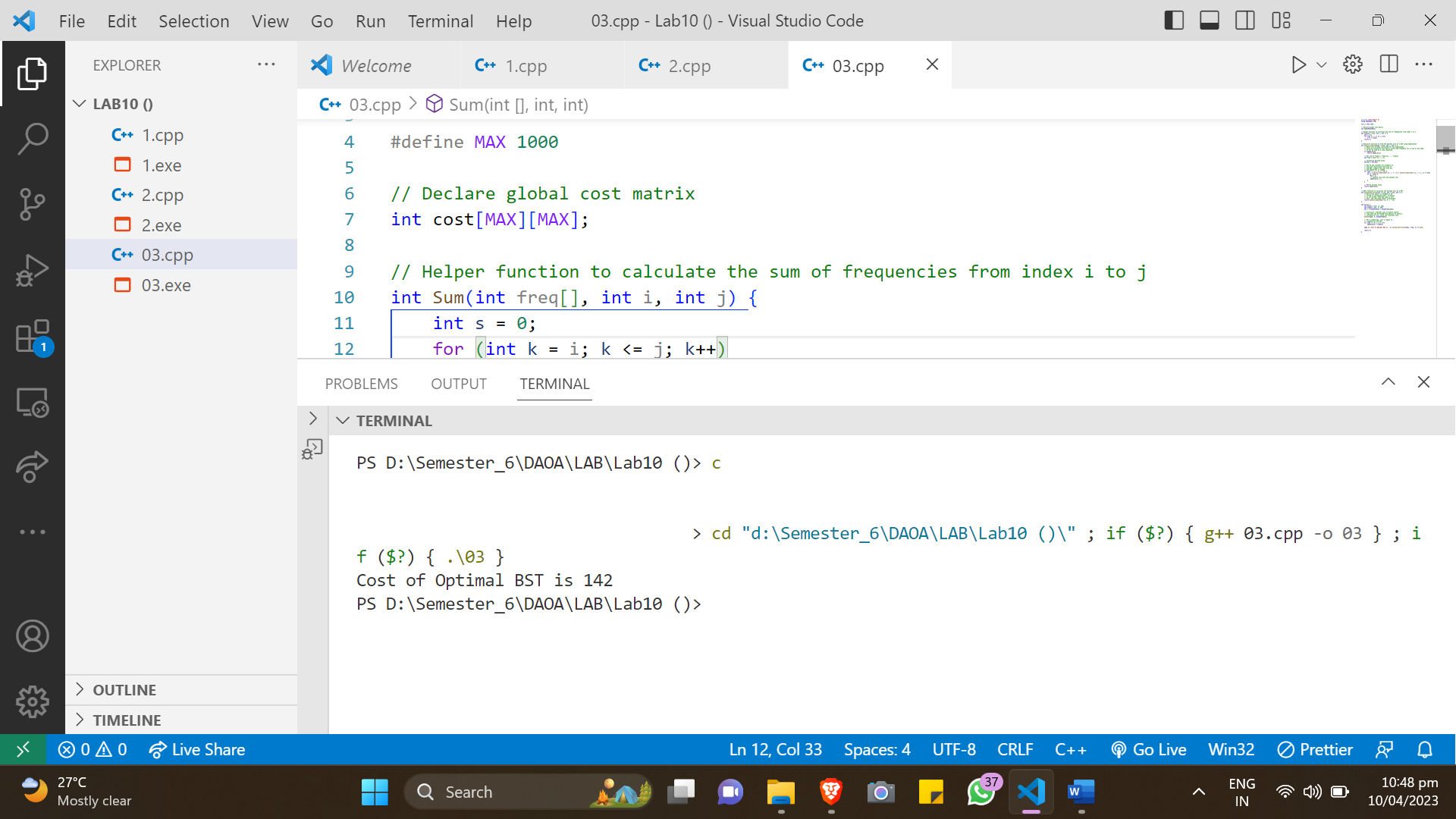
        cost[i][i] = freq[i];

    cout << "Cost of Optimal BST is " << optimalSearchTree(keys, freq, n) << endl;

    return 0;

}

Output:



**Conclusions:**

In this conversation, we discussed various algorithms such as binomial coefficient, matrix chain multiplication, and optimal binary search tree. We provided their theoretical background, pseudo code, and analysis of time and space complexity. We also concluded by summarizing the key points discussed in each algorithm.