EXPERIMENT NO. 07

Roll No.: 24141001
Batch: I1

Title: Optimal Binary Search trees using Dynamic Programming.

Objectives:

- Understand the concept of Binary Search Trees (BST) and how search cost varies based on tree structure.
- Solve the Optimal Binary Search Tree (OBST) problem using Dynamic Programming.
- Minimize the expected search cost using given key frequencies and dummy keys.
- Implement the OBST algorithm in C.
- Analyze OBST's time and space complexity.

Theory:

A Binary Search Tree (BST) allows efficient searching, insertion and deletion.
However, the search cost depends on how balanced the tree is.

In many applications (e.g., compilers, dictionaries), each key has a probability of being searched.
A poorly structured BST leads to high search time.

Optimal Binary Search Tree (OBST)

Given:

- Keys: K1, K2, …, Kn (sorted)
- Successful search probabilities: p1, p2, …, pn
- Unsuccessful search probabilities: q0, q1, …, qn

Goal:
Construct a BST that minimizes the expected cost of searching.

Dynamic Programming is used because:

- The structure has optimal substructure
- Overlapping subproblems exist
- A brute-force solution is exponential

OBST uses three DP tables:

- Cost[i][j] → Minimum cost between keys i to j
- Weight[i][j] → Total frequency (pi + qi) between i and j
- Root[i][j] → Root key index producing minimum cost

Algorithm:

```
OBST(p[], q[], n):

1. Create matrices: cost[n+1][n+1], weight[n+1][n+1], root[n+1][n+1]

2. For i = 0 to n:
      cost[i][i] = q[i]
      weight[i][i] = q[i]

3. For length = 1 to n:
      For i = 0 to n - length:
         j = i + length

         weight[i][j] = weight[i][j-1] + p[j] + q[j]
         cost[i][j] = ∞

         For r = i+1 to j:
            temp = cost[i][r-1] + cost[r][j] + weight[i][j]
            If temp < cost[i][j]:
               cost[i][j] = temp
               root[i][j] = r

4. Return cost[0][n] and root[][]
```

Program:

```
#include <stdio.h>
#include <limits.h>
```

```
int main() {
    int n, i, j, k;
    float p[20], q[20];
    float cost[20][20], weight[20][20];

    printf("Enter number of keys: ");
    scanf("%d", &n);

    printf("Enter successful search probabilities p[i]:\n");
    for (i = 1; i <= n; i++)
        scanf("%f", &p[i]);

    printf("Enter unsuccessful search probabilities q[i]:\n");
    for (i = 0; i <= n; i++)
        scanf("%f", &q[i]);

    // Initialize cost and weight
    for (i = 0; i <= n; i++) {
        cost[i][i] = q[i];
        weight[i][i] = q[i];
    }

    // OBST using Dynamic Programming
    for (int length = 1; length <= n; length++) {
        for (i = 0; i <= n - length; i++) {
            j = i + length;
            cost[i][j] = INT_MAX;
            weight[i][j] = weight[i][j - 1] + p[j] + q[j];

            for (k = i + 1; k <= j; k++) {
                float temp = cost[i][k - 1] + cost[k][j] + weight[i][j];
                if (temp < cost[i][j]) {
                    cost[i][j] = temp;
                }
            }
        }
    }

    printf("\nMinimum cost of Optimal BST = %.2f\n", cost[0][n]);
```

```
    return 0;
}
```

OUTPUT:

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

PS C:\Users\GCEK3\Desktop\New folder\output> cd 'c:\Users\GCEK3\Desktop\New folder\output\output'
PS C:\Users\GCEK3\Desktop\New folder\output\output> & .\'EXP7.exe'
Enter number of keys: 3
Enter successful search probabilities p[i]:
0.3
0.2
0.5
Enter unsuccessful search probabilities q[i]:
0.1 0.1 0.1 0.1

Minimum cost of Optimal BST = 3.00
PS C:\Users\GCEK3\Desktop\New folder\output\output>
```

Applications of OBST:

1. Compiler Design

Used in constructing optimal parsing tables and identifying common keywords efficiently.

2. Database Indexing

Minimizes search time for keys accessed with different frequencies.

3. File Systems

Used for directory lookup optimization.

4. Artificial Intelligence

Search optimization in decision trees.

5. Data Compression

Forms the basis for efficient symbol lookup.

Time and space complexity:
Time Complexity: $O(n^3)$
Space Complexity: $O(n^2)$

Conclusion:

The construction of Optimal Binary Search Trees using Dynamic Programming ensures the minimum expected search cost based on search frequencies. It is a powerful optimization method used in compilers, databases, and search applications where the frequency of accesses is known. Though the DP solution has high time complexity ($O(n^3)$), it provides the most optimal BST structure.