Previous Next

115. Distinct Subsequences

April 12, 2020 | 2.2K views

Average Rating: 5 (24 votes)

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

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).

It's guaranteed the answer fits on a 32-bit signed integer.

Example 1:

```
Input: S = "rabbbit", T = "rabbit"
Output: 3
Explanation:
As shown below, there are 3 ways you can generate "rabbit" from S.
(The caret symbol ^ means the chosen letters)

rabbbit
^^^^ ^^
rabbbit
^^ ^^^^
rabbbit
^^^ ^^^
```

Example 2:

```
Input: S = "babgbag", T = "bag"
Output: 5
Explanation:
As shown below, there are 5 ways you can generate "bag" from S.
(The caret symbol ^ means the chosen letters)

babgbag
^^ ^
babgbag
^^    ^
babgbag
^    ^^
babgbag
  ^  ^^
babgbag
    ^^^
```

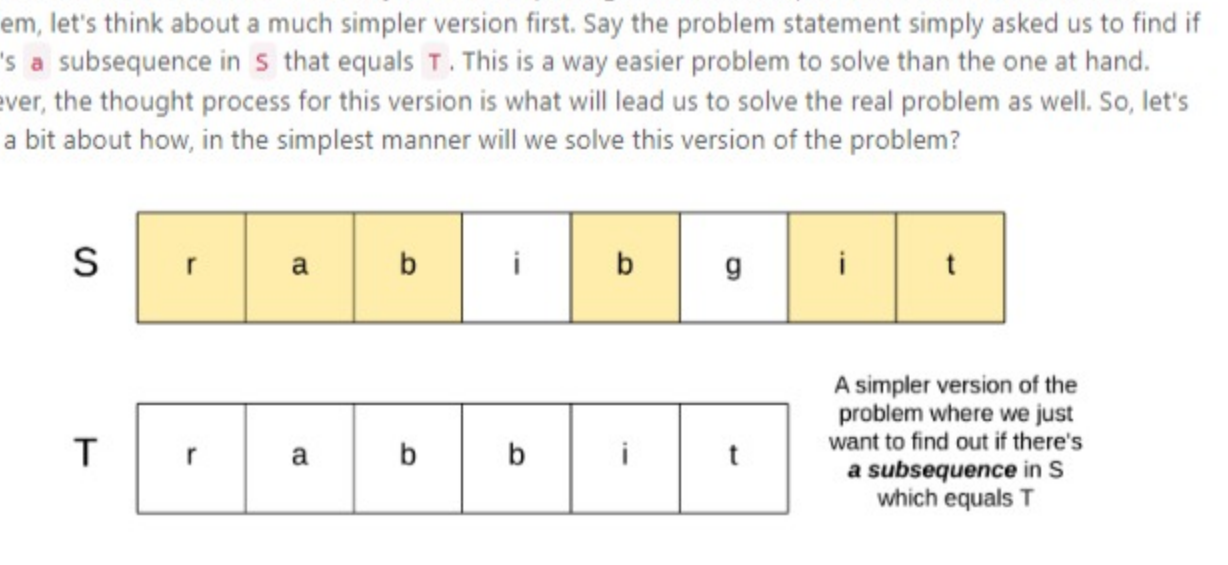
Solution

This is one of the best problems for illustrating the transition from a recursive solution to an iterative one and finally, to a space optimized iterative solution. Trust me, you're going to have a lot of fun solving this problem and this is the kind of problem that an interviewer may ask in an interview to grill the candidate on the various aspects of optimization. Start with the recursive solution and then drill your way through to an iterative solution with a highly reduced space complexity. Without further ado, let's look at the solutions.

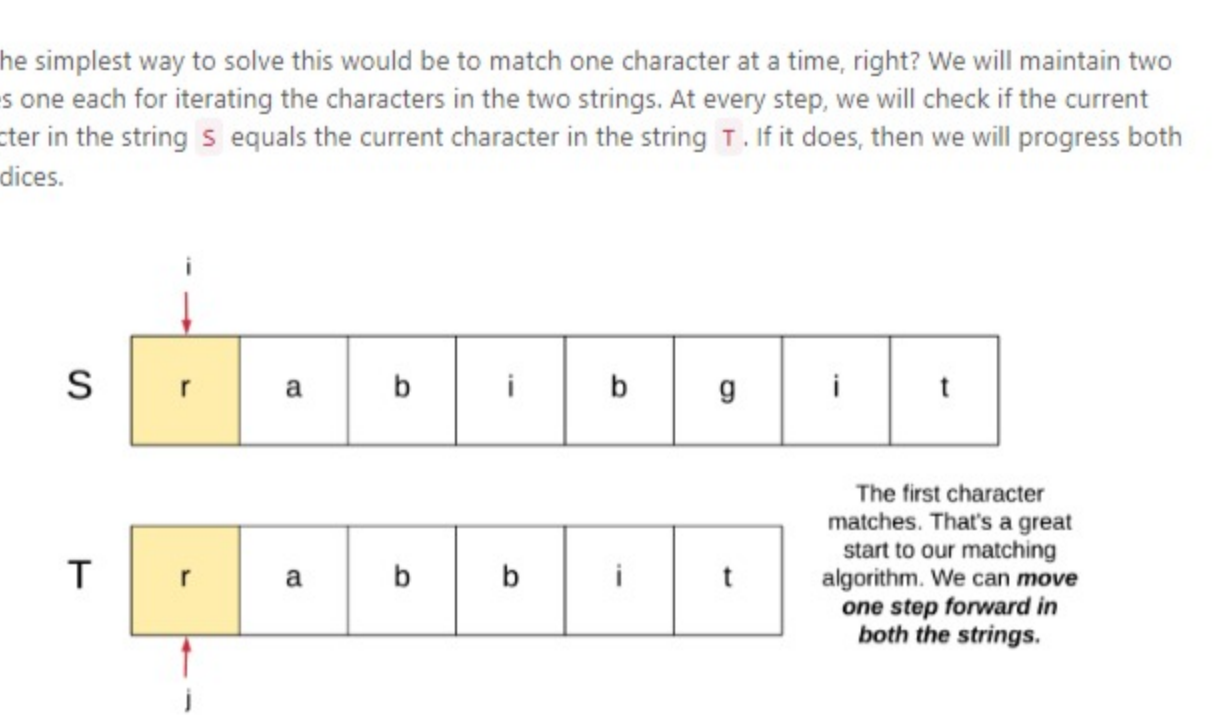
Approach 1: Recursion + Memoization

Intuition

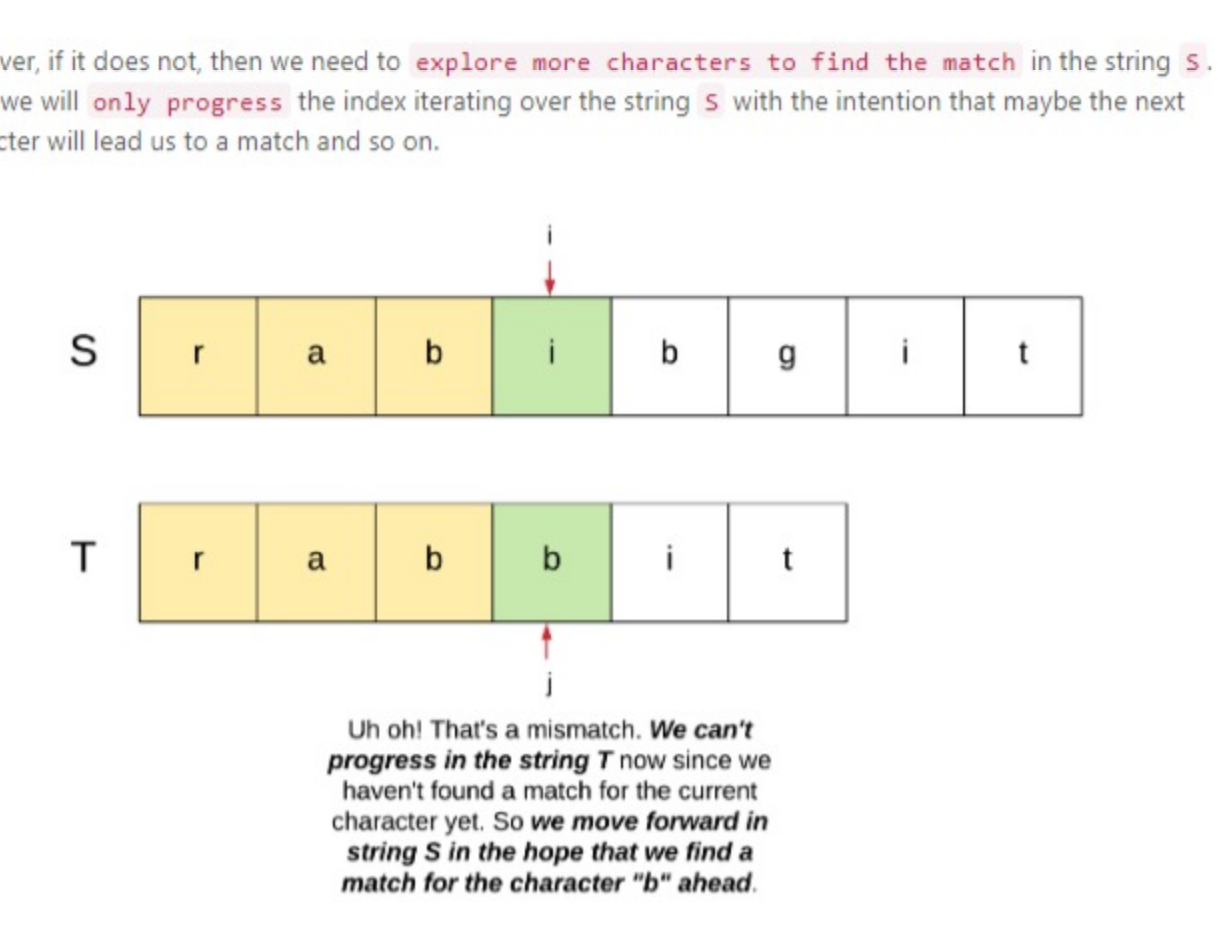
This problem is all about the "choice" you make in picking out the subsequences. Before we solve this problem, let's think about a much simpler version first. Say the problem statement simply asked us to find if there's `a` subsequence in `S` that equals `T`. This is a way easier problem to solve than the one at hand. However, the thought process for this version is what will lead us to solve the real problem as well. So, let's think a bit about how, in the simplest manner will we solve this version of the problem?

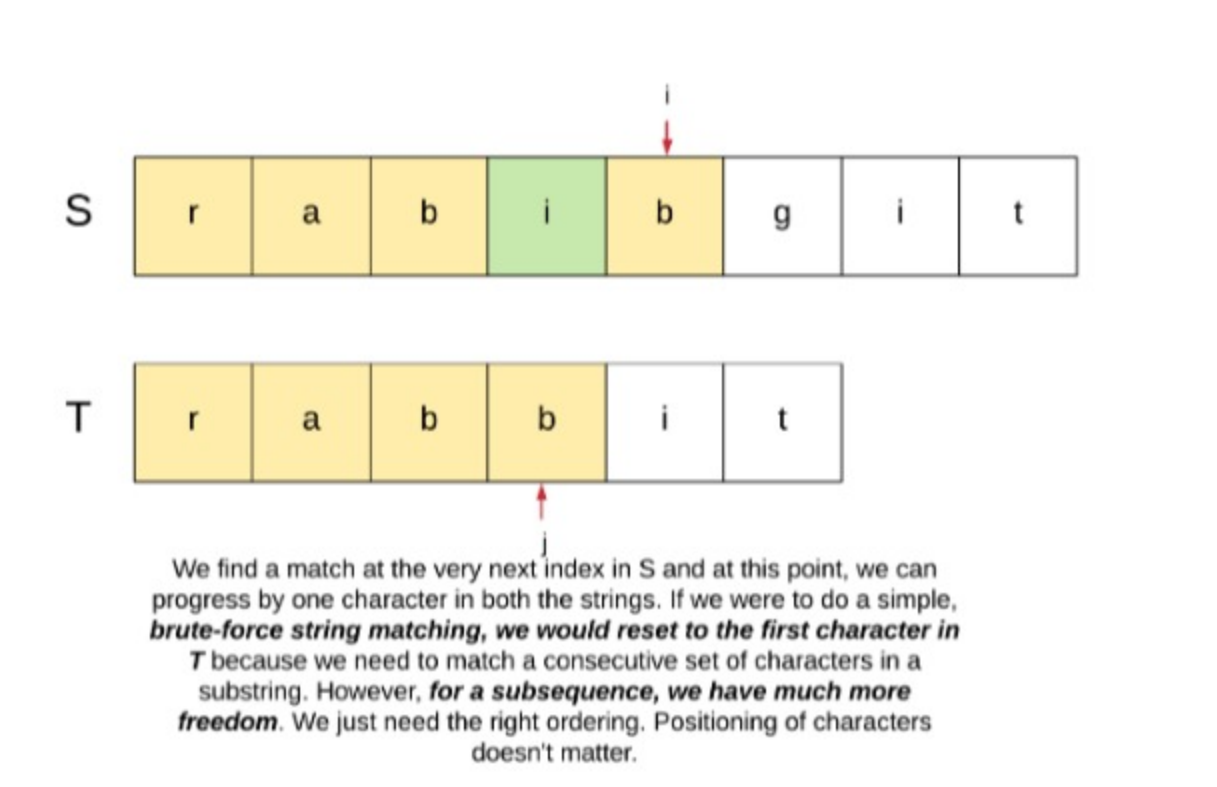


Well the simplest way to solve this would be to match one character at a time, right? We will maintain two indices one each for iterating the characters in the two strings. At every step, we will check if the current character in the string `S` equals the current character in the string `T`. If it does, then we will progress both the indices.



However, if it does not, then we need to `explore more characters to find the match` in the string `S`. Thus, we will `only progress` the index iterating over the string `S` with the intention that maybe the next character will lead us to a match and so on.

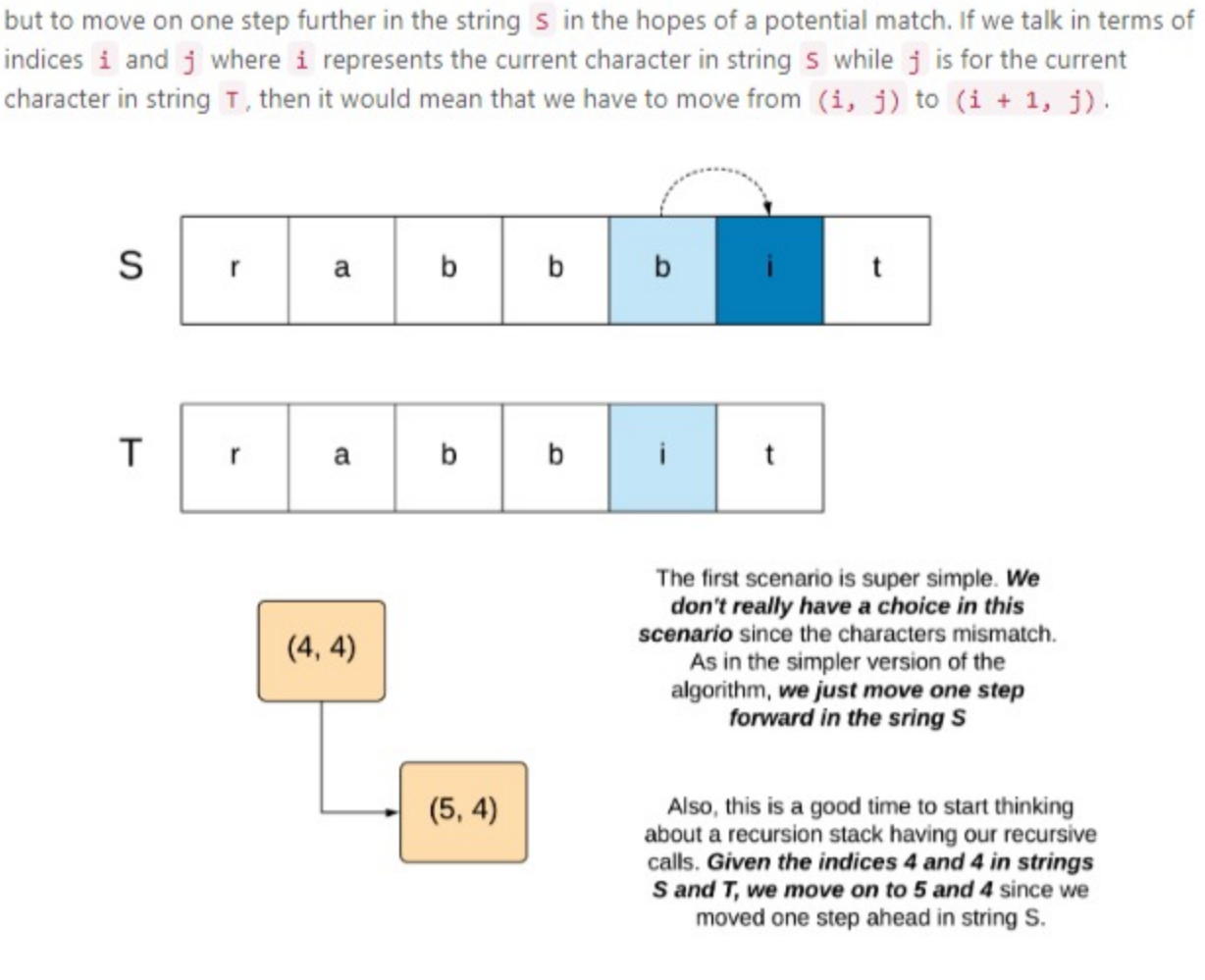




Once the last character in the string `T` gets a match in the string `S`, we will return `true` representing that we have found ***a*** subsequence in `S` that equals `T`. What we're gonna look at in our solution here is just an extension of this simple algorithm. The original problem asks us the `number of matched subsequences`. That basically means that our algorithm will have to `explore` all possible subsequences and count the ones that match the string `T`. That's a brute force way of solving this and one that won't pass the time limits since there are 2^N subsequences in a string of length N.

As mentioned before, the algorithm will be following the same ideology as before. We will be using a couple of indices to match one character at a time. Let's look at the two different scenarios again that we could possibly encounter and also look at what we have to do differently this time around.

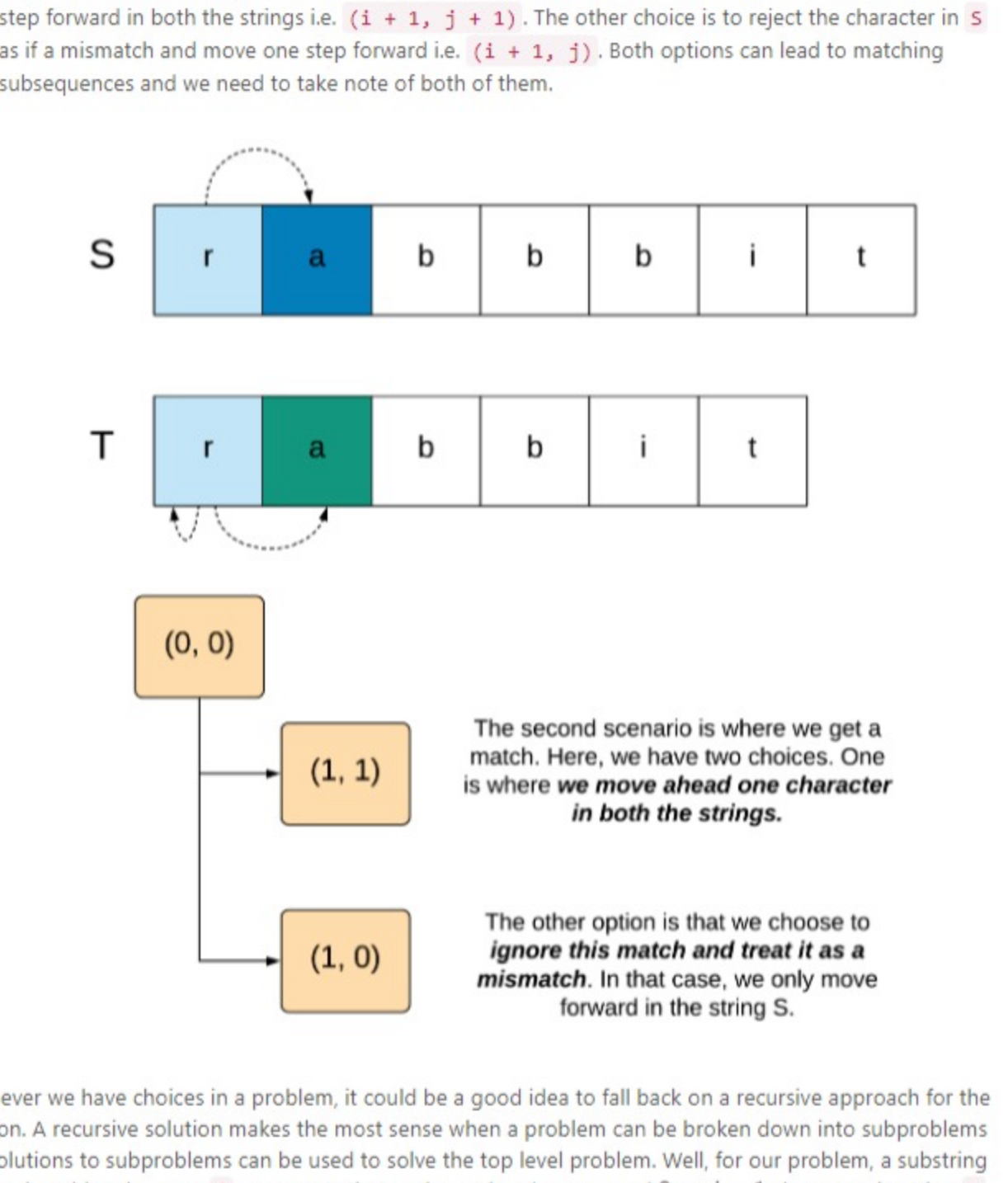
1. The first scenario is where the current characters do not match. In this case, we don't have any choice but to move on one step further in the string `S` in the hopes of a potential match. If we talk in terms of indices `i` and `j` where `i` represents the current character in string `S` while `j` is for the current character in string `T`, then it would mean that we have to move from `(i, j)` to `(i + 1, j)`.



2. The second scenario is a bit more interesting. Suppose the two characters match up. Now, in this case we can simply move one character each in both the strings i.e. `(i + 1, j + 1)` which is what we did for our simpler version of this problem. However, we need to find `all` possible subsequence matches, right? So, it's possible that we find the same character as `i`, at another index down the line, and from that point on we are able to find the remainder of the string `T` as well? Let's look at `s = rabbbit` and `t = rabit`.

   Here, when we match the character `b` at indices `i = 2` and `j = 2`, we can clearly see that the rest of string `T` i.e. `it` is present at the end of string `S`. This is one subsequence. However, if we `reject` the `b` at `i = 2` and instead move one step forward, we see another `b` at `i = 3` (and at `i = 4`) which we can use as the match for the corresponding `b` in string `T`. For our problem, we need to consider all three of them to get the answer.

   So, as mentioned before, one choice in this scenario, when we have a character match is to move one step forward in both the strings i.e. `(i + 1, j + 1)`. The other choice is to reject the character in `S` as if a mismatch and move one step forward i.e. `(i + 1, j)`. Both options can lead to matching subsequences and we need to take note of both of them.



Whenever we have choices in a problem, it could be a good idea to fall back on a recursive approach for the solution. A recursive solution makes the most sense when a problem can be broken down into subproblems and solutions to subproblems can be used to solve the top level problem. Well, for our problem, a substring is our subproblem because `i` represents that we have already processed $0 \cdots i-1$ characters in string `S`. Similarly, `j` represents that we have processed $0 \cdots j-1$ characters in string `T`.

Every recursive approach needs some variables that help define the state of the recursion. In our case, we have been talking about these two indices that will help us iterate over our strings one character at a time. Hence, `i` and `j` together will define the state of our recursion.

Since we've defined the state of our recursion function, we know what the inputs would be. Now, we need to think about what this function would return and how that would tie up with the input we are providing. The return value is not that hard to figure out really. Given two indices `i` and `j`, our function would return the number of distinct subsequences in the `substring` $s[i \cdots M]$ that equal the `substring` $t[j \cdots N]$ where M and N represent the lengths of the two string respectively.

It's time to bring some concreteness to the choices that we have been talking about in the previous few paragraphs. So, given the two indices `i` and `j`, we need to compare the characters in the corresponding strings and see if they match or not.

- If the characters match, then we have two possible branches where the recursion can go. `func(i + 1, j)` is where we `ignore` the current match in string `S` and move forward. `func(i + 1, j + 1)` is where we move forward in both the strings. Both of these contribute to the overall answer for this scenario as explained before. Thus, we have the following recursive relation:

```
func(i, j) = func(i + 1, j) + func(i + 1, j + 1)
```

- The second scenario is where the characters don't match. We don't really have any choice here but to move forward in the string `S` and hope to find the match somewhere later in the string. Hence:

```
func(i, j) = func(i + 1, j)
```

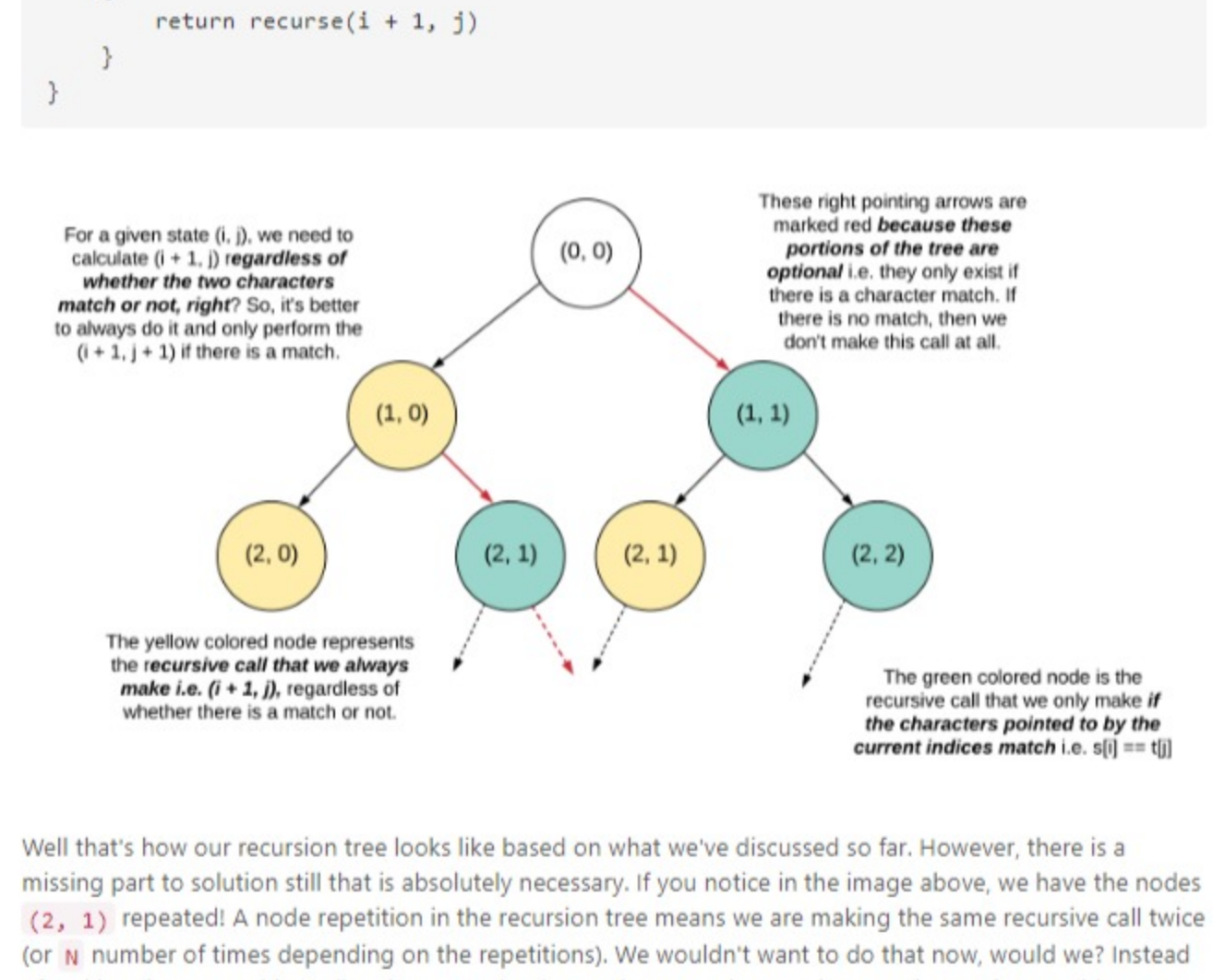
The final thing to discuss in our recursion based solution is the base case. There are two scenarios where we would break from our recursion and start to backtrack. We have two different strings of potentially different lengths. When one of them finishes, there's no point in going any further. So, `i == M` or `j == N` will form our base case. However, what we return in our base case is what will tie this whole thing together.

If we exhausted the string `S`, but there are still characters to be considered in string `T`, that means we ended up rejecting far too many characters and eventually ran out! Here, we return a 0 because now, there's no possibility of a match. However, if we exhausted the string `T`, then it means we found a subsequence in `S` that matches `T` and hence, we return a 1.

Another way of thinking about this scenario is that `func(i, N) = 1` because $t[N \cdots N]$ is an empty string and $s[i \cdots M]$ is non-empty. Every string has a subsequence which equals an empty string. Hence, we return a 1 in this base case.

```
func recurse(i, j)
{
    if (i == M or j == N)
    {
        return j == N ? 1 : 0
    }

    if (s[i] == t[j])
    {
        return recurse(i + 1, j) + recurse(i + 1, j + 1)
    }
    else
    {
        return recurse(i + 1, j)
    }
}
```



Well that's how our recursion tree looks like based on what we've discussed so far. However, there is a missing part to solution still that is absolutely necessary. If you notice in the image above, we have the nodes `(2, 1)` repeated! A node repetition in the recursion tree means we are making the same recursive call twice (or `N` number of times depending on the repetitions). We wouldn't want to do that now, would we? Instead of making these repetitive calls, why not cache the results somewhere and re-use them? That would prune our recursion tree's size so much! This is what we call `memoization` or simply, `caching`.

So, we use a dictionary with `(i, j)` as the key and the result of the function call `recurse(i, j)` as the value. Whenever we enter a recursive call, we first check for the base cases. If none of the base cases is hit yet, we check if the tuple `(i, j)` is present in the dictionary or not. If it is, we simply return the value. No need to repeat the calculations that we have already done before.

Algorithm

1. Define a function called `recurse` that takes in two integer values `i` and `j` where the first represents the current character to be processed in the string `S` and the second represents the current character in string `T`
2. Initialize a dictionary called `memo` that will cache the results for our different recursion calls.
3. We check the base case. If either of the strings is finished, we return a `0` or a `1` depending on whether we are able to process the entire string `T` or not. There's another base case that we need to consider here. If the `remaining` length of the string `S` is less than that of string `T`, then there's possibility of a match. If we detect this, then also we prune the recursion and return a `0`.
4. Next, we check if the current pair of indices exist in our dictionary or not. If they do, then we simply return the stored/cached value.
5. If not, we move on with the normal processing. We compare the characters `s[i]` and `t[j]`.
6. We store the result of `recurse(i + 1, j)` in a variable. As mentioned in the figure above, we need the result of this recursion irrespective of whether the characters match or not.
7. If the characters match, we add `recurse(i + 1, j + 1)` to the variable.
8. Finally, store this variable's value in the dictionary with the pair `(i, j)` as the key and return the value as the answer.

Java | Python | Copy

```
class Solution:
    def numDistinct(self, s: str, t: str) -> int:

        # Dictionary for memoization
        memo = {}

        def uniqueSubsequences(i, j):

            M, N = len(s), len(t)

            # Base case
            if i == M or j == N or M - i < N - j:
                return int(j == len(t))

            # Check if the result is already cached
            if (i, j) in memo:
                return memo[i,j]

            # Always make this recursive call
            ans = uniqueSubsequences(i + 1, j)

            # If the characters match, make the other
            # one and add the result to "ans"
            if s[i] == t[j]:
                ans += uniqueSubsequences(i + 1, j + 1)

            # Cache the answer and return
```

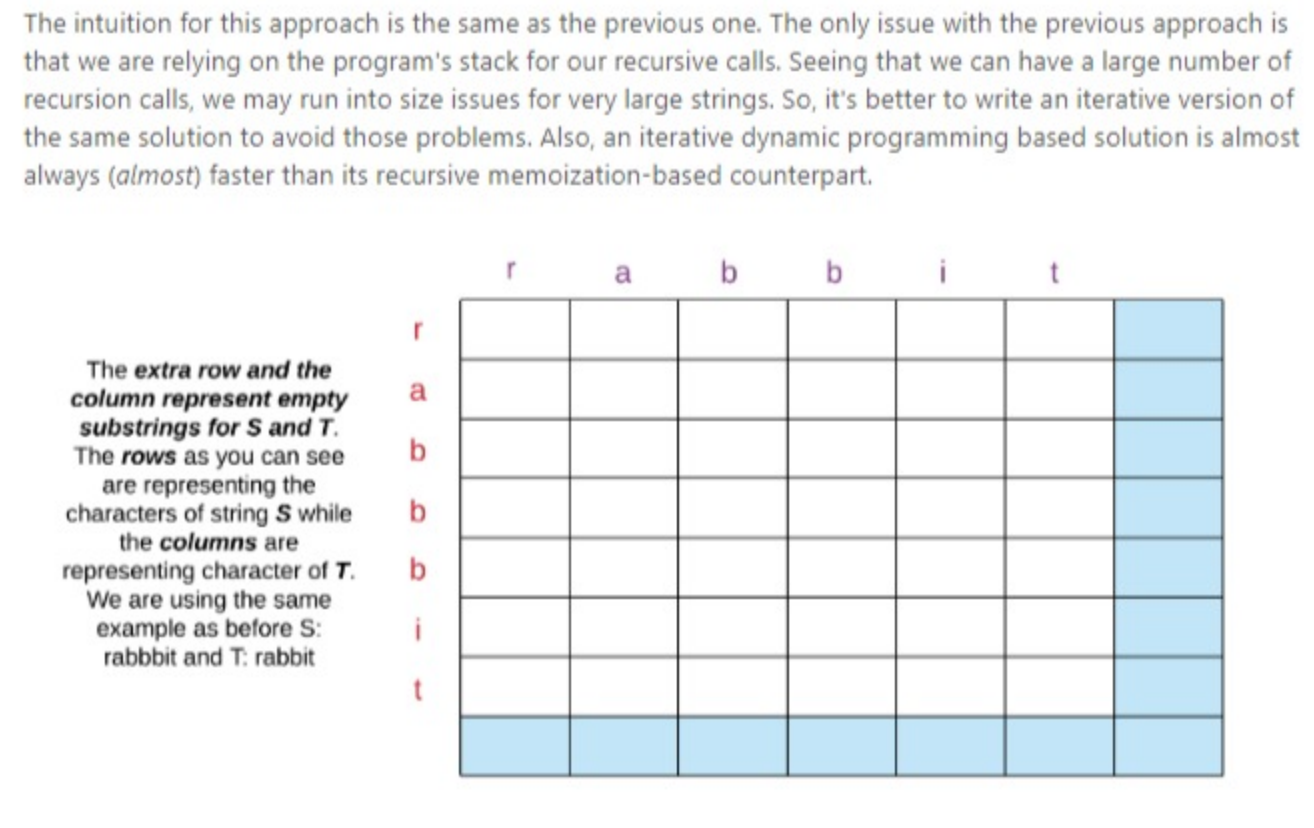
Complexity Analysis

- Time Complexity: The time complexity for a recursive solution is defined by two things: the number of recursive calls that we make and the time it takes to process a single call.
  - If you notice the solution closely, all we are doing in the function is to check the dictionary for a key, and then we make a couple of function calls. So the time it takes to process a single call is actually $O(1)$.
  - The number of unique recursive calls is defined by the two state variables that we have. Potentially, we can make $O(M \times N)$ calls where M and N represent the lengths of the two strings. Thus, the time complexity for this solution would be $O(M \times N)$.
- Space Complexity: The maximum space is utilized by the dictionary that we are using and the size of that dictionary would also be controlled by the total possible combinations of `i` and `j` which turns out to be $O(M \times N)$ as well. We also have the space utilized by the recursion stack which is $O(M)$ where M is the length of string `S`. This is because in one of our recursion calls, we don't progress at all in the string `T`. Hence, we would have a branch in the tree where only the index `i` progresses one step until it reaches the end of string `S`. The number of nodes in this branch would be equal to the length of string `S`.

Approach 2: Iterative Dynamic Programming

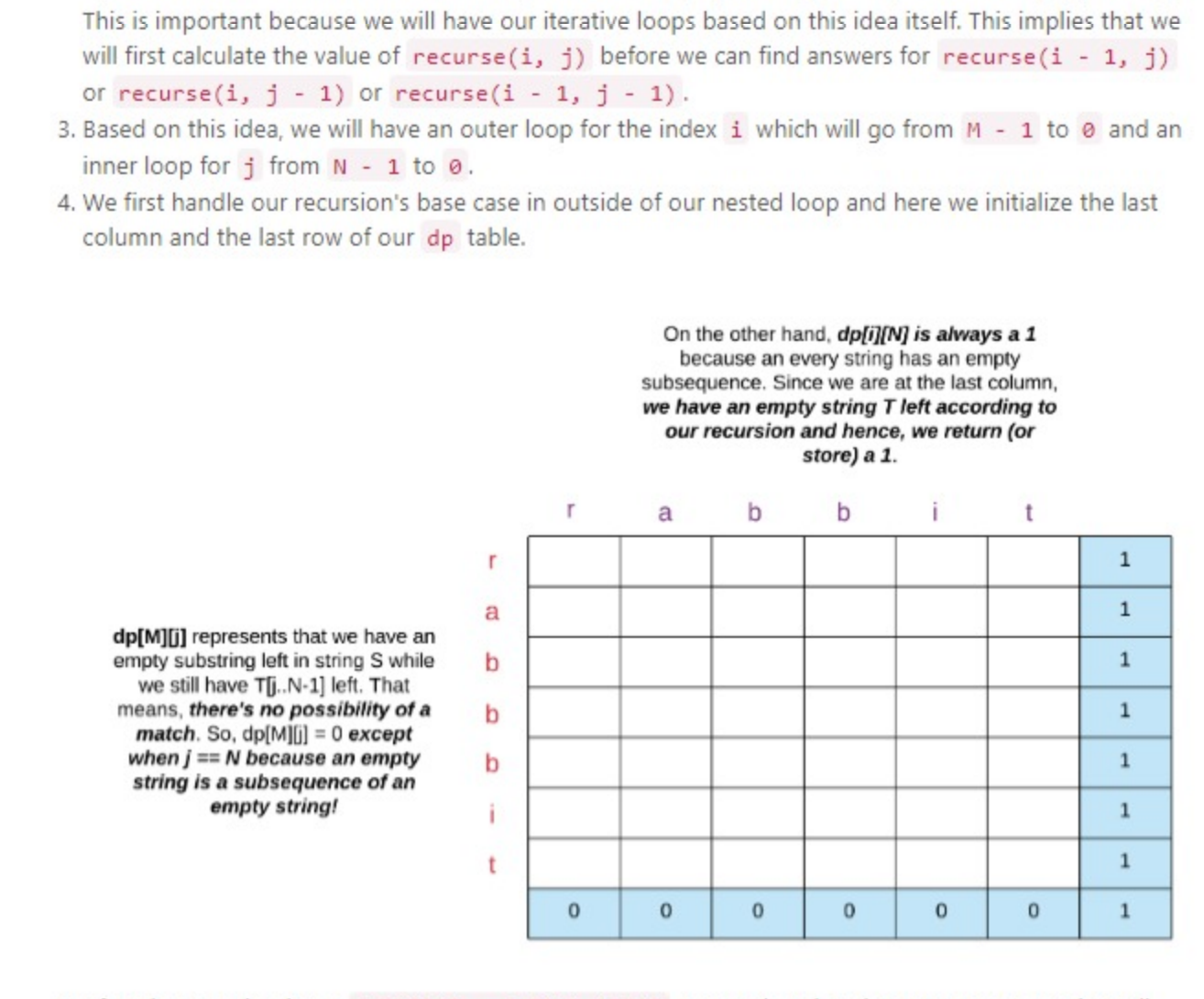
Intuition

The intuition for this approach is the same as the previous one. The only issue with the previous approach is that we are relying on the program's stack for our recursive calls. Seeing that we can have a large number of recursion calls, we may run into size issues for very large strings. So, it's better to write an iterative version of the same solution to avoid those problems. Also, an iterative dynamic programming based solution is almost always (*almost*) faster than its recursive memoization-based counterpart.

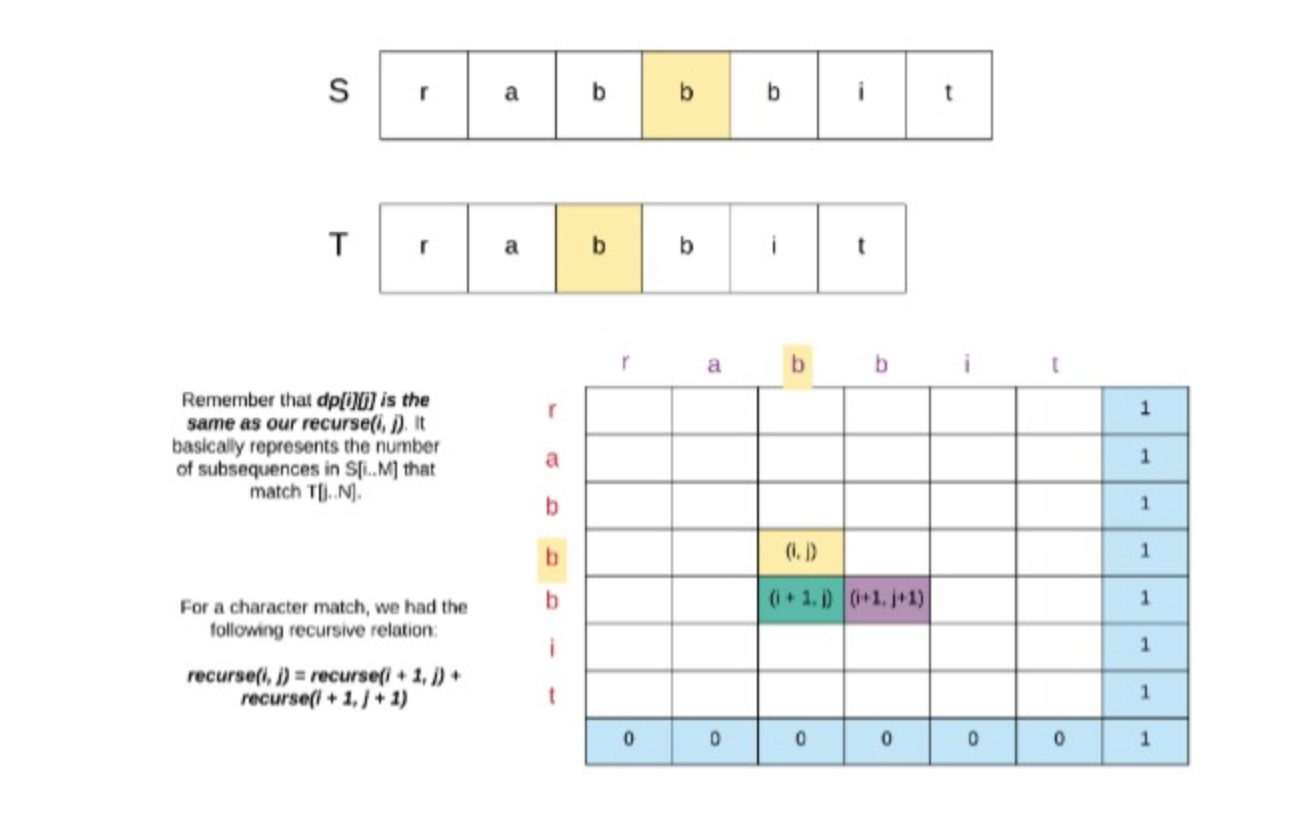


Algorithm

1. Initialize a 2D array `dp` of size $M \times N$ where M represents the length of string `S` while N represents the length of string `T`.
2. An important thing to remember here is what `recurse(i, j)` actually represents. It basically represents the number of distinct subsequences in string $s[i \cdots M]$ that equals the string $t[j \cdots N]$. This is important because we will have our iterative loops based on this idea itself. This implies that we will first calculate the value of `recurse(i, j)` before we can find answers for `recurse(i - 1, j)` or `recurse(i, j - 1)` or `recurse(i - 1, j - 1)`.
3. Based on this idea, we will have an outer loop for the index `i` which will go from `M - 1` to `0` and an inner loop for `j` from `N - 1` to `0`.
4. We first handle our recursion's base case in outside of our nested loop and here we initialize the last column and the last row of our `dp` table.



5. After that, we simply set `dp[i][j] = dp[i + 1][j]`. Remember that there was one recursive call that we need to make irrespective of whether there is a character match or not?
6. Then we check if the characters `s[i]` and `t[j]` match or not. If they do, then we add `dp[i + 1][j + 1]` to `dp[i][j]`. In the recursion based solution, we were caching this value in the dictionary. Here, the dictionary is replaced by the `dp` array.



7. Finally, after both the loops are finished, we return `dp[0][0]`.

Java | Python | Copy

```
class Solution:
    def numDistinct(self, s: str, t: str) -> int:

        M, N = len(s), len(t)

        # Dynamic Programming table
        dp = [[0 for i in range(N + 1)] for j in range(M + 1)]

        # Base case initialization
        for j in range(N + 1):
            dp[M][j] = 0

        # Base case initialization
        for i in range(M + 1):
            dp[i][N] = 1

        # Iterate over the strings in reverse so as to
        # satisfy the way we've modeled our recursive solution
        for i in range(M - 1, -1, -1):
            for j in range(N - 1, -1, -1):

                # Remember, we always need this result
                dp[i][j] = dp[i + 1][j]

                # If the characters match, we add the
                # result of the next recursion call (in this
                # case, the value of a cell in the dp table
```

Complexity Analysis

- Time Complexity: The time complexity is much more clear in this approach since we have two `for` loops with clearly defined executions. The outer loop runs for $M+1$ iterations while the inner loop runs for $N+1$ iterations. So, combined together we have a time complexity of $O(M \times N)$.
- Space Complexity: $O(M \times N)$ which is occupied by the 2D `dp` array that we create.

Approach 3: Space optimized Dynamic Programming

Intuition

The overall intuition for the algorithm remains the same as the initial recursive approach. However, it turns out that we can reduce the overall space complexity of our iterative solution. If you notice in the solution above, to calculate any value `dp[i][j]`, we only need elements from the next row i.e. `dp[i + 1]` isn't it? We need the values `dp[i + 1][j]` and `dp[i + 1][j + 1]`. Hence, for calculating the values in a particular row, we only ever need the values in the next row. So, this brings us to our final solution for this problem.

We simply need to have a one dimensional array of size N (length of string `T`) since that's the size of a single row in our previous 2D matrix. The first row that we create would be all zeros except the last value which will be 1. This will represent the last row of our 2D matrix. That's how we start processing the different cells and work our way up to `dp[0][0]`. Hence we call the iterative approaches bottom-up.

> If you think about it, the last value in this array will always remain 1 just like the last column in our 2D matrix will always be one since the last columns means `j == N` and hence the value 1. We simply update the values of this array in-place just as we would fill up our top-level rows one at a time. The only difference is that instead of using an entire matrix, we are re-using our 1D array thus saving on a ton of space. Let's look at the official algorithm for this.

Algorithm

1. Initialize an array called `dp` whose size equals that of the string `T`. We simply want to use a single row instead of a 2D matrix unlike the previous solution. Since a single row was of size N, that's what we will use to setup our 1D array.
2. As explained in the intuition section, the last cell in this array will always be 1. Notice the nested loops in the solution before:

```
for(i = M; i >= 0; i--)
{
    for(j = N; j >= 0; j--)
    {
        // Logic
    }
}
```

   For every character in the string `S`, we process the entire string `T` looking for potential matches. Thus, for every character in `S`, we start all the way from N in the current row and go to `0`. Since we have to do this now with just one array in hand, we can just use a simple variable that we set to `1` at the beginning of our inner loop. 3. Now, a very important thing to note is that we can easily use 2 arrays here. One would represent the current row that has to be updated while the other one would represent the next row which was updated in the previous iteration. That would simplify a lot of things. But why waste an additional array when we can get the job done with just a single array? This means, we need to do all the operations in-place.

   `dp[i][j] = dp[i + 1][j]` is in a sense, a useless operation because without the `i`, it's essentially the same cell in the 1D array, isn't it? The other operation `dp[i][j] = dp[i + 1][j + 1]` is the more interesting one. That's like saying `dp[j] = dp[j + 1]`. An easy enough operation, but since we are doing the operations in-place, this value would have been just updated, right?

   > So, we need to use an additional variable to keep track of the original value of the very next cell in the array. That's all we need to update the value of the current cell. So before updating the current cell we record its value in a temporary variable and after we update it's value, we set our "prev" variable to this original value so that it can be used to update the cell behind the current one i.e. dp[j-1].

Let's look at the solutions to make things crystal clear.

Java | Python | Copy

```
class Solution:
    def numDistinct(self, s: str, t: str) -> int:

        M, N = len(s), len(t)

        # Dynamic Programming table
        dp = [0 for j in range(N)]

        # Iterate over the strings in reverse so as to
        # satisfy the way we've modeled our recursive solution
        for i in range(M - 1, -1, -1):

            # At each step we start with the last value in
            # the row which is always 1. Notice how we are
            # starting the loop from N - 1 instead of N like
            # in the previous solution.
            prev = 1

            for j in range(N - 1, -1, -1):

                # Record the current value in this cell so that
                # we can use it to calculate the value of dp[j - 1]
                old_dpj = dp[j]

                # If the characters match, we add the
                # result of the next recursion call (in this
                # case, the value of a cell in the dp table
```

Complexity Analysis

- Time Complexity: $O(M \times N)$
- Space Complexity: $O(N)$ since we are using a single array which is the size of the string `T`. This is a major size reduction over the previous solution and this is a much more elegant solution than the initial recursive solution we saw earlier on.

Rate this article:

Previous Next

Comments: 4 Sort By

Type comment here... (Markdown is supported)

Preview Post

wisedev 9 May 6, 2020 8:32 AM

Very well written explanation, I really doubt if this kind of problem can really be solved in less than an hour, unless you have already seen/solved it before!!

7 Share Reply

SHOW 2 REPLIES

PuneetSaha 9 April 22, 2020 6:38 AM

Great explaination sachin!

3 Share Reply

SHOW 1 REPLY

little_late 51 June 25, 2020 1:07 AM

Great explanation! This question is very similar to the longest common subsequence, where only the common subsequence length was enough for the answer.

0 Share Reply

wlbberman 0 May 20, 2020 6:48 AM

You can also iterate through `s` beforehand to store the a mapping from character to indices where the character occurs. Afterwards, you can iterate through t while holding a pointer to what index you're at in `s`, incrementing the pointer to the index where the next character occurs with out checking all characters in between. I don't know if this decreases the worst case time complexity, but I believe it decreases the average case time complexity.

Read More

0 Share Reply