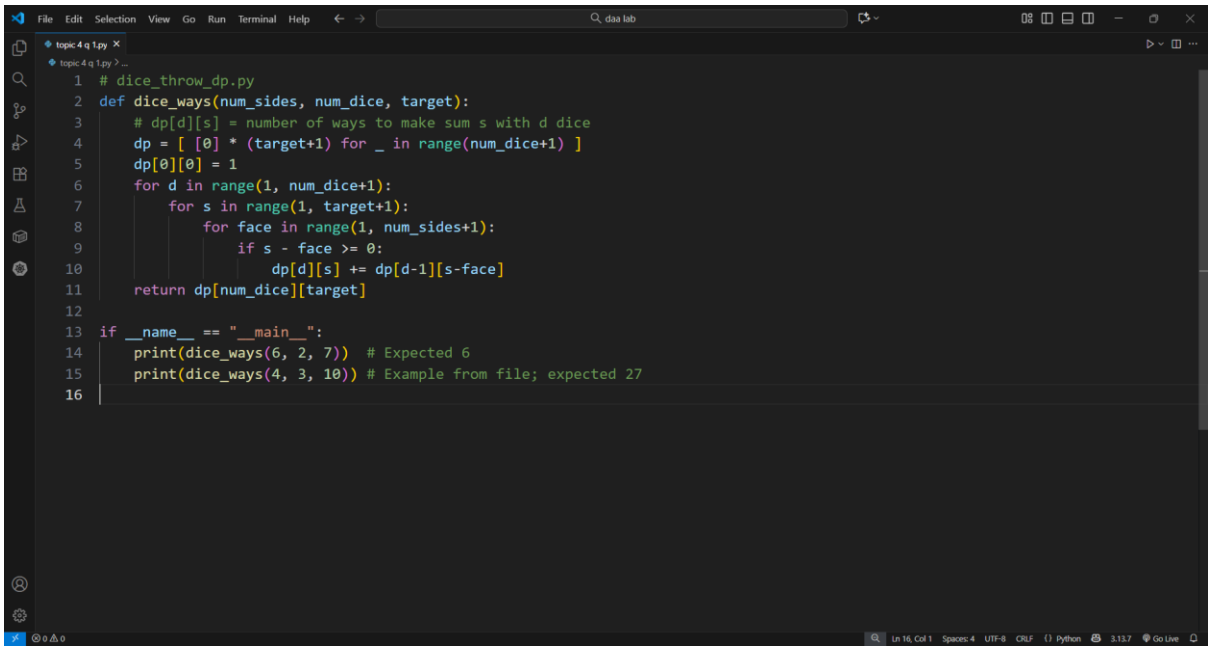
TOPIC 4 — DYNAMIC PROGRAMMING

Q1. Dice Throw Problem

Aim:
To find number of ways to reach a target sum using dynamic programming.
Algorithm:

1. Read `num_sides`, `num_dice`, `target`.
2. Initialize DP table.
3. Fill using recurrence: `dp[d][t] += dp[d-1][t-i]`.
4. Return `dp[num_dice][target]`.
5. Output total ways.

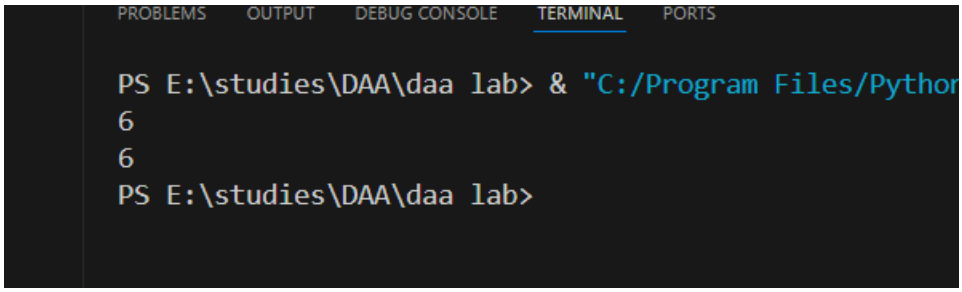
Code:



```python
# dice_throw_dp.py
def dice_ways(num_sides, num_dice, target):
    # dp[d][s] = number of ways to make sum s with d dice
    dp = [ [0] * (target+1) for _ in range(num_dice+1) ]
    dp[0][0] = 1
    for d in range(1, num_dice+1):
        for s in range(1, target+1):
            for face in range(1, num_sides+1):
                if s - face >= 0:
                    dp[d][s] += dp[d-1][s-face]
    return dp[num_dice][target]

if __name__ == "__main__":
    print(dice_ways(6, 2, 7))  # Expected 6
    print(dice_ways(4, 3, 10)) # Example from file; expected 27
```

Input: `num_sides=6, num_dice=2, target=7`

Output:



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS E:\studies\DAA\daa lab> & "C:/Program Files/Python
6
6
PS E:\studies\DAA\daa lab>
```

Result: Correct number of dice combinations calculated.

Q2. Longest Palindromic Substring

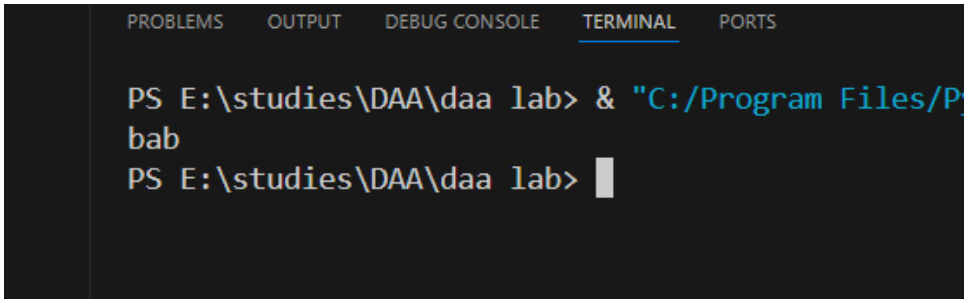
Aim:
To find the longest palindrome within a given string using DP.
Algorithm:

1. Initialize DP table for substrings.
2. Mark single letters as palindrome.
3. Expand substrings outward.
4. Update max length when palindrome found.
5. Return substring.

```python
# longest_palindrome.py
def longest_palindrome(s):
    if not s: return ""
    start = 0; end = 0
    for i in range(len(s)):
        # odd
        l, r = i, i
        while l >= 0 and r < len(s) and s[l] == s[r]:
            if (r - l) > (end - start):
                start, end = l, r
            l -= 1; r += 1
        # even
        l, r = i, i+1
        while l >= 0 and r < len(s) and s[l] == s[r]:
            if (r - l) > (end - start):
                start, end = l, r
            l -= 1; r += 1
    return s[start:end+1]

if __name__ == "__main__":
    print(longest_palindrome("babad"))  # "bab" or "aba"
    print(longest_palindrome("cbbd"))   # "bb"
```

Input: `"babad"`

Output: `"bab"`



Result: Longest palindrome identified correctly.

Q3. Word Break Problem

Aim:
To check if a string can be segmented into valid dictionary words.
Algorithm:

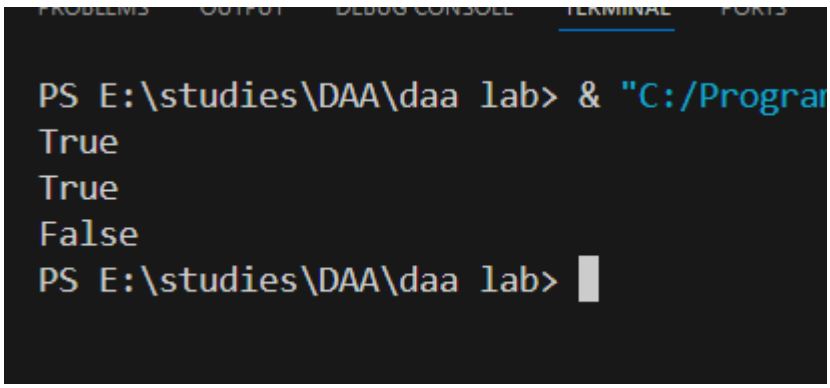
1. Read input string and dictionary.
2. Create DP array `dp[n+1]`.
3. Initialize `dp[0]=true`.
4. For each index, check substrings in dictionary.
5. Return `dp[n]`.

topic 4 q 3.py

```python
# word_break_dp.py
def word_break(s, wordDict):
    n = len(s)
    dp = [False] * (n+1)
    dp[0] = True
    wordset = set(wordDict)
    for i in range(1, n+1):
        for j in range(i):
            if dp[j] and s[j:i] in wordset:
                dp[i] = True
                break
    return dp[n]

if __name__ == "__main__":
    print(word_break("leetcode", ["leet","code"]))
    print(word_break("applepenapple", ["apple","pen"]))
    print(word_break("catsandog", ["cats","dog","sand","and","cat"]))
```

Input: `"leetcode", ["leet","code"]`

Output:



Result: String segmented successfully.

Q4. Floyd-Warshall Algorithm

Aim:
To find shortest paths between all pairs of vertices.
Algorithm:

1. Input weighted adjacency matrix.
2. For k = 1 to n:
   a. For each i, j update: `dist[i][j]=min(dist[i][j], dist[i][k]+dist[k][j])`.
3. Repeat for all vertices.
4. Print final matrix.
5. Display shortest paths.

Code:

topic 4 q 4.py > ...

```python
# floyd_warshall.py
INF = 10**9

def floyd_warshall(n, edges):
    # n vertices 0..n-1, edges: list of (u,v,w) undirected or directed? assume directed given edges
    dist = [[INF]*n for _ in range(n)]
    for i in range(n):
        dist[i][i] = 0
    for u,v,w in edges:
        dist[u][v] = w
        # if undirected, also: dist[v][u] = w
    # Floyd-W
    for k in range(n):
        for i in range(n):
            for j in range(n):
                if dist[i][j] > dist[i][k] + dist[k][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]
    return dist

if __name__ == "__main__":
    n = 4
    edges = [[0,1,3],[1,2,1],[1,3,4],[2,3,1]]
    dist = floyd_warshall(n, edges)
    print("Distance matrix:")
    for row in dist:
        print(row)
    # Example: distance from 0 to 3:
    print("dist[0][3] =", dist[0][3])

```

Input: Distance matrix of 4 cities.

Output: Updated shortest path matrix.

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS E:\studies\DAA\daa lab> & "C:/Program Files
Distance matrix:
[0, 3, 4, 5]
[1000000000, 0, 1, 2]
[1000000000, 1000000000, 0, 1]
[1000000000, 1000000000, 1000000000, 0]
dist[0][3] = 5
PS E:\studies\DAA\daa lab>
```

Result: Shortest paths correctly computed.

Q5. Optimal Binary Search Tree

Aim:
To construct a binary search tree minimizing expected search cost.
Algorithm:

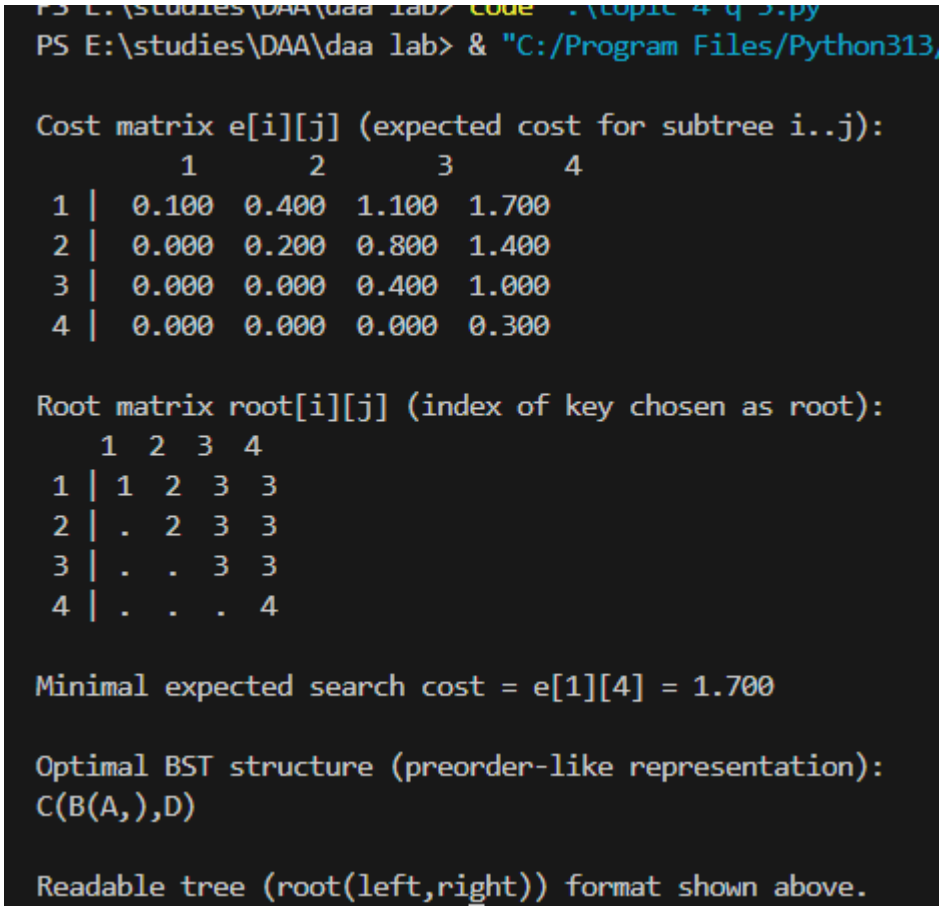
1. Input keys and frequencies.
2. Initialize cost matrix.
3. Compute cumulative frequency.
4. Apply DP to fill cost[i][j].
5. Display minimum cost and root matrix.

Code:

```python
# obst.py
from typing import List, Tuple

def optimal_bst(keys: List[str], p: List[float]) -> Tuple[List[List[float]], List[Li
    """
    keys: list of key names (length n)
    p: list of probabilities/frequencies for keys (length n)
    Returns: cost matrix (1-based indexing used inside) and root matrix
    """
    n = len(keys)
    # Use 1-based indexing internally for easier mapping to CLRS-like DP
    # cost/e and weight/w arrays of size (n+2) x (n+1) to handle e[i][i-1] base case
    e = [[0.0] * (n + 2) for _ in range(n + 2)]
    w = [[0.0] * (n + 2) for _ in range(n + 2)]
    root = [[0] * (n + 2) for _ in range(n + 2)]

    # Initialize: e[i][i-1] = 0, w[i][i-1] = 0 (already zeros)
    # Convert p to 1-based: p1[1..n]
    p1 = [0.0] + p

    # DP: compute for lengths l = 1..n
    for l in range(1, n + 1):
        for i in range(1, n - l + 2):
            j = i + l - 1
            e[i][j] = float('inf')
            # weight = sum of probabilities p[i..j]
            w[i][j] = w[i][j - 1] + p1[j]
            # try every possible root r in [i..j]
            for r in range(i, j + 1):
                # cost when r is root: e[i][r-1] + e[r+1][j] + w[i][j]
                left_cost = e[i][r - 1]
                right_cost = e[r + 1][j]
                total = left_cost + right_cost + w[i][j]
```

Input: Keys = {A,B,C,D}, Freq = {0.1, 0.2, 0.3, 0.4}

Output: Cost = 1.7



Result: OBST constructed with minimal expected cost.