CS-218-HW4-Basic

Lakhan Kumar Sunilkumar

16 May 2025

Student ID: 862481700

1 A. Symmetry Makes Perfect: Codeforces Submission ID: 319842757

Explanation:

So we are given with a candy string, where we know that each character represents a flavor. Each flavor among the first k lowercase English letters has a specified cost of insertion. The given string is free and it may not be a palindrome. The goal of this problem is to make the string symmetric using only insertions, at any position, with the goal of minimizing the total cost of insertions and note that, no deletions or replacements are allowed.

Plan of Attack:

To solve this problem efficiently, a **dynamic programming (DP)** strategy is used. For any substring $s[i..j]$, we can recursively build solutions for larger substrings from smaller ones.

- Define a DP table `dp[i][j]` as the minimum cost to make the substring $s[i..j]$ a palindrome.
- If the characters at both ends are equal ($s[i] == s[j]$), no insertion is needed at this level.
- Otherwise, let's consider two cases:
  - Insert $s[i]$ after position j: `dp[i+1][j] + cost[s[i]]`
  - Insert $s[j]$ before position i: `dp[i][j-1] + cost[s[j]]`
- Take the minimum of these two values to update `dp[i][j]`.
- We build the table from substrings of length 2 up to n, filling it bottom-up.

Algorithm Steps:

1. Read the total number of flavors k, and build a dictionary `w` of costs indexed by flavor.
2. Read the initial candy string s and compute its length n.
3. Initialize a 2D DP table of size $n \times n$ with zeros.
4. Iterate over substring lengths from 2 to n:
   - For each valid pair i, j such that $j = i + \text{length} - 1$, apply the recurrence:
     - If $s[i] == s[j]$, then `dp[i][j] = dp[i+1][j-1]`
     - Else, choose the cheaper insertion and set `dp[i][j]` accordingly.
5. The final result is stored in `dp[0][n-1]`.

Code Structure

```
main():
    Read two inputs: length of string (unused) and number of flavors k
    and convert it to int

    Initialize an empty dictionary w to store insertion costs for each flavor

    For i = 0 to k - 1:
        Read cost of the (i-th) flavor
        Map flavor character ('a' + i) to its cost in w

    Read the candy string s
    Let n be the length of s

    Initialize dp[n][n] with all zeros
    (dp[i][j] = min cost to make s[i..j] a palindrome)

    For substring length l from 2 to n:
        For start index i from 0 to n - l:
            Set j = i + l - 1  (end of the substring)

            If s[i] == s[j]:
                No insertion needed; use dp[i+1][j-1]
            Else:
                Option 1: Insert s[i] at the end → dp[i+1][j] + w[s[i]]
                Option 2: Insert s[j] at the start → dp[i][j-1] + w[s[j]]
                Take minimum of the two and store in dp[i][j]

    Print dp[0][n-1] as the minimum cost to make the entire string a palindrome
```

Time Complexity:

- There are $O(n^2)$ substrings to process.
- Each DP state takes constant time to compute.

Overall Time Complexity: $\mathbf{O(n^2)}$
Space Complexity: $\mathbf{O(n^2)}$ for the DP table.

References

- https://www.youtube.com/watch?v=EN7yGmZ9v2M/
- https://www.geeksforgeeks.org/minimum-insertions-to-form-a-palindrome-dp-28/
- https://stackoverflow.com/questions/57178027/find-minimum-cost-to-convert-to-palindrome

2 B. Cake Cutting: Codeforces Submission ID: 319941469

Explanation:

So for a given a circular cake divided into n slices, each of different sizes. Two players, Preston and Celvin, take alternate turns to pick slices only by following the rules mentioned below:

- Preston picks first.
- On each turn, a player can only pick a slice that is adjacent to one that has already been taken.
- Celvin always plays greedily that is he picks the largest valid slice available to him at that moment.
- The goal is to maximize the total amount of cake Preston can get by choosing his starting slice optimally.

Since the cake is circular, the adjacency rule wraps around from the last slice to the first.

Plan of Attack:

To approach this optimally, a **dynamic programming (DP)** solution is implemented with interval logic. The circular nature of the cake is handled by duplicating the array to simulate all contiguous segments of size n. A DP table `dp[i][j]` stores the maximum amount of cake Preston can collect from interval $[i, j]$, assuming both players play optimally (Celvin being greedy).

For each sub-interval:

- Preston has two options: take the left end or the right end.
- Celvin then reacts greedily by picking the better of the two adjacent options.
- Based on Celvin's response, we update the interval and proceed recursively.

Algorithm Steps:

1. Read the number of slices n and the slice values.
2. Duplicate the array to size $2n$ to handle the circular wraparound.
3. Initialize a 2D DP table of dimensions $2n \times 2n$.
4. Fill base case: `dp[i][i]` $= A[i]$ for all i.
5. Iterate over all possible lengths from 2 to n, and for each interval:
   - If Preston takes the left slice:
     - Celvin chooses between $A[i+1]$ and $A[j]$.
     - Update `dp[i][j]` accordingly.
   - If Preston takes the right slice:
     - Celvin chooses between $A[i]$ and $A[j-1]$.
     - Update `dp[i][j]` accordingly.
   - Use the maximum of the two results.
6. The final answer is the maximum of `dp[i][i+n-1]` over all valid i in the duplicated array.

Code Structure

```
main():
    Read the number of cake slices from input

    Read the size of each slice and store it in a list

    Duplicate the list of slice sizes to handle the circular structure of the cake
    This allows us to treat any contiguous n-length interval as a valid slice sequence

    Create a 2D table (dp) to store the best possible amount of cake Preston can collect
    Each cell dp[i][j] represents the optimal result for the interval from i to j

    Fill in base cases: if the interval contains only one slice, Preston simply takes it

    For every possible interval length from 2 up to n:
        For every valid starting position of the interval in the duplicated array:
            Identify the end of the interval based on the current length

            Assume Preston chooses the leftmost slice:
                Simulate Celvin's greedy response:
                    If the slice on the immediate right is larger than the one at the end,
                    Celvin picks it and we shrink the interval from the left again
                    Otherwise, Celvin picks the end slice and we shrink from both ends

                Record the total cake Preston gets from this route

            Assume Preston chooses the rightmost slice:
                Simulate Celvin's greedy response:
                    If the slice at the beginning is larger than the one just before the end,
                    Celvin picks it and we shrink from both ends
                    Otherwise, Celvin picks the one before the end and we shrink the interval from the

                Record the total cake Preston gets from this route

            Take the maximum of both strategies and store it in the dp table

    After building the dp table:
        Slide a window of length n across the duplicated array
        For each starting index, extract the best result from dp[i][i+n-1]
        Keep track of the overall best result from all windows

    Output the maximum cake Preston can collect when playing optimally
```

Time Complexity:

- There are $O(n^2)$ interval pairs.
- Each dp state is computed in $O(1)$ time.

Overall Time Complexity: $\mathbf{O(n^2)}$
Space Complexity: $\mathbf{O(n^2)}$ for the DP table.

References

- https://www.atlantis-press.com/article/125968563.pdf

3 C. The Met Codeforces Submission ID: 319843693

Explanation:

So we are given with a tree structure where each node represents an exhibit and there is only one unique path between any two exhibits (i.e., a tree). The objective is to determine the optimal location for placing a brochure booth such that the maximum distance from this booth to any exhibit is minimized.

This is equivalent to finding the **radius** of the tree. To calculate this:

- First, compute the **diameter** of the tree which is the longest shortest path between any two nodes.
- Then, take the ceil of half of this diameter. This value is the minimum possible maximum distance from the optimal node to any other node.

Plan of Attack:

- Use Breadth-First Search (BFS) twice to find the diameter of the tree:
  - BFS from an arbitrary node to find the farthest node u.
  - BFS from u to find the farthest node v and the distance d between them.
- Compute the radius as $\lceil d/2 \rceil$.

Algorithm Steps:

1. Read integer input n representing the number of nodes in the tree.
2. Initialize an adjacency list `adj` of size $n + 1$ (1-based indexing).
3. For each of the $n - 1$ edges:
   - Read the two integers u and v representing a bidirectional edge between nodes u and v.
   - Append each node to the other's adjacency list.
4. Define a `bfs(start)` function to:
   - Initialize an array `dist` to store distance from the start node, set all to -1 initially.
   - Set distance of `start` to 0 and add it to the queue.
   - While queue is not empty:
     - Dequeue a node u.
     - For each neighbor w of u:
       - If w has not been visited, update its distance to `dist[u] + 1`.
       - Enqueue w.
       - Track the farthest node and its distance during the traversal.
   - Return the farthest node and its distance.
5. Call `bfs(1)` to find one endpoint u of the tree diameter.
6. Call `bfs(u)` to find the diameter length d.
7. Output $(d + 1)//2$ as the radius of the tree.

Code Structure

```
main():
    Read number of exhibits (n)
    Initialize an adjacency list of n+1 empty lists

    For each of the n - 1 connections:
        Read two nodes u and v
        Add v to the adjacency list of u
        Add u to the adjacency list of v

    Define function bfs(start):
        Initialize distance list of size n+1 with -1
        Set distance of start node to 0
        Create a queue and enqueue start node
        Track the farthest node and max distance found so far

        While queue is not empty:
            Dequeue a node u
            For each neighbor w of u:
                If w is unvisited:
                    Set distance of w to distance of u + 1
                    Enqueue w
                    Update farthest node if this is the longest so far

        Return farthest node and distance

    Run bfs from node 1 to find one endpoint u of the diameter
    Run bfs from u to find the diameter length d
    Print ceil(d / 2) as (d + 1) // 2
```

Time Complexity:

- BFS runs in $O(n)$ for trees.
- Two BFS runs are performed.

Overall Time Complexity: O(n)
Space Complexity: O(n) for the adjacency list and distance array.

References

- `https://cs.stackexchange.com/questions/22855/algorithm-to-find-diameter-of-a-tree-using-bfs-dfs-wh`
- `https://www.geeksforgeeks.org/graph-measurements-length-distance-diameter-eccentricity-radius-cent`
- `https://www.youtube.com/watch?v=04x5aGu6Oew`