2896. Apply Operations to Make Two Strings Equal

Medium | String | Dynamic Programming | Leetcode Link

Problem Description

In this problem, we are given two binary strings of equal length, `s1` and `s2`, as well as an integer cost `x`. Our goal is to transform `s1` into `s2` through a series of operations, each of which incurs some cost. There are two types of operations we can perform:

1. We can flip (change from `0` to `1` or from `1` to `0`) the bits at two chosen indices `i` and `j` in `s1`. This operation has a fixed cost of `x`.
2. We can flip the bits at consecutive indices `i` and `i+1` in `s1`, provided `i < n - 1`. This operation has a cost of `1`.

Our objective is to calculate the minimum cost required to make `s1` identical to `s2`. If it's not possible to make the strings identical, we should return `-1`.

An important observation is that an even number of differences between `s1` and `s2` is required to make the transformation feasible because each flip operation changes the state of exactly two bits.

Intuition

The intuition behind the solution begins with the understanding that we are only able to change two characters at a time. If there is an odd number of differing characters between `s1` and `s2`, it's impossible to achieve equality, and the function returns `-1`. When the number of differing characters is even, there is potential to reach the goal.

We use an array `idx` to store the indices of `s1` where the characters differ from those in `s2`. The first step in our approach is a quick check of the length of `idx`: if it is odd, we return `-1` right away as it would be impossible to make the strings equal.

Next, we define a function `dfs(i, j)` to represent the minimum cost needed to flip the characters in the sub-array `idx[i..j]`. We're looking for the answer to `dfs(0, len(idx) - 1)`.

The calculation for `dfs(i, j)` considers three possible scenarios:

1. **Flipping endpoints (`i` and `j`):** We flip characters at the two endpoints, using the first operation type at a cost of `x`. Then we recursively call `dfs` to handle the sub-problem for `idx[i+1]` to `idx[j-1]`.
2. **Flipping near `i`:** We flip the characters at `idx[i]` and `idx[i+1]` using the second operation type. We add the distance between `idx[i]` and `idx[i+1]` to the cost of the recursive `dfs` call from `idx[i+2]` to `idx[j]`.
3. **Flipping near `j`:** We flip the characters at `idx[j-1]` and `idx[j]` in similar fashion, adding the distance between these indices to the cost of the recursive `dfs` call for `idx[i]` to `idx[j-2]`.

We seek the minimum cost of these three approaches to be the result for `dfs(i, j)`.

Memoization is used to cache the results of `dfs(i, j)` to avoid recomputing sub-problems. This is done by storing previously calculated results of `dfs(i, j)` and retrieving them instead of recalculating when they are requested again.

This solution approach uses depth-first search combined with memoization (caching) to efficiently solve the problem.

Solution Approach

The given solution utilizes Depth-First Search (DFS) with memoization as its core algorithm. The DFS explores all combinations of pairs that can be flipped to transform `s1` into `s2`, while memoization ensures that the minimum cost for any given range of indices is calculated only once and then reused. The use of memoization is a common pattern when there is a possibility of computing the same values repeatedly, as seen in many dynamic programming problems.

The code defines a recursive function `dfs(i, j)`, which calculates the minimum cost of making subarrays of `idx`, specifically `idx[i..j]`, identical. The Python `@cache` decorator is used to store the results of `dfs(i, j)` calls, so identical sub-problems are not re-calculated from scratch—a typical memoization pattern.

There are three potential scenarios covered by the recursive function:

1. **Flipping endpoints (`i` and `j`):** The cost is calculated by flipping the characters at `i` and `j`, which consumes a fixed cost of `x`.

```
a = dfs(i + 1, j - 1) + x
```

2. **Flipping near the start index (`i` and `i + 1`):** The cost is the cost of flipping consecutive characters starting from `i` which costs `1`. However, since `1` unit cost is specific to two adjacent characters, the actual cost is the distance between `idx[i]` and `idx[i + 1]`. This distance is then added to the cost of the recursive call starting from `i + 2`.

```
b = dfs(i + 2, j) + idx[i + 1] - idx[i]
```

3. **Flipping near the end index (`j - 1` and `j`):** This is similar to the previous case, but here we start flipping from `j`.

```
c = dfs(i, j - 2) + idx[j] - idx[j - 1]
```

In each call of `dfs(i, j)`, the function computes the minimum cost among the three scenarios (`a`, `b`, and `c`).

The index array `idx` helps to keep track of the indices where the two strings differ, emphasizing that we only care about the positions that need transformation. This can be seen as a space optimization over considering the entire string for each recursive call.

The condition `if m & 1` checks whether we have an odd number of differing bits by using bitwise `AND` with `1`. If `m` is odd, it immediately returns `-1`, as an odd number of differing bits cannot be solved with the given operations.

Lastly, `return dfs(0, m - 1)` kicks off the recursive calls with the full range of differing indices.

This method provides a balance between the brute force approach (which would be too slow) and a more optimized approach that keeps the time complexity in check by only re-calculating necessary sub-problems when needed. Exploring all combinations through DFS ensures that all possibilities are considered, while memoization prevents unnecessary work, thus optimizing the solution.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach:

Suppose we have the binary strings `s1 = "1101"` and `s2 = "1000"` with an integer cost `x = 3`. We want to transform `s1` into `s2` with the minimum cost using the operations described.

First, we identify the indices where `s1` and `s2` differ, which are `idx = [1, 3]`. There are two bits different, so it's an even number, and thus, the problem is solvable.

Let's apply the Depth-First Search (DFS) approach with memoization step by step:

1. The function `dfs(0, 1)` is called to solve the entire range of differing indices.
2. Inside `dfs(0, 1)`, we calculate the cost of flipping endpoints (`i = 0`, `j = 1`). The cost for flipping `idx[0]` and `idx[1]` is `x = 3`.

```
a = dfs(1 - 1, 1 + 1) + x
a = dfs(0, 2) + 3  # Out of bounds, but assume dfs(0, 2) = 0
a = 0 + 3
a = 3
```

3. Then we compute the cost of flipping near `i` (`i = 0` and `i + 1 = 1`). The cost of flipping `idx[0]` and `idx[1]` is distance `idx[1] - idx[0]`, which is `3 - 1 = 2`.

```
b = dfs(0 + 2, 1) + idx[1] - idx[0]
b = dfs(2, 1)  # Out of bounds, but assume dfs(2, 1) = 0
b = 0 + 2
b = 2
```

4. Since there are only two indices, there's no distinct cost computation for flipping near `j`, as it would be the same as flipping near `i`.

The minimum cost among the scenarios is `min(a, b)`, which is `min(3, 2)`. Thus, the minimum cost is `2`.

However, we realize that since we used a hypothetical out-of-bounds result for the recursive `dfs(0, 2)` and `dfs(2, 1)`, in a real situation, the actual function would need to check for boundaries and return appropriate values (usually 0 if there are no more characters to flip). This is handled in the implementation where each recursive call checks for the validity of the indices.

In this case, only the operation of flipping bits at consecutive indices `i` and `i+1` (the second type of operation) is applied, resulting in a total minimum cost of `2` to transform `s1` into `s2`.

This example demonstrates how the algorithm applies DFS with memoization to find the minimum cost to transform one binary string into another efficiently.

Python Solution

```python
from functools import lru_cache  # Remember to import the necessary caching decorator

class Solution:
    def min_operations(self, s1: str, s2: str, x: int) -> int:
        # Helper function to recursively determine the minimum operations
        # necessary to make substrings equal.
        @lru_cache(maxsize=None)  # Cache results to avoid recomputation
        def dfs(i: int, j: int) -> int:
            # Base case: If the pointers cross each other, return 0 since we don't need to make operations.
            if i > j:
                return 0

            # Recurrence relation:
            # 1. Swap any character in s1 with 'x' and solve for the smaller substring.
            swap_with_x = dfs(i + 1, j - 1) + x

            # 2. Move the first pointer by two and add the difference between current and previous indices.
            # This essentially skips one element that doesn't need changes.
            skip_left = dfs(i + 2, j) + idx[i + 1] - idx[i]

            # 3. Move the second pointer by two and add the difference between current and previous indices.
            # This skips one element from the end that doesn't need changes.
            skip_right = dfs(i, j - 2) + idx[j] - idx[j - 1]

            # Return the minimum operations among the three alternatives.
            return min(swap_with_x, skip_left, skip_right)

        # Step 1: Identify indices where the characters in s1 and s2 differ.
        idx = [i for i in range(len(s1)) if s1[i] != s2[i]]
        # Number of indices that are different in s1 and s2.
        num_diff_indices = len(idx)

        # Step 2: If there is an odd number of different indices, it's impossible to make s1 equal to s2, return -1.
        if num_diff_indices % 2:
            return -1

        # Step 3: Apply the dfs function to the entire range of different indices.
        return dfs(0, num_diff_indices - 1)

# If intending to use, the Solution class is meant to be instantiated first
# solution = Solution()
# and then the min_operations method can be called with the appropriate parameters:
# result = solution.min_operations(s1, s2, x)

```

Java Solution

```java
class Solution {
    private List<Integer> mismatchIndices = new ArrayList<>(); // Store indices of mismatched characters
    private Integer[][] memoization; // Memoization array for dynamic programming
    private int operationCost; // Cost of a swap operation

    // Calculate the minimum number of operations to make s1 equal to s2 given the operation cost
    public int minOperations(String s1, String s2, int operationCost) {
        int length = s1.length();

        // Identify all mismatched character positions and add them to the list
        for (int i = 0; i < length; ++i) {
            if (s1.charAt(i) != s2.charAt(i)) {
                mismatchIndices.add(i);
            }
        }

        int mismatches = mismatchIndices.size();

        // If the number of mismatches is odd, return -1 since they cannot be paired
        if (mismatches % 2 == 1) {
            return -1;
        }

        this.operationCost = operationCost;
        memoization = new Integer[mismatches][mismatches];

        // Initiate dynamic programming with a depth-first search
        return depthFirstSearch(0, mismatches - 1);
    }

    // Recursive depth-first search to find the minimum operation cost
    private int depthFirstSearch(int left, int right) {
        // When left > right, all characters are matched
        if (left > right) {
            return 0;
        }
        // Return the already computed value if available
        if (memoization[left][right] != null) {
            return memoization[left][right];
        }

        // Calculate the cost by swapping adjacent indices
        int cost = depthFirstSearch(left + 1, right - 1) + operationCost;

        // Calculate the cost by changing the left-most mismatched character
        if (left + 1 <= right) {
            int leftSwapCost = mismatchIndices.get(left + 1) - mismatchIndices.get(left);
            cost = Math.min(cost, depthFirstSearch(left + 2, right) + leftSwapCost);
        }

        // Calculate the cost by changing the right-most mismatched character
        if (right - 1 >= left) {
            int rightSwapCost = mismatchIndices.get(right) - mismatchIndices.get(right - 1);
            cost = Math.min(cost, depthFirstSearch(left, right - 2) + rightSwapCost);
        }

        // Store the computed value in the memoization array
        memoization[left][right] = cost;
        return cost;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <string>
#include <cstring>
#include <functional>

class Solution {
public:
    int minOperations(std::string s1, std::string s2, int swapCost) {
        std::vector<int> mismatchedIndices;
        // Collect indices where characters from s1 and s2 differ
        for (int i = 0; i < s1.size(); ++i) {
            if (s1[i] != s2[i]) {
                mismatchedIndices.push_back(i);
            }
        }

        int mismatchCount = mismatchedIndices.size();

        // If there's an odd number of mismatches, it's impossible to make strings equal
        if (mismatchCount % 2 != 0) {
            return -1;
        }

        // If there are no mismatches, no operations are required
        if (mismatchCount == 0) {
            return 0;
        }

        int dp[mismatchCount][mismatchCount];

        // Initialize the dynamic programming table with -1 (uncomputed state)
        memset(dp, -1, sizeof(dp));

        // Depth-First Search (DFS) to find minimum cost through dynamic programming
        std::function<int(int, int)> dfs = [&](int left, int right) {
            // Base case: if left index greater than right, no operations needed
            if (left > right) {
                return 0;
            }
            // If already computed, just return the value
            if (dp[left][right] != -1) {
                return dp[left][right];
            }
            // Calculate the minimum cost among three choices
            dp[left][right] = std::min({
                dfs(left + 1, right - 1) + swapCost,                                      // Swap the outer characters
                dfs(left + 2, right) + mismatchedIndices[left + 1] - mismatchedIndices[left], // Resolve leftmost pair
                dfs(left, right - 2) + mismatchedIndices[right] - mismatchedIndices[right - 1]  // Resolve rightmost pair
            });
            return dp[left][right];
        };

        // Perform DFS and return the minimal operation cost
        return dfs(0, mismatchCount - 1);
    }
};

```

Typescript Solution

```typescript
// Function to calculate the minimum operations needed to make s1 equal to s2
function minOperations(s1: string, s2: string, x: number): number {

    // Array to hold indices where characters in s1 and s2 differ
    const differingIndices: number[] = [];

    // Populate the differingIndices array with the positions at which s1 and s2 differ
    for (let i = 0; i < s1.length; ++i) {
        if (s1[i] !== s2[i]) {
            differingIndices.push(i);
        }
    }

    // The number of differing positions
    const numDiffering = differingIndices.length;

    // If number of differing positions is odd, return -1 (not possible to make strings equal)
    if (numDiffering % 2 === 1) {
        return -1;
    }

    // If strings are already equal, no operations are needed
    if (numDiffering === 0) {
        return 0;
    }

    // Initialize a memoization array for dynamic programming
    const memo: number[][] = Array.from({ length: numDiffering }, () =>
        Array.from({ length: numDiffering }, () => -1));

    // Recursive depth-first search function to find the minimum operations
    const dfs = (leftIndex: number, rightIndex: number): number => {
        if (leftIndex > rightIndex) {
            return 0;
        }
        if (memo[leftIndex][rightIndex] !== -1) {
            return memo[leftIndex][rightIndex];
        }

        // Compute the minimum operations
        // Case 1: Swap a pair of differing characters
        memo[leftIndex][rightIndex] = dfs(leftIndex + 1, rightIndex - 1) + x;

        // Case 2: Change one character and move to the next pair
        memo[leftIndex][rightIndex] = Math.min(memo[leftIndex][rightIndex],
            dfs(leftIndex + 2, rightIndex) + differingIndices[leftIndex + 1] - differingIndices[leftIndex]);

        // Case 3: Change the other character and move to the previous pair
        memo[leftIndex][rightIndex] = Math.min(memo[leftIndex][rightIndex],
            dfs(leftIndex, rightIndex - 2) + differingIndices[rightIndex] - differingIndices[rightIndex - 1]);

        return memo[leftIndex][rightIndex];
    };

    // Call the dfs function starting from the first and last index of the differing positions
    return dfs(0, numDiffering - 1);
}

```

Time and Space Complexity

The time complexity of the given code is `O(n^2)` where `n` is the length of `idx`, which derives from the number of mismatched characters between `s1` and `s2`. For each call to `dfs(i, j)`, there are up to three recursive calls, but the function is memoized using the `@cache` decorator. This ensures that each possible state `(i, j)` is calculated only once. The state `(i, j)` represents a subproblem of finding the minimum operations for the substring from `idx[i]` to `idx[j]`, and there are `O(n^2)` such states because `i` can range from `0` to `n-1` and `j` can range from `i` to `n-1`.

The space complexity of the code is also `O(n^2)` due to the memoization cache that stores the computed result for each unique `(i, j)` pair. Each entry in the cache takes constant space, and since there are `O(n^2)` pairs, the overall space used by the cache is `O(n^2)`, plus the space used for the `idx` list and the recursion stack.