# CS 0445 Fall 2021 Assignment 3

**Online: Monday, October 18, 2021**
**Due:** All source files (both those provided to you and those that you wrote) plus a completed Assignment Information Sheet zipped into a single .zip file and submitted to the proper directory in the submission site by **11:59PM on Wednesday, November 3, 2021.**
**Late Due Date:** 11:59PM on Friday, November 5, 2021

**Purpose:** Now that we have looked at some interesting recursive programs, we can try some recursive programming for ourselves. In this assignment you will re-implement the MyStringBuilder class, with (mostly) the identical methods and data as in Assignment 2. However, now you must use recursive methods for all of your operations where iteration was used previously. This will give you good practice in programming recursively and will help you to better understand designing and implementing recursive methods. To give you some experience with some more challenging recursive algorithms, we will also add two new methods to the requirements.

**Goal and Details:** You will implement class MyStringBuilder2, which has the same functionality as MyStringBuilder as specified in Assignment 2. However you will add two additional methods to the class, as described below. Furthermore, in the MyStringBuilder2 class you **cannot have any loops of any kind in your code**. Any method that previously utilized loops **must now be implemented using recursion**. Further (as before) you may not use any of the StringBuilder, StringBuffer or String versions of the methods that you are trying to implement (since they are not recursive you could not use them anyway), and you must operate directly on your MyStringBuilder2 object within your methods. **The same rules about unnecessary copying and traversing apply for this class – do not do more work than you need to do.** See the MyStringBuilder2 class skeleton for more details on the functionality, especially for details on the two additional required methods. Also see the code at the end of this document for some help with implementing your methods recursively.

Test your MyStringBuilder2 class with two programs: Assig3.java and Assig3B.java. The output must be identical to that shown in the files A3Out.txt and A3BOut.txt. Note that Assig3.java is identical to Assig2.java, with the exception of the class name for the variables and objects. In Assig2.java the class is MyStringBuilder and in Assig3.java the class is MyStringBuilder2. Assig3B.java tests two additional methods that you must write for the MyStringBuilder2 class, as described below.

**Additional Methods:**
In order to get some practice with backtracking algorithms, you will add 2 methods that utilize backtracking to the requirements:

```
public int lastIndexOf(String str)
```

This method is similar in functionality to the indexOf() method that you implemented previously. However, now you must return the match closest to the end of the MyStringBuilder2 rather than the match closest to the beginning. In order to avoid finding many unnecessary matches it is better to "start" this search from the end of the MyStringBuilder2 rather than the beginning. One way of doing this is to proceed "backward" through the list (since it is circular and doubly-linked). Another approach that will incorporate the same basic strategy of your indexOf() method is a simple backtracking algorithm.. The idea is that you will recurse down the list without actually trying to match the string until you get to the "end". The matching is only done AFTER the recursive call and only if the previous call did not match. Think about this and trace it on paper to see how you would set this up. If you think about this carefully, you will see that the indexOf() and lastIndexOf() methods can be implemented recursively with very similar code. However you decide to implement lastIndexOf(), it must be recursively done.

`public` **`MyStringBuilder2 [] regMatch(String [] pats)`**

This method is a VERY simplified version of a regular expression match that is built into the Java String class (and is also in many other programming languages). The idea is this: `pats` is an array of strings, each of which represents a **set** of characters. To "match" a single pattern you simply must find one or more of the characters within it – not taking into account any order. The first found, longest possible answer that satisfies the match will always be returned. For example, consider the following:

```
String patA = "ABC";
String [] patterns = { patA };
MyStringBuilder2 B = new MyStringBuilder2("****BBABA999ABCABC");
MyStringBuilder2 [] ans = B.regMatch(patterns);
```

In this case `ans` would contain the single string `"BBABA"`. Note that it does not matter that 'B' was found first or that 'C' was not found at all. All of the characters in the match came from the set containing 'A', 'B' and 'C'. Note also that it does not matter that a longer match could be found later in the string ("ABCABC"). The first match that satisfies the pattern will always be returned.

If additional pattern strings are added to the array of String argument, then all of the individual patterns must be matched in order with no gaps for the overall match to succeed. If the match succeeds the resulting array of MyStringBuilder2 will have the same length as the argument array of String and each `ans[i]` in the result will correspond to the part of the match attributed to `pats[i]` in the argument. For example, consider the following:

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

The call above is trying to match all three of patA, patB and patC in that order in a contiguous substrtring of the MyStringBuilder2. In this case `ans` would contain the strings "CC", "3", "ZZZ", since "CC" matches patA, "3" matches patB and "ZZZ" matches patC, and the characters together are one contiguous substring within the MyStringBuilder2. Note that patA and patB match with "BBB" and "22" but patC cannot match afterward so that overall match fails. Similarly, "AA" matches patA but patB cannot match afterward so that overall match fails. It is backtracking that enables this overall solution to succeed, since previous matches of patA and patB must be given up and started again later in the string.

It is possible that a pattern string is a superset of another pattern string. In this instance it is possible for a single character in the MyStringBuilder2 to match either of the pattern strings. For these situations the algorithm should be "greedy" with the matches: match as many characters in `pats[i]` before moving on to `pats[i+1]`. However, the overall success of the match takes precedence, so one or more characters that initially matched `pats[i]` may have to be given back in order for `pats[i+1]` to also match. For example, consider the following:

```
String patA = "ABC123XYZ", patB = "123", patC = "XYZ";
String [] patterns = { patA, patB, patC };
MyStringBuilder B = new MyStringBuilder("BBB222YYYCCC");
MyStringBuilder [] ans = B.regMatch(patterns);
```

Note that in this case patA could match the entire MyStringBuilder2, since all of the characters in the MyStringBuilder2 fall within patA. However, if that were so then patB and patC could not match anything and the overall match would fail. In order for the overall match to succeed the algorithm must backtrack – remove some of the previously matched characters so that they can be used for the next part of the match. In the case above `ans` would contain the strings "BBB22", "2", and "YYY". Note that all of the 'Y' characters are "given back" from the initial patA match but only one '2' is "given back". This is because the algorithm only backtracks as far as necessary before it proceeds forward again. In order to match patB only 1 '2' character is needed and all of the 'Y' characters are then available for patC to match.

The coding to implement this method will take some thought and you will definitely want to try some cases with a pencil and paper before implementing it. I have several examples in the sample output that

should help you to better understand what is needed and how to start. I recommend doing this method last since it is likely the most challenging method in the assignment.

**Note:** During the backtracking phase of this method you may want to remove a character from the end of a MyStringBuilder2 that represents the matched string. The idea is that when you go forward you add a character to the end and when you backtrack you remove that character, making the current answer behave like a "Stack" ADT. Luckily, since your MyStringBuilder2 uses a circular, doubly-linked list, the last character can be removed very efficiently, as was demonstrated in Assignment 2.

**Hints and Help:** Recursive algorithms take some getting used to, and if you are not experienced in using them it may be difficult to even know how to begin. Therefore, I have provided two methods for you in this assignment sheet, with some documentation**.** Recitation Exercise 5 will also be a lot of help with this assignment. I have posted the solution to Recitation Exercise 5 on the course Canvas site. Use these examples to see how you might want to approach your other methods. Keep in mind that some of the methods are more complicated than the others. For all of the methods, I recommend tracing execution on paper as you are developing the code. I have provided a skeleton of the MyStringBuilder2.java file for you. Use this as a starting point for your implementation.

```
// Constructor to make a new MyStringBuilder2 from a String.  The constructor
// itself is NOT recursive - however, it calls a recursive method to do the
// actual set up work.  This should be your general approach for all of the
// methods, since the recursive methods typically need extra parameters that
// are not given in the specification.
public MyStringBuilder2(String s)
{
	if (s != null && s.length() > 0)
		makeBuilder(s, 0);
	else  // no String so initialize empty MyStringBuilder2
	{
		firstC = null;
		length = 0;
	}
}

// Recursive method to set up a new MyStringBuilder2 from a String
private void makeBuilder(String s, int pos)
{
	// Recursive case - we have not finished going through the String
	if (pos < s.length()-1)
	{
		// Note how this is done - we make the recursive call FIRST, then
		// add the node before it.  In this way EVERY node we add is
		// the front node, and it enables us to avoid having to make a
		// special test for the front node.  However, many of your
		// methods will proceed in the normal front to back way.
		makeBuilder(s, pos+1);
		CNode temp = new CNode(s.charAt(pos));
		CNode back = firstC.prev;
		temp.prev = back;
		back.next = temp;
		temp.next = firstC;
		firstC.prev = temp;
		firstC = temp;
		length++;
	}
	else if (pos == s.length()-1) // Special case for last char in String
	{                             // This is a base case and initializes
```

```
                                        // firstC in a circular way
            firstC = new CNode(s.charAt(pos));
            firstC.next = firstC;
            firstC.prev = firstC;
            length = 1;
      }
      else  // This case should never be reached, due to the way the
            // constructor is set up.  However, I included it as a
      {     // safeguard (in case some other method calls this one)
            length = 0;
            firstC = null;
      }
}
```

```
// Again note that the specified method is not actually recursive - rather it
// calls a recursive method to do the work.  Note that in this case we also
// create a char array before the recursive call, then pass it as a
// parameter, then construct and return a new string from the char array.
// Carefully think about the parameters you will be passing to your recursive
// methods.  Through them you must be able to move through the list and
// reduce the "problem size" with each call.
public String toString()
{
      char [] c = new char[length];
      getString(c, 0, firstC);
      return (new String(c));
}

// Here we need the char array to store the characters, the pos value to
// indicate the current index in the array and the curr node to access
// the data in the actual MyStringBuilder2.  Note that these rec methods
// are private - the user of the class should not be able to call them.
private void getString(char [] c, int pos, CNode curr)
{
      if (pos < length) // Not at end of the list
      {
            c[pos] = curr.data;  // put next char into array
            getString(c, pos+1, curr.next);  // recurse to next node and
      }                                      // next pos in array
}
```

Just as with Assignment 2, be careful not to make any unnecessary copies of the data and not to do any unnecessary traversals. For example, a recursive getNodeAt() method may be useful but you should not use it to find both the beginning and the end of a string that you want to delete, since that would require two traversals from the beginning of the MyStringBuilder2.

**Extra Hint:** Before implementing these methods look again at your iterative implementations from Assignment 2. Some of the iterative methods had two parts (ex: replace() – first delete and then insert) and at least one had nested loops (indexOf()). You may be more comfortable breaking these up into several recursive sub-algorithms. For example, in the replace() method you may have a recursive method to delete the previous string and a separate recursive method to insert the new string. However, if you do this be careful **not** to start the insert at the beginning of the MyStringBuilder2; rather it should start from the current position in the MyStringBuilder2 (to avoid an extra traversal). Definitely trace this on paper to see how to best set up the recursive method(s). For another example, in the iterative indexOf() method there are nested loops – the outer loop moves the beginning of the search down the MyStringBuilder and the inner loop tries to match the characters with the target. These could also be written as two recursive sub-algorithms in your MyStringBuilder2 class if you find that approach to be simpler than writing a single recursive algorithm.

After you have completed your recursive MyStringBuilder2 class, you will test it with main programs Assig3.java and Assig3B.java. As stated previously, Assig3.java is the same as Assig2.java with the exception of the "2" added to the MyStringBuilder name. Assig3B.java tests the two additional methods, lastIndexOf() and regMatch(). Your output should be identical to the output shown in A3Out.txt and A3BOut.txt.

Make sure you **submit ALL** of the following in your **single .zip file** to get full credit:

1) All of the files that you wrote, well-formatted and reasonably commented:
   a) MyStringBuilder2.java (recursive version)
   b) Any other Java files that you may have also written
2) The test driver programs, unchanged from how you downloaded them:
   a) Assig3.java
   b) Assig3B.java
3) Your completed Assignment Information Sheet. Be sure to help the TA with grading by clearly indicating here what you did and did not get working for the project.

As with Assignments 1 and 2, the idea from your submission is that your TA can compile and run your programs WITHOUT ANY additional files, so be sure to test them thoroughly before submitting them.

Some of the MyStringBuilder2 methods are **VERY challenging recursively.** If you cannot get the programs working as given, clearly indicate any changes you made and clearly indicate why (ex: "I could not get the indexOf() method to work, so I eliminated code that tested it") on your Assignment Information Sheet [one test the TA may do is to compile and run your MyStringBuilder2.java file with different copies of the test files – so if you modified them in any way in order to get your program to work, it is essential that you state as much in your Assignment Information Sheet].

**Extra Credit:** This assignment is already very challenging and you may not have time to try any extra credit. However, if you want to try some extra credit, there are many ways by which you can improve / enhance the regMatch() method. For example, real regular expression functionality allows options with how many characters to match (ex: 0 or more, 1 or more, etc). Real regular expression handling also allows shorthand notation for the patterns (ex: [A-Z]+ means one or more capital letters; [0-9]* means 0 or more digits). Handling this notation can also get you some extra credit.