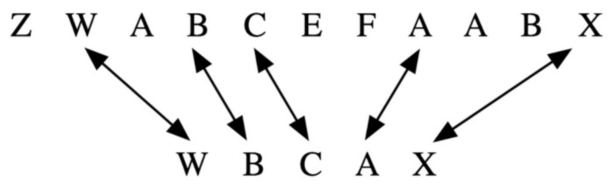
COM498 Algorithms and Data Structures

Challenge 2.3 - Longest Common Subsequence

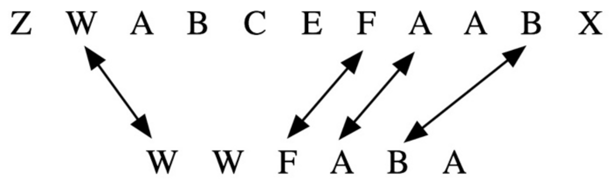
This exercise makes use of the **`ArrayBag`** implementation from Practical 2: please be sure to have a completed, working implementation of the **`ArrayBag`** class (**`ArrayBag.java`**) prior to attempting this exercise.

In the application created in this exercise, two strings of characters will be taken as input and the longest subsequence of characters common to both strings will be determined. We want to find the longest sequence of letters that is common between two strings. For one string to be a subsequence of the other, all letters in the first string must match up uniquely with a letter in the second string. The matches have to be in the same order, but they do not need to be consecutive. For example, "WBCAX" is a subsequence of "ZWABCEFAABX" as can be seen from the matching:



On the other hand, "WBAFX" is not a subsequence of "ZWABCEFAABX" since there is no way to match up the letters in the correct order.

As another example, "WWFABA" is not a subsequence of "ZWABCEFAABX". There are a couple issues that we run into with this example. Firstly, we can only match up one character with one character, so the subsequence check fails due to an excess of "W" characters. Secondly, while "ABA" is a subsequence of "ZWABCEFAABX", "FABA" is not:



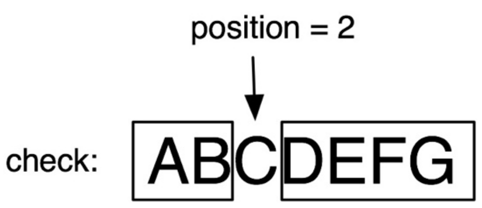
In order to find the longest common subsequence between two strings, we must first create an algorithm to determine if one string is actually a subsequence of another string. The following source code for the **`isSubsequence()`** performs this function on two strings that are input as parameters (i.e. **`strCheck`**, **`strAgainst`**). If the size of the first string is less than or equal to the size of the second string, then there is a possibility that the first string is a subsequence of the second string. The algorithm then iterates through the two strings, checking if the characters of the first string match the characters of the second string in order to determine if a subsequence occurs. When a subsequence occurs, the number of characters found that match will equal the length of the first string (note: an empty string is accepted as a subsequence of any other string).

```java
public static boolean isSubsequence(String strCheck, String strAgainst) {

   boolean bResult = false;

   // Add code to check if a subsequence exists
   // Only check if it is no longer than the against string:
   if (strCheck.length() <= strAgainst.length()) {
      int i = 0;
      for (int j = 0; i < strCheck.length() && j < strAgainst.length(); j++) {
         if (strCheck.charAt(i) == strAgainst.charAt(j)) {
            i++;
         }
      }
      bResult = (i == strCheck.length());
   }
   return bResult;
}
```

Now that we have an algorithm that determines if one string is a subsequence of another, we will consider an algorithm that will find the longest string that is a substring of two input strings. This algorithm will take a *brute force* approach to generating all and checking all possible subsequence (note: while the algorithm will work, there are much more efficient algorithms that should be used for this problem!). The algorithm will be used within the Longest Common Subsequence application. Pseudocode for the application is as follows:

```
Longest Common Subsequence (first, second)
  Create an empty bag
  Put the first string into the bag
  Assign the empty string to the longest match (subsequence)
  LOOP (while bag is not empty)
        Remove a test string from the bag
        IF longest match is shorter than test string
            IF test string is a subsequence of the second string
                  Set the longest match to the test string
            Otherwise if the test string is at least longer than the longest match
                  Generate new strings from test by removing each single character
                  Put the new strings into the bag
        Print the bag of strings to check
  Report the longest match
```

One detail of the algorithm that we need to explore further is the step that generates all strings that are one smaller than the test string. Given a position in the string, we can use the **`substring()`** method to get the characters before and after that position:



For example, use of the **`substring()`** method to generate a new string that concatenates the characters before and after the character located at position would be:

```
newString = checkString.substring(0, position) +
            checkString.substring(position+1);
```

Furthermore, with the fixed-sized array implementation of the ADT bag, we need to be careful with the **`add()`** method, which may not always succeed, particularly if there is not enough free space to store another entry in the bag. In such cases, the **`add()`** method will return **`false`** and the entry will not be added, which may have an effect on the application. Subsequently, every time we add a new entry into the bag, we need to examine the returned value. If we ever receive false when trying to add an entry into the bag, we can immediately stop the simulation and report that there was a problem.

Continuing with your Java project Bag from Section 2.2/2.3, extract the Java class file *LongestCommonSubsequence.java* from the **Week 2 Lab Files** archive on Blackboard, then copy the file into the **src** project folder on your computer. The *LongestCommonSubsequence.java* file contains the skeleton of the **`LongestCommonSubsequence`** class, which will be completed during this exercise. To complete the implementation of the application, carry out the following steps (at each step where the application is compiled and run, check the expected result is obtained, otherwise revise the code and retest):

1. Examine the skeleton in *LongestCommonSubsequence.java*.

   *Compile and run the **`main()`** method. If all has gone well, the program will run and accept input. It will report that the strings to check is null and an exception will occur.*

2. The goal now is to create the bag of strings to check. In the **`main()`** method, create a new bag and assign it to the **`testBag`** variable. Add the input string **`strFirst`** to the bag.

   *Compile and run the program. Enter ABD and BCD for the two strings. You should see **Bag [ABD]** and an exception will occur when it attempts to check the strings.*

3. The next goal is to remove a string from the bag and check if it is a subsequence of the input string **`strSecond`**. This check will be encapsulated in the method **`isSubsequence()`**, as given above. In the **`main()`** method remove an item from **`testBag`** and store it in the string variable **`strTest`**.

4. In accordance with the pseudocode given above, call the method **`isSubsequence()`** with the string you just removed from the bag (**`strTest`**) and the second input string (**`strSecond`**). If the method returns true, report that it was a subsequence and set the longest subsequence to **`strTest`**.

   Compile and run the program. Enter A and ABC for the two input strings. The code should report that it was a subsequence, with A being the longest common subsequence.

5. The next goal is to complete the body of the **`while`** loop in the **`main()`** method. Parts of the **`while`** loop have already been completed; however, you will now need to complete the code for the following steps:

   ```
   Generate new strings from test by removing each single character
   Put the new strings into the bag
   ```

   To do this, you will need to create a for loop that iterates over the positions of characters in **`strTest`** and creates the smaller test string (**`strNew`**) using the **`substring()`** method, as previously discussed. For each smaller test string created, add the new string (**`strNew`**) to the **`testBag`**.

   *Compile and run the program. Enter ABCD and FAC for the two input strings. The code should initially report the new bag of strings to check is **Bag[BCD ACD ABD ABC]** and the longest common subsequence eventually obtained is AC.*

   *Run the code again and enter BA and ABCA for the two input strings. The code should initially report the new bag of strings to check is **Bag[ ]** and the longest subsequence eventually obtained is BA.*

   *Run the code again and enter D and ABCA for the two input strings. The code should initially report the new bag of strings to check is **Bag[ ]** and the longest subsequence should remain empty.*

   *Run the code again and enter ABC and CBACBA for the two input strings. The code should initially report the new bag of strings to check is **Bag[BC AC AB]** and the longest common subsequence eventually obtained is AB or BC or AC*.*

   ** Note: the longest common subsequence may not necessarily be unique.*

Please upload a screen shot of your output (from the final run above) to the 2.3 Challenge link in the Progress Monitoring section on Blackboard