**Subsequence Pattern Matching**

**Problem Statement**[#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/JY7yZ2VyJnP#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}.

### Basic Solution [#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/JY7yZ2VyJnP#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.

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:

class SPM {

  public int findSPMCount(String str, String pat) {

    return findSPMCountRecursive(str, pat, 0, 0);

  }

  private int findSPMCountRecursive(String str, String pat, int strIndex, int patIndex) {

    // if we have reached the end of the pattern

    if(patIndex == pat.length())

      return 1;

    // if we have reached the end of the string but pattern has still some characters left

    if(strIndex == str.length())

      return 0;

    int c1 = 0;

    if(str.charAt(strIndex) == pat.charAt(patIndex))

      c1 = findSPMCountRecursive(str, pat, strIndex+1, patIndex+1);

    int c2 = findSPMCountRecursive(str, pat, strIndex+1, patIndex);

    return c1 + c2;

  }

  public static void main(String[] args) {

    SPM spm = new SPM();

    System.out.println(spm.findSPMCount("baxmx", "ax"));

    System.out.println(spm.findSPMCount("tomorrow", "tor"));

  }

}

The time complexity of the above algorithm is exponential O(2^{m})*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)*O*(*m*) which is used to store the recursion stack.

### Top-down Dynamic Programming with Memoization [#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/JY7yZ2VyJnP#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:

class SPM {

  public int findSPMCount(String str, String pat) {

    Integer[][] dp = new Integer[str.length()][pat.length()];

    return findSPMCountRecursive(dp, str, pat, 0, 0);

  }

  private int findSPMCountRecursive(Integer[][] dp, String str, String pat, int strIndex, int patIndex) {

    // if we have reached the end of the pattern

    if(patIndex == pat.length())

      return 1;

    // if we have reached the end of the string but pattern has still some characters left

    if(strIndex == str.length())

      return 0;

    if(dp[strIndex][patIndex] == null) {

      int c1 = 0;

      if(str.charAt(strIndex) == pat.charAt(patIndex))

        c1 = findSPMCountRecursive(dp, str, pat, strIndex+1, patIndex+1);

      int c2 = findSPMCountRecursive(dp, str, pat, strIndex+1, patIndex);

      dp[strIndex][patIndex] = c1 + c2;

    }

    return dp[strIndex][patIndex];

  }

  public static void main(String[] args) {

    SPM spm = new SPM();

    System.out.println(spm.findSPMCount("baxmx", "ax"));

    System.out.println(spm.findSPMCount("tomorrow", "tor"));

  }

}

### Bottom-up Dynamic Programming [#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/JY7yZ2VyJnP#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:

1

2

3

4

if str[strIndex] == pat[patIndex] {

  dp[strIndex][patIndex] = dp[strIndex-1][patIndex-1]

}

dp[i1][i2] += dp[strIndex-1][patIndex]





#### Code [**#**](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/JY7yZ2VyJnP#code)

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

class SPM {

  public int findSPMCount(String str, String pat) {

    // every empty pattern has one match

    if(pat.length() == 0)

      return 1;

    if(str.length() == 0 || pat.length() > str.length())

      return 0;

    // dp[strIndex][patIndex] will be storing the count of SPM up to str[0..strIndex-1][0..patIndex-1]

    int[][] dp = new int[str.length()+1][pat.length()+1];

    // for the empty pattern, we have one matching

    for(int i=0; i<=str.length(); i++)

      dp[i][0] = 1;

    for(int strIndex=1; strIndex<=str.length(); strIndex++) {

      for(int patIndex=1; patIndex<=pat.length(); patIndex++) {

        if(str.charAt(strIndex-1) == pat.charAt(patIndex-1))

          dp[strIndex][patIndex] = dp[strIndex-1][patIndex-1];

        dp[strIndex][patIndex] += dp[strIndex-1][patIndex];

      }

    }

    return dp[str.length()][pat.length()];

  }

  public static void main(String[] args) {

    SPM spm = new SPM();

    System.out.println(spm.findSPMCount("baxmx", "ax"));

    System.out.println(spm.findSPMCount("tomorrow", "tor"));

  }

}

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.