Experiment No.7

Name :-Shivani Kolekar

Roll No:-24141031

Batch:-I2

Title:- Tptimal binary search using dynamic Programming

Programm:-

```
#include <stdio.h>
#include <limits.h>   // For INT_MAX
#define MAX 20       // Maximum number of keys
// Function to find the Optimal BST
void optimalBST(float p[], float q[], int n) {
   float e[MAX + 1][MAX + 1];  // Expected cost
   float w[MAX + 1][MAX + 1];  // Weight sums
   int root[MAX + 1][MAX + 1]; // Root table
   // Initialize tables
   for (int i = 1; i <= n + 1; i++) {
      e[i][i - 1] = q[i - 1];
      w[i][i - 1] = q[i - 1];
   }
   // Build tables for increasing lengths
   for (int l = 1; l <= n; l++) {
      for (int i = 1; i <= n - l + 1; i++) {
         int j = i + l - 1;
         e[i][j] = INT_MAX;
         w[i][j] = w[i][j - 1] + p[j] + q[j];
```

```c
            for (int r = i; r <= j; r++) {
                float t = e[i][r - 1] + e[r + 1][j] + w[i][j];
                if (t < e[i][j]) {
                    e[i][j] = t;
                    root[i][j] = r;
                }
            }
        }
    }
    // Print results
    printf("\nMinimum expected search cost = %.4f\n", e[1][n]);
    printf("\nRoot matrix (showing optimal root for each subtree):\n");
    for (int i = 1; i <= n; i++) {
        for (int j = i; j <= n; j++) {
            printf("root[%d][%d] = %d\n", i, j, root[i][j]);
        }
    }
}
int main() {
    int n = 3; // number of keys
    // Probabilities of successful searches
    float p[] = {0, 0.15, 0.10, 0.05}; // 1-indexed (p[0] unused)
    // Probabilities of unsuccessful searches
    float q[] = {0.05, 0.10, 0.05, 0.05};
    printf("Optimal Binary Search Tree Problem\n");
    printf("----------------------------------\n");
```

```
    printf("Number of keys = %d\n", n);

    optimalBST(p, q, n);

    return 0;

}
```

Output:-

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

PS C:\Users\shiva> cd "c:\c language\" ; if ($?) { gcc DAAappl_1.c -o DAAappl_1 } ; if ($?) { .\DAAappl_1 }
Optimal Binary Search Tree Problem
----------------------------------
Number of keys = 3

Minimum expected search cost = 1.2500

Root matrix (showing optimal root for each subtree):
root[1][1] = 1
root[1][2] = 1
root[1][3] = 2
root[2][2] = 2
root[2][3] = 2
root[3][3] = 3
PS C:\c language>
```

Complexity

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

Applications of Optimal Binary Search Trees (OBST)

1. Efficient Searching in Static Databases

- When you know how frequently each key is accessed, an OBST minimizes the average search time.
- Example:
    - A dictionary or word list used in a spell-checker.
    - Searching for reserved words in a compiler (like if, else, for, etc.).
- If some keys (words) are looked up more often, OBST ensures they appear closer to the root, making searches faster on average.

---

2. Compiler Design – Keyword Lookup

- Compilers often need to recognize reserved keywords (e.g., int, while, return, if).
- Since some keywords appear more frequently than others, compilers use OBSTs to minimize the average number of comparisons during lexical analysis.

---

3. Data Compression and Encoding

- OBSTs can be used to assign shorter search paths to frequent symbols, similar in spirit to Huffman coding.
- Used when you have to decide a binary decision structure minimizing weighted cost.

---

4. Information Retrieval Systems

- In search engines or document databases, OBSTs can improve query response time when certain terms or files are accessed more frequently.

---

5. Optimal Storage and Access of Static Data

- When the data set does not change often (static), and access probabilities are known, OBSTs provide the most efficient tree structure for retrieval.
- Example:
    - Lookup tables in embedded systems.
    - Static routing tables or configuration tables.

---

6. Predictive Text and Autocomplete Systems

- When a predictive text model knows the probability of each word or phrase, an OBST can minimize the average number of comparisons or lookups required.

Conclusion:- The Optimal Binary Search Tree (OBST) uses Dynamic Programming to build a BST that minimizes the average search cost based on known access probabilities. It ensures faster lookups in static search applications like compilers and databases.