132. Palindrome Partitioning II

Hard String Dynamic Programming Leetcode Link

Problem Description

The problem is centered around a given string `s` and the goal is to partition the string in such a way that each substring of the partition is a palindrome. A palindrome is defined as a string that reads the same forward and backward. The main objective here is to find the minimum number of cuts necessary to achieve this partitioning. A cut defines a division between two characters in the string, creating separate substrings that are palindromes.

For example, if `s` is "aab", a partition like "aa"|"b" is valid, as both "aa" and "b" are palindromes. The minimum number of cuts in this example is 1.

Intuition

The intuition behind the solution is to first precalculate which substrings are palindromes and then use dynamic programming to find the minimum number of cuts needed.

To determine if substrings are palindromes or not, we use a two-dimensional array `g`, where `g[i][j]` will be `True` if the substring from `s[i]` to `s[j]` (inclusive) is a palindrome. This is calculated in a bottom-up manner starting from the end of the string, using the fact that a substring `s[i:j]` is a palindrome if its outer letters are equal (`s[i] == s[j]`) and the substring `s[i + 1:j - 1]` is itself a palindrome.

Then we use a one-dimensional array `f`, where `f[i]` represents the minimum number of cuts needed to partition the string `s` up to index `i`. The relationship between `g` and `f` is crucial; we iterate over all possible end indices `i` of a palindrome and, for each, iterate over all possible start indices `j`. If `g[j][i]` is `True` (so `s[j:i+1]` is a palindrome), we know that one possible way to partition the string up to `i` is to cut right before `j` and then right after `i`. Therefore, `f[i]` can be updated to be the minimum between its current value and `f[j-1] + 1` (because we take one cut before `j`). If `j` is zero, we do not need a cut, as it means the entire `s[0:i+1]` is a palindrome.

The base case for `f` is that `f[i]` is at most `i`, because the worst case is cutting the string at every character making each letter a palindrome substring. At the end of the process, `f[-1]` (last element of `f`) gives us the minimum number of cuts needed for the entire string `s`.

Solution Approach

The solution uses dynamic programming and the concept of palindrome checking to solve the problem in an efficient manner. It involves two phases: precalculation of palindrome substrings and finding the minimum number of cuts using the precalculated data.

Precomputation of Palindrome Substrings

A two-dimensional table `g` is created to store whether a substring `s[i...j]` is a palindrome or not. It uses a bottom-up approach to fill this table. The base cases are that all single characters are palindromes (`g[i][i] = True` for all `i`), and for two character substrings, they are palindromes if both characters are the same (`g[i][i+1] = s[i] == s[i+1]`).

For substrings longer than two characters, `g[i][j]` is `True` if the first and last characters are the same (`s[i] == s[j]`) and the substring between them, `s[i+1...j-1]`, is also a palindrome (`g[i+1][j-1] == True`). This is done by the following nested loop:

```
for i in range(n - 1, -1, -1):
    for j in range(i + 1, n):
        g[i][j] = s[i] == s[j] and g[i + 1][j - 1]
```

Dynamic Programming for Minimum Cuts

The dynamic programming array `f` is initialized to represent the worst-case number of cuts for each index, which is when all characters up to that index are cut separately (`f[i] = i`). Then the algorithm iterates over each end index `i` and checks every start index `j` to see if a palindrome is formed (`g[j][i] == True`).

To update the minimum cuts for `f[i]`, the code takes the minimum of the current value `f[i]` or `f[j-1] + 1`, the latter representing a cut at `j` producing one more cut than the minimum cuts needed for `s[0...j-1]`. The `if j else 0` part handles the case where the entire substring `s[0...i]` is a palindrome, so no cut is needed, setting `f[i]` to 0 in that case.

```
f = list(range(n))
for i in range(n):
    for j in range(i + 1):
        if g[j][i]:
            f[i] = min(f[i], i + (f[j - 1] if j else 0))
```

The final answer representing the minimum cuts needed for the whole string `s` is stored in `f[-1]`.

By using a combination of a palindrome precomputation with dynamic programming, the proposed solution efficiently minimizes the cuts required to partition the string into palindrome substrings.

Example Walkthrough

Let's walk through an example of the solution approach with the input string `s = "aab"` as per the problem description.

Precomputation of Palindrome Substrings

First, we initialize the two-dimensional table `g`, which will store `True` or `False` values indicating whether the substring `s[i...j]` is a palindrome.

Given the string `s = "aab"`, our table `g` starts off with the following base cases filled:

- Single characters are always palindromes: `g[0][0] = True`, `g[1][1] = True`, `g[2][2] = True`.
- For two adjacent characters: `g[0][1] = True` because `s[0] == s[1]` ("aa"), `g[1][2] = False` because `s[1] != s[2]` ("ab").

Now, we need to check substrings longer than two characters:

- Since our string length `n = 3`, there is no substring longer than two characters and we skip this step.

So, our precomputation step completes with the table `g` looking like this:

```
1 g = [ [True, True, False],
2       [False, True, False],
3       [False, False, True] ]
```

Dynamic Programming for Minimum Cuts

We now initialize the dynamic programming array `f` with worst-case number of cuts for each index: `f = [0, 1, 2]`.

Next, we iterate over each end index `i` and check every start index `j` to see if a palindrome is formed:

1. For `i = 1`:
   - `j = 0`: `g[0][1] == True`, so we check `f[1]` against `1 + f[j - 1] if j else 0`, which translates to `1 + 0 = 1`. Therefore, `f[1]` stays the same.
   - `j = 1`: We skip this since `g[j][i] == g[1][1] == True` and is a base case for single characters.
2. For `i = 2`:
   - `j = 0`: `g[0][2] == False`, we do not update `f[2]`.
   - `j = 1`: `g[1][2] == False`, also do not update `f[2]`.
   - `j = 2`: `g[2][2] == True`, we check `f[2]` against `1 + f[j - 1] if j else 0`, which is `1 + 1 = 2`. Therefore, we update `f[2]` with the minimum, which remains `2`.

At the end of the process, `f` looks like this: `f = [0, 1, 2]`.

The final answer is the minimum number of cuts needed to partition the string into palindrome substrings, and it is stored in `f[-1]`, which is `2` in this case. However, we have overlooked one small detail: in the second iteration for `i = 2`, when `j = 1`, since `g[1][2]` is `False`, we do not update `f[2]`. Instead, we should look one character before index `1`. Here is how it plays out:

- When `j = 0`, and since `g[0][1] == True`, we get `f[2]` could be updated to `1 + f[j - 1] if j else 0` = `1` instead of the previous `2`.

This correction gives us the final updated `f = [0, 1, 1]` and `f[-1] = 1`, which represents the minimum cuts needed for the entire string `s`.

Therefore, the minimum number of cuts required for "aab" to make every substring a palindrome is 1, with the cut after the first "aa" to isolate the "b".

Python Solution

```
1  class Solution:
2      def minCut(self, s: str) -> int:
3          # Length of the string `s`
4          length = len(s)
5
6          # Initialize a 2D list where palindrome[i][j] will be True if the
7          # substring s[i:j+1] is a palindrome
8          palindrome = [[True] * length for _ in range(length)]
9
10         # Fill the palindrome table
11         # We start from the end towards the beginning because each cell depends on the next cells
12         for start in range(length - 1, -1, -1):
13             for end in range(start + 1, length):
14                 # A substring is a palindrome if its outer characters are equal
15                 # and the substring excluding the outer characters is a palindrome
16                 palindrome[start][end] = s[start] == s[end] and palindrome[start + 1][end - 1]
17
18         # Initialize a list to store the minimum number of cuts needed for
19         # a palindrome partitioning of the substring s[:i+1]
20         cuts = list(range(length))
21
22         # Calculate the minimum cuts needed for each substring
23         for end in range(length):
24             for start in range(end + 1):
25                 # If the substring s[j:i+1] is a palindrome
26                 if palindrome[start][end]:
27                     # If j is 0, then s[i:i+1] is a palindrome and doesn't need a cut
28                     # Otherwise, update the minimum cuts for s[i:i+1]
29                     cuts[end] = min(cuts[end], 0 if start == 0 else 1 + cuts[start - 1])
30
31         # Return the minimum cuts needed for the whole string
32         return cuts[-1]
```

Java Solution

```
1  class Solution {
2      public int minCut(String s) {
3          int length = s.length();
4          // Create a 2D matrix to track if the substring from i to j is a palindrome
5          boolean[][] isPalindrome = new boolean[length][length];
6
7          // Initially fill the matrix with true values, because a single character is a palindrome
8          for (boolean[] row : isPalindrome) {
9              Arrays.fill(row, true);
10         }
11
12         // Populate the isPalindrome matrix by checking substrings
13         for (int start = length - 1; start >= 0; --start) {
14             for (int end = start + 1; end < length; ++end) {
15                 // A substring is a palindrome if its first and end characters are the same
16                 // and the substring between them is also a palindrome
17                 isPalindrome[start][end] = s.charAt(start) == s.charAt(end) && isPalindrome[start + 1][end - 1];
18             }
19         }
20
21         // Create an array to hold the minimum number of cuts needed for each prefix of the string
22         int[] minCuts = new int[length];
23
24         // Initialize with the maximum number of cuts possible (i.e., every character can be a cut)
25         for (int i = 0; i < length; ++i) {
26             minCuts[i] = i;
27         }
28
29         // Build the minCuts array by comparing the minimum cuts for each palindrome partition
30         for (int end = 1; end < length; ++end) {
31             for (int start = 0; start <= end; ++start) {
32                 // If the substring from start to end is a palindrome
33                 if (isPalindrome[start][end]) {
34                     // If the palindrome starts at 0, no need to cut; otherwise, add 1 to the previous minCut value
35                     minCuts[end] = Math.min(minCuts[end], start == 0 ? 0 : minCuts[start - 1] + 1);
36                 }
37             }
38         }
39
40         // Return the minCut for the entire string (last position in the minCuts array)
41         return minCuts[length - 1];
42     }
43 }
44
```

C++ Solution

```
1  class Solution {
2  public:
3      int minCut(string s) {
4          int length = s.size(); // The length of the input string.
5          vector<vector<bool>> isPalindrome(length, vector<bool>(length, true));
6
7          // Preprocessing: fill the isPalindrome table
8          // Initially assume all substrings will be 'true' if the string from index i to j
9          // is NOT a palindrome.
10         // Otherwise, 'true' - this means that isPalindrome[i][j] will always be 'true' (1-letter palindromes).
11         for (int start = length - 1; start >= 0; --start) {
12             for (int end = start + 1; end < length; ++end) {
13                 isPalindrome[start][end] = (s[start] == s[end]) && isPalindrome[start + 1][end - 1];
14             }
15         }
16
17         // Array to store the minimum cut count that can make the substring a palindrome.
18         vector<int> minCuts(length);
19         // Initialization: the array cuts[i] is the number of cuts needed for string[0...i]
20         for (int i = 0; i < length; ++i) {
21             minCuts[i] = i;
22         }
23
24         // Figure out the minimum cut needed for substring [0...i] using dynamic programming
25         for (int i = 1; i < length; ++i) {
26             for (int j = 0; j <= i; ++j) {
27                 // If the substring from j to i is a palindrome
28                 if (isPalindrome[j][i]) {
29                     // cuts[i] will be 0 if 'i==0...i' is a palindrome itself, otherwise
30                     // it will be 'cuts[j-1] + 1' because we make a cut before 'j' and the count of cuts for 'i=0...j-1'
31                     // if 'j' is not zero.
32                     minCuts[i] = (j == 0) ? 0 : min(minCuts[i], 1 + minCuts[j - 1]);
33                 }
34             }
35         }
36
37         // Return the cuts for the whole string
38         return minCuts[length - 1];
39     }
40 };
41
```

Typescript Solution

```
1  // Function to calculate the minimum number of cuts needed to partition the string such that
2  // each partition is a palindrome.
3  function minCut(s: string): number {
4      // Calculate the length of the string.
5      const length = s.length;
6
7      // Create a 2D array (table) to store palindrome information.
8      // g[i][j] will be true if the substring s[i...j] is a palindrome.
9      const isPalindrome: boolean[][] = Array.from({ length }
10         , () => Array(length).fill(true));
11
12     // Fill the palindrome table with correct values.
13     for (let start = length - 1; start >= 0; --start) {
14         for (let end = start + 1; end < length; ++end) {
15             palindromeTable[start][end] = s[start] === s[end] && palindromeTable[start + 1][end - 1];
16         }
17     }
18
19     // Initialize an array to store the minimum cuts needed for substring s[0...i].
20     const minCuts: number[] = Array(length)
21         .fill(0)
22         .map((_, index) => index);
23
24     // Populate the minCuts array with the minimum number of necessary cuts.
25     for (let end = 1; end < length; ++end) {
26         for (let start = 0; start <= end; ++start) {
27             // If the current substring is a palindrome.
28             if (palindromeTable[start][end]) {
29                 // Update the minimum cuts for the current position.
30                 // If current substring starts from the beginning, no cut required.
31                 // Else, add 1 to the cuts required for the substring ending at the previous position.
32                 minCuts[end] = Math.min(minCuts[end], start > 0 ? 1 + minCuts[start - 1] : 0);
33             }
34         }
35     }
36
37     // Return the minimum number of cuts needed for the entire string.
38     return minCuts[length - 1];
39 }
40
41
```

Time and Space Complexity

Time Complexity

The time complexity of the solution is determined by two nested loops to fill the `g` matrix and two nested loops to compute the `f` array.

- Filling the `g` matrix requires a nested loop where `i` ranges from `n - 1` to `0` and `j` ranges from `i + 1` to `n - 1`. Each element `g[i][j]` is computed once, resulting in $O(n^2)$ for this part as each cell of an `n x n` grid is computed through the loops.
- Computing the `f` array involves another nested loop where `i` ranges from `1` to `n - 1` and for each `i`, `j` ranges from `0` to `i`. Inside this loop, a constant time operation is performed to update `f[i]`. The number of operations can be represented by the sum of the first `n` natural numbers minus one (since `j` starts from `0`), which is `(n * (n - 1)) / 2`, also resulting in $O(n^2)$ time.

Thus, the overall time complexity is $O(n^2) + O(n^2)$ which simplifies to $O(n^2)$.

Space Complexity

The space complexity of the solution is mainly determined by the space required to store the `g` matrix and the `f` array.

- The `g` matrix is an `n x n` boolean matrix where `n` is the length of the input string `s`, taking up $O(n^2)$ space.
- The `f` array is a one-dimensional array of length `n`, taking up $O(n)$ space.

So, the overall space complexity of the function is $O(n^2) + O(n)$ which simplifies to $O(n^2)$ as the n^2 term dominates for large n.