Week 1 Interviewing Challenge Solution

Problem Statement

Write a function - **int getMinimumPartitions (string S)** that returns the minimum number of partitions needed to be made on S such that all partitioned substrings of S are palindromic.

Solution

It makes sense to divide this problem into two broad steps -

Step 1 -> Find all of the palindromic substrings within string S

Step 2 -> Use these results to find the minimum partitions required

Step 1 -> Find all palindromic substrings

We're going to declare a 2D matrix **-> bool isPalindrome[S.length()][S.length()]**, where isPalindrome[i][j] is true if the substring starting at index i and ending at index j is palindrome. You could also argue that we could further reduce the space we use, since some elements (**(S.length() ^ 2) / 2 - S.length() / 2**) to be exact) will be unused, but for simplicity, let's just go ahead with these S.length() ^ 2 elements.

Now, we can actually find all these palindromic substrings in different ways -

1. The most basic approach would be implementing a **brute force algorithm**.
   In this implementation, we would be extracting every possible substring and evaluating whether it is palindrome or not.
   However, such an approach is definitely not optimal time-wise. If implemented correctly, it would have an **O(n^3)** time complexity and wouldn't need any additional space.

2. Another option is using **dynamic programming**.
   Let's take an index position k < S.length().
   Let's assume that we have computed all palindromic substrings whose ending indices are **less than k**.
   So how do we use these results to find all palindromic substrings ending at k?
   Well, first as our base case we can say that isPalindrome[k][k] will always be true. We can also say that isPalindrome[k-1][k] will be true if k > 0 and s[k-1] is equal to s[k].
   After this, for any index i such that i >= 0 and i < k - 1, isPalindrome[i][k] will be true if s[i] is equal to s[k] and isPalindrome[i+1][k-1] is true. And to make it clear, we have already evaluated the value of isPalindrome[i+1][k-1] in the outer loop's previous iteration.
   So we can just run an inner loop where i counters from 0 to k-1.
   Thus, the overall time complexity of this implementation is **O(n^2)**.

3. The last approach is called **expanding around the center**. Here, for every character in the string, we assume it is the center of a palindromic substring and try to find all possible palindromic substrings where it is at the center.
   We can do this by maintaining a **low** pointer (equal to current index - 1) and a **high** pointer (equal to current index + 1).
   Basically, if s[low] is equal to s[high], isPalindrome[low][high] will be true.
   We continue to decrement low, increment high, and mark palindromic substrings as long as low and high are within the bounds of the string and while s[low] is equal to s[high].
   However, this only gives us all palindromic substrings of odd length. To also take even lengths into consideration, we assume the current character and

the next character as the centers of the strings and more or less do the same process.

This approach also has an **O(n^2)** time complexity.

Step 2 -> Using these results to find the minimum partitions required

We can proceed with this step by visualizing it as a **dynamic programming** approach.

Let's declare an array **int numPartitions[S.length()]**, where numPartitions[i] represents the minimum number of partitions needed to form palindromic substrings assuming i is the string's endpoint.

First, we can maintain the base case that numPartitions[0] will always be 0.

Next, let's say we have computed all values in numPartitions upto index i-1. So, now we have to try to use these results to find numPartitions[i].

First of all, if isPalindrome[0][i] is true, then numPartitions[i] will be 0.

Or else, we run a loop where j counters from 1 to i. We also create a variable min which is initialized to i.

For every iteration of this loop, if isPalindrome[j][i] is true and numPartitions[j-1] + 1 is less than min, then min is changed to numPartitions[j-1] + 1.

Ultimately, we store the value of min in numPartitions[i].

Applying this method, our desired result is the last element in numPartitions.

This implementation has an **O(n^2)** time complexity.

Thus, the overall complexity of the solution is **O(n^2)** time and **O(n^2)** space.

You can find the C++ implementation for this solution here. This solution applies the dynamic programming approach for both steps.

Feel free to message **@Abhik Mazumder** on Slack for any questions.