2002. Maximum Product of the Length of Two Palindromic Subsequences

Medium | Bit Manipulation | String | Dynamic Programming | Backtracking | Bitmask | Leetcode Link

Problem Description

The problem is about finding two non-overlapping palindromic subsequences in a given string `s`. A subsequence is a sequence that can be derived from another sequence by deleting some or no elements without changing the order of the remaining elements. The goal is to maximize the product of the lengths of these two palindromic subsequences. A string is palindromic if it reads the same forward and backward. The main challenge is to ensure that the subsequences are disjoint, meaning they do not share characters at the same index positions.

Intuition

The intuition behind the solution involves the following steps:

1. Generate all possible subsequences of the string.
2. Check which of these subsequences are palindromic.
3. Attempt to pair each palindromic subsequence with another, ensuring they do not overlap (disjoint).
4. Calculate the product of the lengths of each pair and keep track of the maximum product found.

The solution uses bitwise operations to efficiently represent and iterate over all subsequences. The bitmask representing each subsequence is used to check if two subsequences are disjoint by using XOR and AND operations. It also counts the number of set bits (`1s`) in the bitmask using `.bit_count()` to determine the length of the subsequence without actually constructing the subsequence string, which saves time and memory. Finally, it uses the precomputed palindromic status of each bitmask to quickly check if a subsequence is palindromic, avoiding repeated calculations.

Solution Approach

The solution provided leverages bit manipulation and dynamic programming to tackle the problem. The approach can be broken down into the following steps:

1. **Precompute Palindromic Subsequences**:
   - We iterate over all possible subsequences represented by bitmasks where each bit corresponds to an index in the string `s`. A bit set to `1` represents the inclusion of that character in the subsequence, while `0` represents exclusion.
   - The variable `p` is an array where each index corresponds to a bitmask of a subsequence. `p` is used to store whether the represented subsequence is palindromic or not. Initially, all values in `p` are set to `True`.
   - The first for loop goes through all possible bitmasks, using `k`. Nested while loops are then used to iterate from the outermost characters toward the center, checking if the characters are equal. If a pair of characters is not equal, `p[k]` is set to `False` and the loop breaks, indicating that this subsequence is not palindromic.
2. **Find Maximized Product of Lengths**:
   - With all palindromic subsequences identified, we loop through them with the bitmask `i`. The bitmask `mx` is computed as the XOR of `i` with the bitmask representing all characters in the string (i.e., `(1 << n) - 1`). This essentially inverts `i`, marking all indices not included in `i`.
   - Then, we initialize `j` with `mx` and enter a nested loop. Inside this loop, we only consider bitmasks `j` that are palindromic (`p[j] == True`). For each such bitmask, we use `.bit_count()` to calculate the length of the palindromic subsequences corresponding to the bitmasks `i` and `j` (stored in variables `a` and `b`, respectively).
   - The product of `a` and `b` is calculated and checked against the current maximum product `ans`. If it is larger, it becomes the new maximum.
3. **Iterate Over All Combinations**:
   - The critical optimization here is to iterate through all smaller bitmasks of `mx` that are still palindromic. This is done by decrementing `j` at each step using `j = (j - 1) & mx`. By ANDing with `mx`, we ensure we get smaller bitmasks that represent subsequences disjoint from subsequence `i`.
4. **Return Result**:
   - After all combinations are checked, the maximum product calculated is stored in `ans`, which is returned as the result.

The algorithm makes use of bit manipulation to efficiently enumerate subsequences and dynamically checks for palindromic properties to reduce redundant calculations. By exploiting bit counts and clever looping, it is able to quickly find the maximum product of the lengths of two disjoint palindromic subsequences.

Example Walkthrough

Let's consider a small example with the string `s = "ababa"`. We want to find two non-overlapping palindromic subsequences that yield the maximum product of their lengths.

1. **Precompute Palindromic Subsequences**:
   - First, we generate all possible subsequences using bit masks. For the string `s = "ababa"` which has a length of `5`, we will have `2^5 - 1 = 31` possible non-empty subsequences.
   - For simplicity, let's consider a few bitmasks and their corresponding subsequences:
     - `00101` which represents the subsequence "aa" (palindromic)
     - `01010` which represents the subsequence "bbb" (not palindromic)
     - `10100` which represents the subsequence "aba" (palindromic)
   - The array `p` would reflect if these subsequences are palindromic (`True` or `False`). For our example:
     - `p[5] = True` since the bitmask `5 (00101)` is palindromic.
     - `p[10] = False` for `10 (01010)` because it's not palindromic.
     - `p[20] = True` for `20 (10100)` since it's palindromic.
2. **Find Maximized Product of Lengths**:
   - Now, we look for pairs of palindromic subsequences that do not overlap and calculate the product of their lengths.
   - If we take the bitmask `20 (10100)` which corresponds to the subsequence "aba", we would then find the inverted bitmask `mx = 31 XOR 20 = 11 (01111)` representing 'b', 'ab', 'bb' or 'abb'.
   - Within the bitmask `11 (01111)`, we look for palindromic subsequences.
   - Let's say we find the bitmask `3 (00011)` which represents the subsequence "ab" and is also palindromic.
   - We check the lengths using `.bit_count()`: the length of `20 (10100)` is `3` and the length of `3 (00011)` is `2`. The product is `3 * 2 = 6`.
3. **Iterate Over All Combinations**:
   - We keep decrementing `j` using `j = (j - 1) & mx` to find all smaller, non-overlapping palindromic subsequences.
   - For instance, if `j` was initially `11 (01111)`, the next `j` would be `7 (00111)`, representing the subsequence "aab", which is not palindromic.
4. **Return Result**:
   - After examining all possible palindrome subsequence combinations, we determine the ones that give us the maximum product. In this example, the maximum product is `6`, given by the subsequences "aba" (length `3`) and "ab" (length `2`).

This walkthrough provides a conceptual understanding of how the solution uses bit masks and dynamic programming to efficiently find the maximum product of lengths of two non-overlapping palindromic subsequences.

Python Solution

```python
class Solution:
    def maxProduct(self, s: str) -> int:
        # Length of the string
        n = len(s)

        # is_palindrome will denote if the binary representation of a number corresponds to a palindromic substring
        is_palindrome = [True] * (1 << n)

        # Precompute all palindromic substrings using bit representation
        for bitmask in range(1, 1 << n):
            left, right = 0, n - 1
            while left < right:
                # Find the next '1' bit from the left
                while left < right and (bitmask >> left & 1) == 0:
                    left += 1
                # Find the next '1' bit from the right
                while left < right and (bitmask >> right & 1) == 0:
                    right -= 1
                # If the corresponding characters do not match, this is not a palindrome
                if left < right and s[left] != s[right]:
                    is_palindrome[bitmask] = False
                    break
                left += 1
                right -= 1

        # Initialize the result for maximum product of the lengths
        max_product = 0

        # Iterate over all possible bitmasks
        for i in range(1, 1 << n):
            # Proceed only if the bitmask represents a palindrome
            if is_palindrome[i]:
                # Inverse bitmask: set bits become unset and vice versa
                inverse_mask = ((1 << n) - 1) ^ i

                # Iterate over all submasks of the inverse bitmask
                j = inverse_mask
                len_a = i.bit_count()  # Count of set bits, giving the length of palindrome A
                while j:
                    # If j represents a palindrome, calculate the product of the lengths
                    if is_palindrome[j]:
                        len_b = j.bit_count()  # Length of palindrome B
                        max_product = max(max_product, len_a * len_b)
                    # Move to the next submask
                    j = (j - 1) & inverse_mask

        return max_product

```

Java Solution

```java
import java.util.Arrays;

class Solution {
    public int maxProduct(String s) {
        // Get the length of the string
        int stringLength = s.length();
        // Initialize a boolean array for palindrome checks with size as all possible subsets
        boolean[] isPalindrome = new boolean[1 << stringLength];
        // Default all entries to true
        Arrays.fill(isPalindrome, true);

        // Check each subset to see if it forms a palindrome
        for (int subset = 1; subset < 1 << stringLength; ++subset) {
            for (int left = 0, right = stringLength - 1; left < stringLength; ++left, --right) {
                // Find the next index 'left' where subset has a bit set
                while (left < right && (subset >> left & 1) == 0) {
                    ++left;
                }
                // Find the next index 'right' where subset has a bit set
                while (left < right && (subset >> right & 1) == 0) {
                    --right;
                }
                // If the characters at 'left' and 'right' don't match, it's not a palindrome
                if (left < right && s.charAt(left) != s.charAt(right)) {
                    isPalindrome[subset] = false;
                    break;
                }
            }
        }

        int maximumProduct = 0; // Initialize the maximum product of palindrome lengths

        // Calculate the product of lengths for all pairs of palindromic subsets
        for (int i = 1; i < 1 << stringLength; ++i) {
            if (isPalindrome[i]) { // If the subset at index i forms a palindrome
                int countA = Integer.bitCount(i); // Count the number of set bits

                // Calculate the complement of subset i
                int complement = ((1 << stringLength) - 1) ^ i;

                // Iterate through all subsets of complement
                for (int j = complement; j > 0; j = (j - 1) & complement) {
                    if (isPalindrome[j]) { // If the subset at index j forms a palindrome
                        int countB = Integer.bitCount(j); // Count the number of set bits
                        // Update the maximum product if the current pair product is larger
                        maximumProduct = Math.max(maximumProduct, countA * countB);
                    }
                }
            }
        }

        return maximumProduct; // Return the maximum product found
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to compute the maximum product of the lengths of two non-overlapping palindromic subsequences
    int maxProduct(string s) {
        int n = s.size(); // Get the size of the input string
        vector<bool> isPalindrome(1 << n, true); // Initialize a vector to track if a subsequence represented by bitmask is a palin

        // Check each subsequence represented by a bitmask to see if it is a palindrome
        for (int mask = 1; mask < (1 << n); ++mask) {
            for (int left = 0, right = n - 1; left < right; ++left, --right) {
                // Advance the left index until it points to a character included in the subsequence
                while (left < right && !(mask >> left & 1)) {
                    ++left;
                }
                // Move the right index back until it points to a character included in the subsequence
                while (left < right && !(mask >> right & 1)) {
                    --right;
                }
                // If the characters at the current left and right indices do not match, this is not a palindrome
                if (left < right && s[left] != s[right]) {
                    isPalindrome[mask] = false;
                    break;
                }
            }
        }

        int maxProduct = 0; // Initialize the maximum product to 0

        // Iterate over all bitmasks to find the maximum product of palindromic subsequence pairs
        for (int i = 1; i < (1 << n); ++i) {
            if (isPalindrome[i]) { // Only consider the bitmask if it represents a palindrome
                int lengthA = __builtin_popcount(i); // Compute the length of palindrome A
                int complementMask = ((1 << n) - 1) ^ i; // Generate a bitmask for the complementary subsequence

                // Find the other palindromic subsequence with the maximum length that can pair with the current one
                for (int j = complementMask; j; j = (j - 1) & complementMask) {
                    if (isPalindrome[j]) {
                        int lengthB = __builtin_popcount(j); // Compute the length of palindrome B
                        maxProduct = max(maxProduct, lengthA * lengthB); // Update the maximum product
                    }
                }
            }
        }

        return maxProduct; // Return the final maximum product of palindromic subsequence lengths
    }
};

```

Typescript Solution

```typescript
// Function to compute the maximum product of the lengths of two non-overlapping palindromic subsequences
function maxProduct(s: string): number {
    const n: number = s.length; // Get the length of the input string
    const isPalindrome: boolean[] = new Array(1 << n).fill(true); // Initialize an array to track if a subsequence is a palindrome

    // Check each subsequence represented by a bitmask to see if it is a palindrome
    for (let mask = 1; mask < (1 << n); ++mask) {
        let left = 0, right = n - 1;
        while (left < right) {
            // Advance the left index until it points to a character included in the subsequence
            while (left < right && !((mask >> left) & 1)) {
                ++left;
            }
            // Move the right index back until it points to a character included in the subsequence
            while (left < right && !((mask >> right) & 1)) {
                --right;
            }
            // If the characters at the current left and right indices do not match, this is not a palindrome
            if (left < right && s[left] !== s[right]) {
                isPalindrome[mask] = false;
                break;
            }
            left++;
            right--;
        }
    }

    let maxProduct = 0; // Initialize the maximum product to 0

    // Iterate over all bitmasks to find the maximum product of palindromic subsequence pairs
    for (let i = 1; i < (1 << n); ++i) {
        if (isPalindrome[i]) { // Only consider the bitmask if it represents a palindrome
            const lengthA = bitCount(i); // Compute the length of palindrome A
            const complementMask = ((1 << n) - 1) ^ i; // Generate a bitmask for the complementary subsequence

            // Find the other palindromic subsequence with the maximum length that can pair with the current one
            for (let j = complementMask; j; j = (j - 1) & complementMask) {
                if (isPalindrome[j]) {
                    const lengthB = bitCount(j); // Compute the length of palindrome B
                    maxProduct = Math.max(maxProduct, lengthA * lengthB); // Update the maximum product
                }
            }
        }
    }

    return maxProduct; // Return the final maximum product
}

// Function to count the number of set bits in a bitmask (equivalent to __builtin_popcount in C++)
function bitCount(mask: number): number {
    let count = 0;
    while (mask) {
        count += mask & 1;
        mask >>= 1;
    }
    return count;
}

```

Time and Space Complexity

The time complexity of the code above can be analyzed as follows:

1. The first for loop, running from `k` in `range(1, 1 << n)`, enumerates all possible subsets of the string `s` where `n` is the length of `s`. For each subset, the while loop checks if it forms a palindrome, which takes `O(n)` time in the worst case. The number of subsets is `2^n`, so this part of the code runs in `O(n * 2^n)` time.

2. The second part of the code contains two nested loops. The outer loop runs for `2^n - 1` iterations, and for each iteration, the inner while loop potentially runs multiple times. The maximal number of times the inner loop can run can be approximated by `2^n` again because it starts at `mx` and decreases until it reaches `0`. However, the average number of iterations is less due to the bitwise AND operation with `mx`. Since the exact number of iterations is hard to determine without a deeper analysis of the distribution of palindromes, we can approximate the time complexity of the inner loop with the upper bound of `O(2^n)` as well. The calculation within the loop includes bit count (`O(log n)`) and a max operation (`O(1)`), which are considerably less than `O(2^n)`, so they don't affect the overall time complexity. Thus, the second part of the code runs in `O(2^n * 2^n)` time.

The space complexity is determined by:

1. The boolean array `p` of size `2^n`, which results in a space complexity of `O(2^n)`.

2. The variables and constant space usage inside the loops do not contribute to the space complexity significantly as compared to `p`.

Therefore, the total time complexity of the algorithm can be estimated as `O(n * 2^n + 2^n * 2^n)` which simplifies to `O(2^n * 2^n)` because `2^n * 2^n` dominates `n * 2^n`. The space complexity is `O(2^n)`.