2606. Find the Substring With Maximum Cost

Medium | Array | Hash Table | String | Dynamic Programming | Leetcode Link

Problem Description

Given a string `s`, a unique character string `chars`, and a corresponding array of integer values `vals` (with the same length as `chars`), the task is to calculate the maximum cost possible among all substrings of the string `s`. A substring is any contiguous sequence of characters within the string.

The cost of a substring is the sum of the values of each character in the substring. The value of a character is:

- If the character isn't present in `chars`, its value is its 1-indexed position in the English alphabet (e.g., 'a' would be 1, 'b' would be 2, and so on up to 'z' being 26).
- If the character is present in `chars`, its value is the corresponding value from `vals` at its index.

An empty substring is considered to have a cost of `0`.

The problem requires determining the maximum sum that can be achieved by any substring's cost within the string `s` by applying the rules above.

Intuition

This problem is a variant of the classical maximum subarray sum problem (also known as Kadane's algorithm), where instead of finding a subarray with a maximum sum in an integer array, the objective is to find a substring with the maximum cost in a string with custom-defined character values.

The intuition behind the solution is to translate each character's cost into an integer and then apply dynamic programming to find the maximum subarray sum, which corresponds to the maximum cost substring in our problem.

As we parse the string `s`, we keep a running tally of the substring cost by adding the value of each character to a running sum `f`. If `f` drops below zero, it means the current substring is reducing the overall cost, so we reset it to zero to start a new potential maximum cost substring.

Concurrently, we track the maximum cost seen thus far in a separate variable `ans`, updating it whenever the running sum `f` exceeds `ans`. The maximum subarray sum problem essentially becomes maintaining the running sum and keeping track of the maximum sum we have seen.

The solution approach involves maintaining two variables:

- `f`, the running sum which is updated as we traverse the string.
- `ans`, the maximum sum encountered so far.

As we iterate over the string `s`, we calculate the value of each character and update `f`. If at any point `f` is less than zero (which would never contribute to a maximum sum), we reset it to zero. With each new character, we evaluate if adding its value to `f` yields a new maximum. If it does, we update `ans` with the value of `f`.

By the end of the iteration, `ans` holds the highest cost possible for any substring of `s`, which is the answer we return.

Solution Approach

The solution for the maximum cost substring problem is implemented using two algorithms: the prefix sum updating and conditional resetting approach (leveraging a dynamic programming technique similar to Kadane's algorithm). Here's how it's broken down:

Prefix Sum + Maintaining Minimum Prefix Sum

- **Prefix Sum**: As we traverse through each character `c` in string `s`, we obtain its value `v`. This is determined either by finding the character's corresponding value in the character-to-value mapping `d` or calculating its alphabetical index if it's not present in the custom chars string. The prefix sum `tot` is updated by adding the value `v` to it: `tot = tot + v`.
- **Maintaining Minimum Prefix Sum**: To determine the cost of the maximum cost substring ending with `c`, we need to consider the minimum prefix sum encountered before `c`. So we subtract this minimum `mi` from the current prefix sum `tot` to get `tot - mi`, and then we compare this to our running answer `ans`. We then update `ans` to be the maximum of itself or the new substring cost: `ans = max(ans, tot - mi)`. At the same time, we update `mi` to be the minimum of itself or the current prefix sum `tot`: `mi = min(mi, tot)`.

Converting to Maximum Subarray Sum Problem

Instead of maintaining and updating both the total `tot` and minimum prefix sum `mi`, we can simplify it with a single variable `f` that keeps track of the cost of the maximum cost substring ending with the current character `c`:

- In each iteration for character `c`, we update `f` to be the maximum between `f` and 0, then add the current character's value `v`: `f = max(f, 0) + v`. This is essentially the step where we consider starting a new substring (if `f` was negative) or continuing with the current one (if `f` was non-negative).
- We then update the answer `ans` to be the maximum of itself or the newly calculated `f`: `ans = max(ans, f)`.

The implementation ensures that at each step, we're considering substrings that either end at the current character or are empty (if a substring with a positive cost does not exist up to that point). By continuously comparing and updating `f` and `ans` as we iterate through the string, we ensure that we find the optimal solution. The final value of `ans` is the maximum cost of any substring in `s`.

The time complexity of this approach is `O(n)`, where `n` is the length of the string `s`. The space complexity is `O(C)`, where `C` is the size of the character set. Since we're dealing with lowercase English letters in this problem, `C` is 26.

Example of Implementation

Here is the pseudocode to illustrate the algorithm:

```
initialize d as a mapping from chars to vals
initialize ans and f to 0  # 'ans' for the final answer, 'f' for the cost of current substring
for each character c in s:
    v = d.get(c, ord(c) - ord('a') + 1)  # Calculate the value of c
    f = max(f, 0) + v  # Update the running sum 'f'
    ans = max(ans, f)  # Update the answer 'ans'
return ans
```

In this pseudocode, `d.get(c, ord(c) - ord('a') + 1)` is getting the value of the character `c` either from the mapping `d` or calculating its alphabet index if it's not in `d`. The `max(f, 0)` is where we reset the running sum `f` if it's negative before adding the new character's value. This logic ensures we're always considering substrings that have a non-negative cost. Finally, we update `ans` with the current running sum `f`, ensuring that after processing all characters, `ans` reflects the maximum cost achievable.

Example Walkthrough

Let's consider a small example to illustrate the solution approach.

Suppose the input string `s` is "dcb", the character string `chars` is "bd" and the corresponding array of integer values `vals` is [4, 6]. This means that the character 'b' has a value of 6 and 'd' has a value of 4 based on `vals`. Characters not in `chars` have a value equal to their position in the English alphabet.

We initialize `d` as a mapping from special `chars` to their associated `vals`, where `d = {'b': 6, 'd': 4}`. We then initialize two variables: `ans` for the maximum sum answer and `f` for the running sum of the cost of the current substring. Both are initially `0`.

Now, we start iterating through each character `c` in `s`.

- First, we look at character `c` = 'd':
  - The character 'd' is in our mapping `d` with a value of 4. So `v = 4`.
  - The running sum `f` becomes `max(f, 0) + v`. Since `f` is `0`, we have `f = 4`.
  - We update `ans`: `ans = max(ans, f)`, which gives us `ans = 4`.
- Next, we move to `c` = 'c':
  - 'c' is not in our mapping `d`, so its value is its alphabetic index, which is 3. `v = 3`.
  - Update `f` to `max(f, 0) + v`. `f` is currently `4`, so now `f = 4 + 3`, `f = 7`.
  - Update `ans`: `ans = max(ans, f)`, `ans = max(4, 7)`, so `ans = 7`.
- Finally for `c` = 'b':
  - 'b' is in `d` with a value of 6. `v = 6`.
  - Update `f` to `max(f, 0) + v`. `f` is `7`, so `f = 7 + 6`, `f = 13`.
  - Update `ans`: `ans = max(ans, f)`, `ans = 13`.

At the end of the string, the maximum cost `ans` is `13`, which corresponds to the substring "dcb", as no smaller substring provides a higher cost. The final answer is `13`.

The time complexity of this algorithm is `O(n)` where `n` is the length of the string `s`, and the space complexity is `O(C)` where `C` is the size of the character set given by `chars`, with a constant space optimization for the English alphabet having 26 characters.

Python Solution

```python
class Solution:
    def maximumCostSubstring(self, s: str, chars: str, vals: List[int]) -> int:
        # Create a dictionary with characters as keys and corresponding values as dictionary values
        char_to_value = {character: value for character, value in zip(chars, vals)}

        # Initialize variables to keep track of the maximum cost and the current cost
        max_cost = running_cost = 0

        # Iterate through each character in the string 's'
        for char in s:
            # Lookup the character value in the dictionary, if not found, calculate the default value
            # The default value is the ASCII code of the character minus the ASCII code of 'a', plus 1.
            value = char_to_value.get(char, ord(char) - ord('a') + 1)

            # Update the running cost, reset to zero if it becomes negative
            running_cost = max(running_cost, 0) + value

            # Update the maximum cost encountered so far
            max_cost = max(max_cost, running_cost)

        # Return the final maximum cost calculated
        return max_cost

```

Java Solution

```java
class Solution {
    public int maximumCostSubstring(String s, String chars, int[] values) {
        // Initialize an array to store values for 'a' to 'z'
        int[] costMapping = new int[26];

        // Fill the array with default values as index + 1 (it seems to be placeholder values)
        for (int i = 0; i < costMapping.length; ++i) {
            costMapping[i] = i + 1;
        }

        // Map the cost of characters given in `chars` using `vals`
        for (int i = 0; i < chars.length(); ++i) {
            // Use the character's ASCII value to find the correct index in `costMapping`,
            // and update that position with the value from `vals`
            costMapping[chars.charAt(i) - 'a'] = values[i];
        }

        int maxCost = 0;   // Maximum cost encountered so far
        int currentCost = 0; // Current cost while evaluating the substring
        int stringLength = s.length(); // Length of the string `s`

        // Iterate through each character of the input string `s`
        for (int i = 0; i < stringLength; ++i) {
            // Get the cost of the current character
            int charCost = costMapping[s.charAt(i) - 'a'];

            // Update the currentCost: reset to 0 if negative, or add the value of the current character
            currentCost = Math.max(currentCost, 0) + charCost;

            // Update the maximum cost encountered so far
            maxCost = Math.max(maxCost, currentCost);
        }

        // Return the maximum cost found for the substring
        return maxCost;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to calculate the maximum cost substring where 's' is the input string,
    // 'chars' is a list of characters, and 'vals' is the corresponding list of values.
    int maximumCostSubstring(string s, string chars, vector<int>& vals) {
        // Initialize a vector with 26 elements (for each letter of the alphabet) with values 1 to 26.
        vector<int> delta(26);
        iota(delta.begin(), delta.end(), 1);

        // Store the size of 'chars' string for later use.
        int charListSize = chars.size();

        // Replace the initial values in 'delta' with the corresponding values from 'vals'.
        for (int i = 0; i < charListSize; ++i) {
            delta[chars[i] - 'a'] = vals[i];
        }

        // Initialize variables to keep track of maximum cost and the current running fragment cost.
        int maxCost = 0, currentFragmentCost = 0;

        // Calculate the maximum cost substring by iterating through each character of the string 's'.
        for (char& c : s) {
            // Get the value of the current character from the delta vector.
            int value = delta[c - 'a'];

            // Calculate the current fragment cost. If the current fragment cost dips below 0,
            // reset it to 0, and add the value of the current character.
            currentFragmentCost = max(currentFragmentCost, 0) + value;

            // Update the maxCost if the currentFragmentCost is larger than the maxCost.
            maxCost = max(maxCost, currentFragmentCost);
        }

        // Return the maximum cost found.
        return maxCost;
    }
};

```

Typescript Solution

```typescript
// Function to calculate the maximum cost of a non-empty substring where cost is
// determined by associated values of characters.
function maximumCostSubstring(s: string, chars: string, values: number[]): number {
    // Create an array to hold the cost value of each alphabet letter, initialized
    // with values 1 to 26 to represent 'a' to 'z'.
    const costs: number[] = Array.from({ length: 26 }, (_, index) => index + 1);

    // Override the cost values based on the 'chars' and 'vals' input.
    // This assigns the custom values to the specific characters in 'chars'.
    for (let i = 0; i < chars.length; ++i) {
        costs[chars.charCodeAt(i) - 'a'.charCodeAt(0)] = values[i];
    }

    // Initialize variables to track the maximum cost, current total cost,
    // and the minimum cost encountered so far.
    let maxCost = 0;
    let totalCost = 0;
    let minCost = 0;

    // Loop through each character in the input string
    for (const char of s) {
        // Add the cost of the current character to the total cost
        totalCost += costs[char.charCodeAt(0) - 'a'.charCodeAt(0)];
        // Update the maximum cost if the current total cost minus the minimum cost
        // encountered so far is greater than the current maximum cost.
        maxCost = Math.max(maxCost, totalCost - minCost);
        // Update the minimum cost encountered so far if needed
        minCost = Math.min(minCost, totalCost);
    }

    // Return the maximum cost of a substring computed
    return maxCost;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code is `O(n)` where `n` is the length of the string `s`. This is because the algorithm iterates over each character of the string exactly once, performing a constant amount of work for each character by looking up a value in the dictionary and updating the variables `f` and `ans`.

Space Complexity

The space complexity of the code is `O(C)` where `C` is the size of the character set. In this problem, the character set size is static and equals `26` because the lowercase alphabet is used. The dictionary `d` contains at most `C` key-value pairs, where `C` represents the unique characters in `chars`, which are in turn mapped to `vals`.