

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:

The screenshot shows a code editor window with a dark theme. The file being edited is 'dice_throw_dp.py'. The code defines a function 'dice_ways' that takes 'num_sides', 'num_dice', and 'target' as parameters. It initializes a DP table 'dp' where 'dp[d][s]' represents the number of ways to make sum 's' with 'd' dice. The base case is 'dp[0][0] = 1'. The recurrence relation is implemented in a nested loop: for each die from 1 to num_dice, and for each sum from 1 to target, it adds the number of ways to reach 'sum - face' with 'd-1' dice to 'dp[d][sum]'. Finally, it prints the result for specific test cases: (6, 2, 7) and (4, 3, 10). The status bar at the bottom indicates the file has 16 lines, 16 columns, 4 spaces, and is in UTF-8 encoding, using Python 3.13.7.

```
topic 4 q lpy
topic 4 q lpy > ...
1 # dice_throw_dp.py
2 def dice_ways(num_sides, num_dice, target):
3     # dp[d][s] = number of ways to make sum s with d dice
4     dp = [ [0] * (target+1) for _ in range(num_dice+1) ]
5     dp[0][0] = 1
6     for d in range(1, num_dice+1):
7         for s in range(1, target+1):
8             for face in range(1, num_sides+1):
9                 if s - face >= 0:
10                     dp[d][s] += dp[d-1][s-face]
11     return dp[num_dice][target]
12
13 if __name__ == "__main__":
14     print(dice_ways(6, 2, 7)) # Expected 6
15     print(dice_ways(4, 3, 10)) # Example from file; expected 27
16
```

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

Output:

The screenshot shows a terminal window with tabs for PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL (which is selected), and PORTS. The terminal output shows the command 'PS E:\studies\DAAlab> & "C:/Program Files/Python' followed by two '6' characters, indicating the program is running correctly. The status bar at the bottom shows the terminal has 16 lines, 16 columns, 4 spaces, and is in UTF-8 encoding, using Python 3.13.7.

```
PS E:\studies\DAAlab> & "C:/Program Files/Python
6
6
PS E:\studies\DAAlab>
```

Result: Correct number of dice combinations calculated.

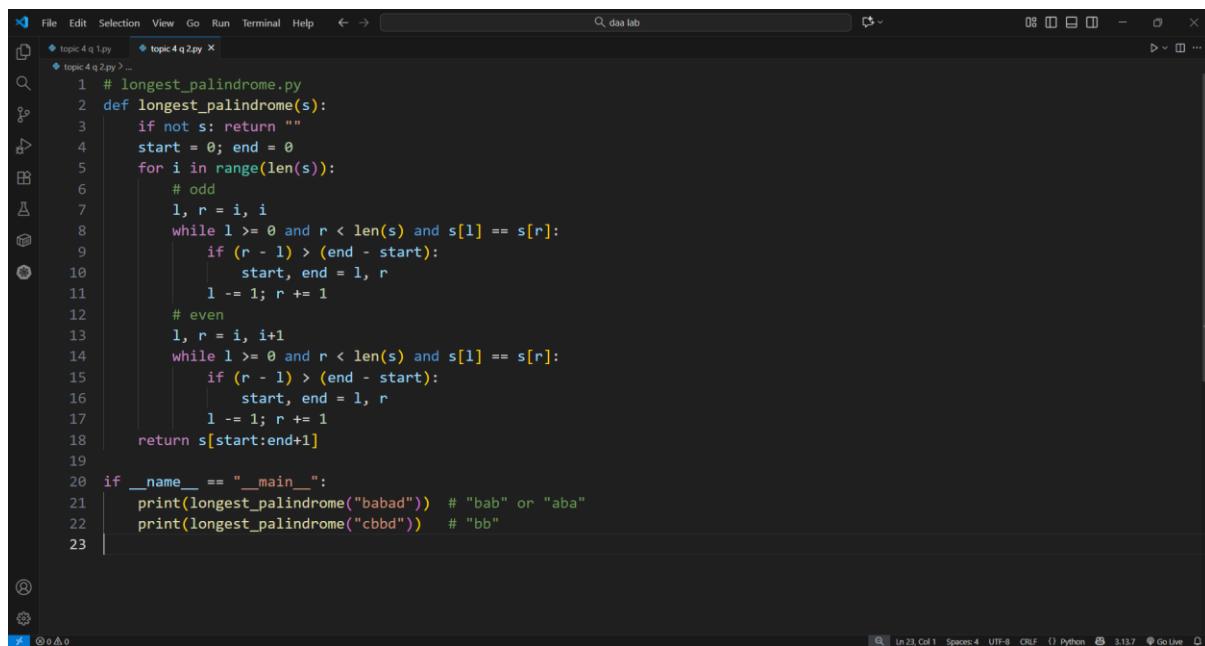
Q2. Longest Palindromic Substring

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.

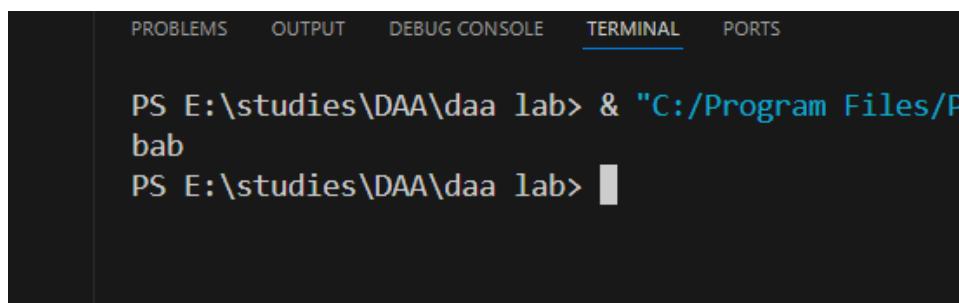


The screenshot shows a code editor window with the file `longest_palindrome.py` open. The code implements a function `longest_palindrome` that finds the longest palindromic substring in a given string `s`. It uses two nested loops to check for odd and even length palindromes centered at each character. The code includes imports for `os` and `sys`, and prints examples for the strings "babad" and "cbbd".

```
File Edit Selection View Go Run Terminal Help ← → 🔍 daa lab
topic4q1.py topic4q2py
topic4q2py
1 # longest_palindrome.py
2 def longest_palindrome(s):
3     if not s: return ""
4     start = 0; end = 0
5     for i in range(len(s)):
6         # odd
7         l, r = i, i
8         while l >= 0 and r < len(s) and s[l] == s[r]:
9             if (r - 1) > (end - start):
10                 start, end = l, r
11                 l -= 1; r += 1
12         # even
13         l, r = i, i+1
14         while l >= 0 and r < len(s) and s[l] == s[r]:
15             if (r - 1) > (end - start):
16                 start, end = l, r
17                 l -= 1; r += 1
18     return s[start:end+1]
19
20 if __name__ == "__main__":
21     print(longest_palindrome("babad")) # "bab" or "aba"
22     print(longest_palindrome("cbbd")) # "bb"
23
```

Input: "babad"

Output: "bab"



The screenshot shows a terminal window with the following output:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS E:\studies\DAA\daa_lab> & "C:/Program Files/Py
bab
PS E:\studies\DAA\daa_lab>
```

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 $dp[0]=\text{true}$.
4. For each index, check substrings in dictionary.
5. Return $dp[n]$.

```
❶ topic4 q 3.py ✘
❷ topic4 q 3.py > ...
1 # word_break_dp.py
2 def word_break(s, wordDict):
3     n = len(s)
4     dp = [False] * (n+1)
5     dp[0] = True
6     wordset = set(wordDict)
7     for i in range(1, n+1):
8         for j in range(i):
9             if dp[j] and s[j:i] in wordset:
10                 dp[i] = True
11                 break
12     return dp[n]
13
14 if __name__ == "__main__":
15     print(word_break("leetcode", ["leet", "code"]))
16     print(word_break("applepenapple", ["apple", "pen"]))
17     print(word_break("catsandog", ["cats", "dog", "sand", "and", "cat"]))
```

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

Output:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS E:\studies\DAA\daa_lab> & "C:/Program Files/Python39/python.exe" word_break_dp.py
True
True
False
PS E:\studies\DAA\daa_lab>
```

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 > ...
1 # floyd_marshall.py
2 INF = 10**9
3
4 def floyd_marshall(n, edges):
5     # n vertices 0..n-1, edges: list of (u,v,w) undirected or directed? assume directed given edges
6     dist = [[INF]*n for _ in range(n)]
7     for i in range(n):
8         dist[i][i] = 0
9     for u,v,w in edges:
10        dist[u][v] = w
11        # if undirected, also: dist[v][u] = w
12    # Floyd-W
13    for k in range(n):
14        for i in range(n):
15            for j in range(n):
16                if dist[i][j] > dist[i][k] + dist[k][j]:
17                    dist[i][j] = dist[i][k] + dist[k][j]
18    return dist
19
20 if __name__ == "__main__":
21     n = 4
22     edges = [[0,1,3],[1,2,1],[1,3,4],[2,3,1]]
23     dist = floyd_marshall(n, edges)
24     print("Distance matrix:")
25     for row in dist:
26         print(row)
27     # Example: distance from 0 to 3:
28     print("dist[0][3] =", dist[0][3])
29
```

Input: Distance matrix of 4 cities.

Output: Updated shortest path matrix.

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS E:\studies\DAAl\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\DAAl\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:

```
1 # obst.py
2 from typing import List, Tuple
3
4 def optimal_bst(keys: List[str], p: List[float]) -> Tuple[List[List[float]], List[List[int]]]:
5     """
6         keys: list of key names (length n)
7         p: list of probabilities/frequencies for keys (length n)
8         Returns: cost matrix (1-based indexing used inside) and root matrix
9     """
10    n = len(keys)
11    # Use 1-based indexing internally for easier mapping to CLRS-like DP
12    # cost/e and weight/w arrays of size (n+2) x (n+1) to handle e[i][i-1] base case
13    e = [[0.0] * (n + 2) for _ in range(n + 2)]
14    w = [[0.0] * (n + 2) for _ in range(n + 2)]
15    root = [[0] * (n + 2) for _ in range(n + 2)]
16
17    # Initialize: e[i][i-1] = 0, w[i][i-1] = 0 (already zeros)
18    # Convert p to 1-based: p1[1..n]
19    p1 = [0.0] + p
20
21    # DP: compute for lengths l = 1..n
22    for l in range(1, n + 1):
23        for i in range(1, n - l + 2):
24            j = i + l - 1
25            e[i][j] = float('inf')
26            # weight = sum of probabilities p[i..j]
27            w[i][j] = w[i][j - 1] + p1[j]
28            # try every possible root r in [i..j]
29            for r in range(i, j + 1):
30                # cost when r is root: e[i][r-1] + e[r+1][j] + w[i][j]
31                left_cost = e[i][r - 1]
32                right_cost = e[r + 1][j]
33                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

```
PS E:\studies\DAAl\daalab> code .\topic 4 q 3.py
PS E:\studies\DAAl\daalab> & "C:/Program Files/Python313"

Cost matrix e[i][j] (expected cost for subtree i..j):
    1      2      3      4
1 | 0.100  0.400  1.100  1.700
2 | 0.000  0.200  0.800  1.400
3 | 0.000  0.000  0.400  1.000
4 | 0.000  0.000  0.000  0.300

Root matrix root[i][j] (index of key chosen as root):
    1  2  3  4
1 | 1  2  3  3
2 | .  2  3  3
3 | .  .  3  3
4 | .  .  .  4

Minimal expected search cost = e[1][4] = 1.700

Optimal BST structure (preorder-like representation):
C(B(A,),D)

Readable tree (root(left,right)) format shown above.
```

Result: OBST constructed with minimal expected cost.