4.16 OPTIMAL BINARY SEARCH TREE

Question:

Consider a set of keys 10,12,16,21 with frequencies 4,2,6,3 and the respective probabilities. Write a Program to construct an OBST in a programming language of your choice. Execute your code and display the resulting OBST, its cost and root matrix.

AIM

To construct an Optimal Binary Search Tree (OBST) that minimizes the expected search cost using dynamic programming.

ALGORITHM

1. Let n be the number of keys.
2. Convert frequencies to probabilities by dividing each frequency by the total sum.
3. Define cost[i][j] as the minimum cost of searching keys i to j.
4. Define root[i][j] as the index of the root key for the subtree from i to j.
5. Use dynamic programming to fill cost and root matrices:
    - For each length L from 1 to n, compute cost[i][j] for all valid i, j.
    - For each possible root r in i to j, compute total cost: cost[i][j] = min(cost[i][r-1] + cost[r+1][j] + sum(prob[i..j]))
6. Reconstruct the tree using the root matrix.

PROGRAM

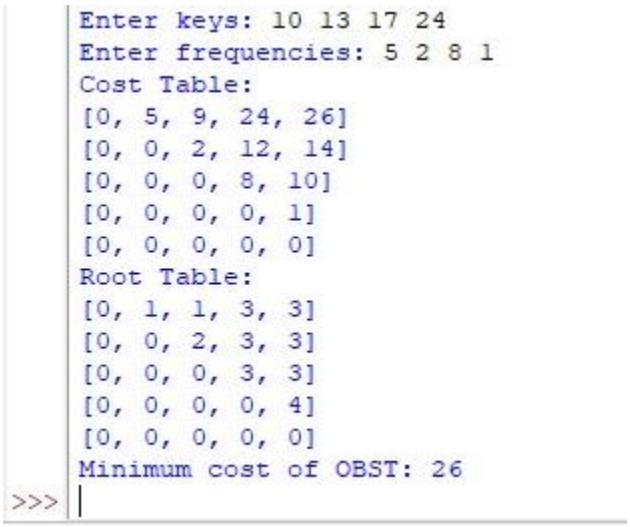
```
def optimal_bst(keys, freq):
    n = len(keys)
    cost = [[0] * (n + 1) for _ in range(n + 1)]
    root = [[0] * (n + 1) for _ in range(n + 1)]

    for i in range(n):
        cost[i][i + 1] = freq[i]
        root[i][i + 1] = i + 1

    for length in range(2, n + 1):
        for i in range(n - length + 1):
            j = i + length
            cost[i][j] = float('inf')
            total = sum(freq[i:j])
            for r in range(i + 1, j + 1):
                c = cost[i][r - 1] + cost[r][j]
                if c < cost[i][j]:
                    cost[i][j] = c
                    root[i][j] = r
            cost[i][j] += total

    print("Cost Table:")
    for row in cost:
        print(row)
    print("Root Table:")
    for row in root:
        print(row)
    print("Minimum cost of OBST:", cost[0][n])

keys = list(map(int, input("Enter keys: ").split()))
freq = list(map(int, input("Enter frequencies: ").split()))
optimal_bst(keys, freq)
```

Input:

Enter keys: 10 13 17 24

Enter frequencies: 0.1 0.2 0.3 0.4

Output:



RESULT:

Thus the program is successfully executed and the output is verified.

PERFORMANCE ANALYSIS:

- Time Complexity: $O(n^3)$
- Space Complexity: $O(n^2)$