Subsequence Pattern Matching

We'll cover the following

Problem Statement

Given a string and a pattern, write a method to count the number of times the pattern appears in the string as a subsequence.

Example 1: Input: string: “baxmx”, pattern: “ax”
Output: 2
Explanation: {baxmx, baxmx}.

Example 2:

Input: string: “tomorrow”, pattern: “tor”
Output: 4
Explanation: Following are the four occurences: {tomorrow, tomorrow, tomorrow, tomorrow}.

Basic Solution

This problem follows the Longest Common Subsequence (https://www.educative.io/collection/page/5668639101419520/5633779737559040/5657535201673216) (LCS) pattern and is quite similar to the Longest Repeating Subsequence (https://www.educative.io/collection/page/5668639101419520/5633779737559040/5653294995210240); the difference is that we need to count the total occurrences of the subsequence.

A basic brute-force solution could be to try all the subsequences of the given string to count all that match the given pattern. We can match the pattern with the given string one character at a time, so we can do two things at any step:

1. If the pattern has a matching character with the string, we can recursively match for the remaining lengths of the pattern and the string.
2. At every step, we can always skip a character from the string to try to match the remaining string with the pattern. So we can start a recursive call by skipping one character from the string.

Our total count will be the sum of the counts returned by the above two options.

Here is the code:

Java JS Python3 C++

```python
def find_SPM_count(str, pat):
  return find_SPM_count_recursive(str, pat, 0, 0)


def find_SPM_count_recursive(str,  pat,  strIndex,  patIndex):

  # if we have reached the end of the pattern
  if patIndex == len(pat):
    return 1

  # if we have reached the end of the string but pattern has still some characters left
  if strIndex == len(str):
    return 0

  c1 = 0
  if str[strIndex] == pat[patIndex]:
    c1 = find_SPM_count_recursive(str, pat, strIndex + 1, patIndex + 1)

  c2 = find_SPM_count_recursive(str, pat, strIndex + 1, patIndex)

  return c1 + c2


def main():
  print(find_SPM_count("baxmx", "ax"))
  print(find_SPM_count("tomorrow", "tor"))


main()
```

The time complexity of the above algorithm is exponential $O(2^m)$, where 'm' is the length of the string, as our recursion stack will not be deeper than `m`. The space complexity is $O(m)$ which is used to store the recursion stack.

Top-down Dynamic Programming with Memoization

We can use an array to store the already solved subproblems.

The two changing values to our recursive function are the two indexes `strIndex` and `patIndex`. Therefore, we can store the results of all the subproblems in a two-dimensional array. (Another alternative could be to use a hash-table whose key would be a string (strIndex + "|" + patIndex)).

Here is the code:

Java JS Python3 C++

```python
def find_SPM_count(str, pat):
  dp = [[-1 for _ in range(len(pat))] for _ in range(len(str))]
```

```
  return find_SPM_count_recursive(dp, str, pat, 0, 0)


def find_SPM_count_recursive(dp, str, pat, strIndex, patIndex):

  # if we have reached the end of the pattern
  if patIndex == len(pat):
    return 1

  # if we have reached the end of the string but pattern has still some characters left
  if strIndex == len(str):
    return 0

  if dp[strIndex][patIndex] == -1:
    c1 = 0
    if str[strIndex] == pat[patIndex]:
      c1 = find_SPM_count_recursive(
        dp, str, pat, strIndex + 1, patIndex + 1)
    c2 = find_SPM_count_recursive(dp, str, pat, strIndex + 1, patIndex)
    dp[strIndex][patIndex] = c1 + c2

  return dp[strIndex][patIndex]


def main():
  print(find_SPM_count("baxmx", "ax"))
  print(find_SPM_count("tomorrow", "tor"))


main()
```

Bottom-up Dynamic Programming

Since we want to match all the subsequences of the given string, we can use a two-dimensional array to store our results. As mentioned above, we will be tracking separate indexes for the string and the pattern, so we will be doing two things for every value of `strIndex` and `patIndex`:

1. If the character at the `strIndex` (in the string) matches the character at `patIndex` (in the pattern), the count of the SPM would be equal to the count of SPM up to `strIndex-1` and `patIndex-1`.
2. At every step, we can always skip a character from the string to try matching the remaining string with the pattern; therefore, we can add the SPM count from the indexes `strIndex-1` and `patIndex`.

So our recursive formula would be:

```
if str[strIndex] == pat[patIndex] {
  dp[strIndex][patIndex] = dp[strIndex-1][patIndex-1]
}
dp[i1][i2] += dp[strIndex-1][patIndex]
```

Code #

Here is the code for our bottom-up dynamic programming approach:

Java JS Python3 C++

```python
def find_SPM_count(str, pat):
  strLen, patLen = len(str), len(pat)
  # every empty pattern has one match
  if patLen == 0:
    return 1

  if strLen == 0 or patLen > strLen:
    return 0

  # dp[strIndex][patIndex] will be storing the count of SPM up to str[0..strIndex-1][0..pa
  dp = [[0 for _ in range(patLen+1)] for _ in range(strLen+1)]

  # for the empty pattern, we have one matching
  for i in range(strLen+1):
    dp[i][0] = 1

  for strIndex in range(1, strLen+1):
    for patIndex in range(1, patLen+1):
      if str[strIndex - 1] == pat[patIndex - 1]:
        dp[strIndex][patIndex] = dp[strIndex - 1][patIndex - 1]
      dp[strIndex][patIndex] += dp[strIndex - 1][patIndex]

  return dp[strLen][patLen]


def main():
  print(find_SPM_count("baxmx", "ax"))
  print(find_SPM_count("tomorrow", "tor"))


main()
```

The time and space complexity of the above algorithm is $O(m * n)$, where 'm' and 'n' are the lengths of the string and the pattern respectively.

Back

Longest Repeating Subsequence

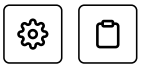
Next

Longest Bitonic Subsequence

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/subsequence-pattern-matching__pattern-5-longest-common-substring__grokking-dynamic-programming-patterns-for-coding-interviews)



Could not connect to the reCAPTCHA service. Please check your internet connection and reload to get a reCAPTCHA challenge.