Distinct Subsequences

Given a string **S** and a string **T**, count the number of distinct subsequences of **T** in **S**.

A subsequence of a string is a new string which is formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters. (ie, `"ACE"` is a subsequence of `"ABCDE"` while `"AEC"` is not).

Here is an example:
S = `"rabbbit"`, **T** = `"rabbit"`

Return `3`.

Solution 1

My solution is using O(n^2) space and running in O(n^2) time. I wonder is there a better way to do that which consumes less memory? I guess run time could not be improved though. Any thought/input would be highly appreciated, thanks!

```
/**
 * Solution (DP):
 * We keep a m*n matrix and scanning through string S, while
 * m = T.length() + 1 and n = S.length() + 1
 * and each cell in matrix Path[i][j] means the number of distinct subsequences o
f
 * T.substr(1...i) in S(1...j)
 *
 * Path[i][j] = Path[i][j-1]            (discard S[j])
 *              +     Path[i-1][j-1]    (S[j] == T[i] and we are going to use S[j
])
 *                 or 0                 (S[j] != T[i] so we could not use S[j])
 * while Path[0][j] = 1 and Path[i][0] = 0.
 */
int numDistinct(string S, string T) {
    int m = T.length();
    int n = S.length();
    if (m > n) return 0;    // impossible for subsequence
    vector<vector<int>> path(m+1, vector<int>(n+1, 0));
    for (int k = 0; k <= n; k++) path[0][k] = 1;    // initialization

    for (int j = 1; j <= n; j++) {
        for (int i = 1; i <= m; i++) {
            path[i][j] = path[i][j-1] + (T[i-1] == S[j-1] ? path[i-1][j-1] : 0);
        }
    }

    return path[m][n];
}
```

written by dragonmigo original link here

Solution 2

The idea is the following:

- we will build an array `mem` where `mem[i+1][j+1]` means that `S[0..j]` contains `T[0..i]` that many times as distinct subsequences. Therefor the result will be `mem[T.length()][S.length()]`.
- we can build this array rows-by-rows:
  - the first row must be filled with 1. That's because the empty string is a subsequence of any string but only 1 time. So `mem[0][j] = 1` for every `j`. So with this we not only make our lives easier, but we also return correct value if `T` is an empty string.
  - the first column of every rows except the first must be 0. This is because an empty string cannot contain a non-empty string as a substring -- the very first item of the array: `mem[0][0] = 1`, because an empty string contains the empty string 1 time.

So the matrix looks like this:

```
  S 0123....j
T +----------+
  |1111111111|
0 |0         |
1 |0         |
2 |0         |
. |0         |
. |0         |
i |0         |
```

From here we can easily fill the whole grid: for each `(x, y)`, we check if `S[x] == T[y]` we add the previous item and the previous item in the previous row, otherwise we copy the previous item in the same row. The reason is simple:

- if the current character in S doesn't equal to current character T, then we have the same number of distinct subsequences as we had without the new character.
- if the current character in S equal to the current character T, then the distinct number of subsequences: the number we had before **plus** the distinct number of subsequences we had with less longer T and less longer S.

An example: `S: [acdabefbc]` and `T: [ab]`

first we check with `a`:

```
           *  *
      S = [acdabefbc]
mem[1] = [0111222222]
```

then we check with `ab`:

```
               *  * ]
      S = [acdabefbc]
mem[1] = [0111222222]
mem[2] = [0000022244]
```

And the result is 4, as the distinct subsequences are:

```
      S = [a   b    ]
      S = [a      b ]
      S = [   ab    ]
      S = [   a   b ]
```

See the code in Java:

```java
public int numDistinct(String S, String T) {
    // array creation
    int[][] mem = new int[T.length()+1][S.length()+1];

    // filling the first row: with 1s
    for(int j=0; j<=S.length(); j++) {
        mem[0][j] = 1;
    }

    // the first column is 0 by default in every other rows but the first, which we need.

    for(int i=0; i<T.length(); i++) {
        for(int j=0; j<S.length(); j++) {
            if(T.charAt(i) == S.charAt(j)) {
                mem[i+1][j+1] = mem[i][j] + mem[i+1][j];
            } else {
                mem[i+1][j+1] = mem[i+1][j];
            }
        }
    }

    return mem[T.length()][S.length()];
}
```

written by balint original link here

Solution 3

Could someone please clarify this problem to me?

> Given a string S and a string T, count the number of distinct subsequences of T in S.
>
> A subsequence of a string is a new string which is formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters. (ie, "ACE" is a subsequence of "ABCDE" while "AEC" is not).
>
> Here is an example: S = "rabbbit", T = "rabbit" count = 3

If I understood correctly, we need to find all distinct subsequences of T and see how many, if any appear in s. How does that equal to 3 in the given example?

written by princessmaja original link here