CS/COE 0445 Fall 2020 Assignment 3
Additional Help for the `regMatch()` method

To get started with this method, I would recommend that in the non-recursive version of the method (i.e. the one that meets the public specification) you create an array of MyStringBuilder equal in length to the argument array of String. Note that if your match succeeds this ultimately will be the return value from this method. As you proceed forward through the matches you will assign MyStringBuilder objects to the locations of this array, and as you backtrack you will remove them.

The key to getting this method working is understanding the different possible states that exist in the recursive calls and the transitions from one state to another. Once you figure these states and transitions out, the method is still not trivial but it should be more manageable.

Below is an informal description of the states that you will have to deal with. It does not show you what to do in every situation, but it does give you an idea of how the program progresses in the forward direction. Given these "forward" transitions you must figure out how the backtracking will work. Some of the backtracking situations are shown later in this handout. Consider a single call to the recursive method, where you are considering:

The current character in the MyStringBuilder2
The current pattern, i, within the array of patterns, to match

1) **No characters in pattern i have matched, with i = 0.** This is the initial state and this state will continue until at least one character in pattern 0 is matched. If you are in this state consider the current character in the MyStringBuilder2:
    a. There are no characters left in the MyStringBuilder2. In this case return FALSE.
    b. If the current character does NOT match pattern 0, stay in this state and move down to the next character in the MyStringBuilder2
    c. If the current character DOES match pattern 0, move to State 2) and move down to the next character in the MyStringBuilder2

2) **At least one character in pattern i is matched**. Since the algorithm is greedy, at this point you will first try to continue matching pattern i. However, if the current character does not match pattern i you may also try to move to pattern i+1. In other words, consider the current character in the MyStringBuilder2:
    a. If the character MATCHES pattern i, stay in this state and move down to the next character in the MyStringBuilder2.
    b. If the character DOES NOT match pattern i and i is the LAST pattern, return TRUE (success)
    c. If the character DOES NOT match pattern i and i is NOT the LAST pattern, move to pattern i+1 and State 3) but stay at the SAME character in the MyStringBuilder2

3) **No characters in pattern i match where i != 0**. In this state we are trying to match the first character for a pattern other than pattern 0. Consider the current character in the MyStringBuilder2:
    a. If the character MATCHES pattern i, move to State 2) and move down to the next character in the MyStringBuilder2.
    b. If the character DOES NOT match pattern i return FALSE – the algorithm must backtrack. This is because we cannot have any gaps in the matching.

To illustrate how the process should work, consider the example below, which was also shown in the Assignment 3 assignment sheet. You may want to draw some pictures as you follow. Only part of the

trace is shown, but see how the matches are built (forward calls) and un-built (backtracking) as the calls move forward and backward. Under each call is a list of the patterns that have matched so far.

```
String patA = "ABC", patB = "123", patC = "XYZ";
String [] patterns = { patA, patB, patC };
MyStringBuilder2 B = new MyStringBuilder2("**BBB22AAYYCC3ZZZ**");
MyStringBuilder2 [] ans = B.regMatch(patterns);
```

Call 1: In State 1), trying to match pattern 0 ("ABC") with character '*'. It does not match, staying in 1)

```
Match 0:
```

Call 2: In State 1), trying to match pattern 0 ("ABC") with character '*'. It does not match, staying in 1)

```
Match 0:
```

Call 3: In State 1), trying to match pattern 0 ("ABC") with character 'B'. It matches, moving to State 2)

```
Match 0: 'B'
```

Call 4: In State 2), trying to match pattern 0 ("ABC") with character 'B'. It matches, staying in 2)

```
Match 0: 'BB'
```

Call 5: In State 2), trying to match pattern 0 ("ABC") with character 'B'. It matches, staying in 2)

```
Match 0: 'BBB'
```

Call 6: In State 2), trying to match pattern 0 ("ABC") with character '2'. It does not match and 0 is not the last pattern. Stay at the same character but move to pattern 1 and State 3)

```
Match 0: 'BBB'     Match 1:
```

Call 7: In State 3), trying to match pattern 1 ("123") with character '2'. It matches, moving to State 2)

```
Match 0: 'BBB'     Match 1: '2'
```

Call 8: In State 2), trying to match pattern 1 ("123") with character '2'. It matches, staying in 2)

```
Match 0: 'BBB'     Match 1: '22'
```

Call 9: In State 2), trying to match pattern 1 ("123") with character 'A'. It does not match and 1 is not the last pattern. Stay at the same character but move to pattern 2 and State 3)

```
Match 0: 'BBB'     Match 1: '22'   Match 2:
```

Call 10: In State 3), trying to match pattern 2 ("XYZ") with character 'A'. It does not match so return FALSE. We must backtrack to Call 9.

```
Match 0: 'BBB'     Match 1: '22'   Match 2: FAIL
```

Back in Call 9: We tried to match the next pattern and failed, so this call also fails. We must backtrack to Call 8.

```
Match 0: 'BBB'     Match 1: '22'   Match 2: FAIL
```

Back in Call 8: We are in State 2) and staying in 2) did not succeed. Try moving to State 3) with pattern 2 and the same character ('2'). *This transition is key to getting this method to work.* The idea is that we may need to recurse TWICE from the same state – once with the same pattern and once trying the next pattern. Both of these must be attempted in order to check all possible cases. In this case we remove the last '2' from Match 1 and are instead trying to match it with pattern 2.

```
Match 0: 'BBB'     Match 1: '2'    Match 2:
```

Call 11: In State 3), trying to match pattern 2 ("XYZ") with character '2'. It does not match so return FALSE. We must backtrack to Call 8.

```
Match 0: 'BBB'     Match 1: '2'    Match 2: FAIL
```

Back in Call 8: We are in State 2) again and failed both with continuing pattern 1 and with moving to pattern 2. Return FALSE and backtrack to Call 7.

```
Match 0: 'BBB'     Match 1: '2'    Match 2: FAIL
```

Back in Call 7: Backtracking leaves us with no characters in pattern 1. Return FALSE and backtrack to Call 6.

```
Match 0: 'BBB'     Match 1: FAIL
```

Back in Call 6: We are in State 2) and failed both with continuing pattern 0 and with moving to pattern 1. Return FALSE and backtrack to Call 5.

```
Match 0: 'BBB'     Match 1: FAIL
```

Back in Call 5: We are in State 2) and staying in 2) did not succeed. Try moving to State 3) with pattern 1 and the same character ('B').

```
Match 0: 'BB'      Match 1:
```

Call 12: In State 3), trying to match pattern 1 ("123") with character 'B'. It does not match so return FALSE. We must backtrack to Call 5.

```
Match 0: 'BB'      Match 1: FAIL
```

Back in Call 5: We are in State 2) and failed both with continuing pattern 0 and with moving to pattern 1. Return FALSE and backtrack to Call 4.

```
Match 0: 'BB'      Match 1: FAIL
```

Back in Call 4: We are in State 2) and staying in 2) did not succeed. Try moving to State 3) with pattern 1 and the same character ('B').

```
Match 0: 'B'       Match 1:
```

Call 13: In State 3), trying to match pattern 1 ("123") with character 'B'. It does not match so return FALSE. We must backtrack to Call 4.

```
Match 0: 'B'       Match 1: FAIL
```

Back in Call 4: We are in State 2) and failed both with continuing pattern 0 and with moving to pattern 1. Return FALSE and backtrack to Call 3.

```
Match 0: 'B'       Match 1: FAIL
```

Back in Call 3: Match 0 is now empty, so stay in State 1) and move to the next character

```
Match 0:
```

This process will clearly continue for some time until the overall match succeeds with "CC", "3", "ZZZ". Not all cases are demonstrated in this handout but I hope that it will give you a good idea of what needs to be considered and how the recursion and backtracking required for this method will proceed.