Analysis

(a) Initially, I had created a "greedy" algorithm. The pseudocode for this is below.

```
/* Java/pseudocode for ``greedy'' interleaving algorithm */
int i = 0;
int j = 0;
for (item : s) {
   if (item == x[i]) {
      increment(i);
   }
   else if (item == y[j]) {
      increment(j);
   }
   else {
      return false;
   }
}
return true;
```

In this algorithm, the increment() method performs i++/j++ if i/j is not the last character of x/y. If i/j is last character of x/y, then reset i/j to zero. In some cases, this algorithm produces the correct result and runs in O(n) time - for example, in the case given by the prompt. However, a simple example reveals that it does not always produce the correct solution. For example, consider the case where $x = 00$, $y = 01$, and $s = 01$. The above algorithm would return *false* when it is clear that s is in fact an interleaving of x and y. Clearly, a better solution is required.

Thinking back to recent topics covered in class, it was evident that this is an ideal problem for a bottom-up dynamic programming algorithm. In this case, if s is an interleaving of x and y, then any subproblem of s is also an interleaving of x and y. Thus, the simplest algorithm that produces the correct result simply iterates through an s.length by s.length array and calculates subproblems of increasing size until reaching the maximum problem size (occurs when, given a solution array a[i][j], $i + j = s.length$). An example of this is shown in Figure 1, and the Java/pseudocode is provided below. Because s can be an interleaving of *repeats* of x and y, and this could possibly be given where s is simply a repeat of only x or only y, it is neccesary to "extend" x and y to be the same length as s.

```
/* Java/pseudocode for dynamic programming interleaving algorithm */
// Extend x and y to s.length (e.g. x=101 and s.length=7 --> extend(x) = 1011011)
x = extend(x);
y = extend(y);

// Create a 2D boolean array to store solution to the subproblems
boolean[][] a = new boolean[lenS+1][lenS+1];

// Nested for-loops to evaluate all subproblems
for (int i = 0; i <= s.length; i++) {
   for (int j = 0; j <= s.length; j++) {
      // This ensures only the the upper left diagonal is evaluates (i+j <= s.length)
      if (i + j <= s.length) {
         // This evaluates the top left square
```

```
        if (i == 0 AND j == 0){
           a[i][j] = true;
        }
        // This evaluates the top row
        else if (i == 0) {
           a[i][j] = a[i][j-1] AND (y.charAt(j-1) == s.charAt(i+j-1));
        }
        // This evaluates the left column
        else if (j == 0) {
           a[i][j] = a[i-1][j] AND (x.charAt(i-1) == s.charAt(i+j-1));
        }
        // This evaluates everything else
        else {
           a[i][j] = (a[i][j-1] AND (y.charAt(j-1) == s.charAt(i+j-1))) OR
                     (a[i-1][j] AND (x.charAt(i-1) == s.charAt(i+j-1)));
        }
     }
  }
}
```

i, j	0	1	2	3	4	5	6	7	8	9
0	T	F	F	F	F	F	F	F	F	F
1	T	T	T	T	F	F	F	F	F	
2	T	T	T	F	F	F	F	F		
3	F	F	T	T	F	F	F			
4	F	F	F	T	T	F				
5	F	F	F	T	F					
6	F	F	F	T						
7	F	F	F							
8	F	F								
9	F									

Figure 1: The example from the prompt (x=101, y=00, s=100010101). The subproblems are solved first, starting at $(0, 0)$ which is always true. Squares colored in red indicate the possible "path" to the solution, and the square colored in blue indicates that s is an interleaving of x and y. Squares colored in grey represent the indices. A square can only be evaluated as true if either square to the left or above is also true *and* if the character matches in s.

From the pseudocode provided above, it is clear that the algorithm runs in $O(n^2)$ time, where n is the length of the string s. Accessing array elements and the charAt() function are both constant-time ($O(1)$), and so the net complexity of this dynamic programming algorithm for determining whether a binary string s is an interleaving of two other binary strings x and y is $O(n^2)$.

(b) See code included in the src folder of zip file for algorithm implementation. Run time was evaluated by generating randomized test files. Ten each of the following were created and evaluated (numbers indicate string length):

- $x = y = 5$ AND $s = 20$
- $x = y = 10$ AND $s = 100$
- $x = y = 100$ AND $s = 1000$

In my code I did include a counter that incremented each time a comparison was made. However, comparisons were only included for the if-else if-else statements, and not the logical comparisons within each statement. Thus, the final tally will be off by some constant value. Regardless, the time complexity is the same $O(n^2)$.

The number of comparisons made per run is provided in each output file. The results are summarized here.

- $x = y = 5$ AND $s = 20 \longrightarrow$ # comparisons = 902
- $x = y = 10$ AND $s = 100 \longrightarrow$ # comparisons = 20502
- $x = y = 100$ AND $s = 1000 \longrightarrow$ # comparisons = 2005002

From this, we can see that the time complexity is indeed $O(n^2)$ (see Figure 2). Thus, we have used dynamic programming to create an algorithm at efficiently and *correctly* determines whether or not s is an interleaving of x and y. In addition, it is worth noting that creating test cases for this is quite tricky. Of the thirty randomized binary strings tested, none of them were interleavings.

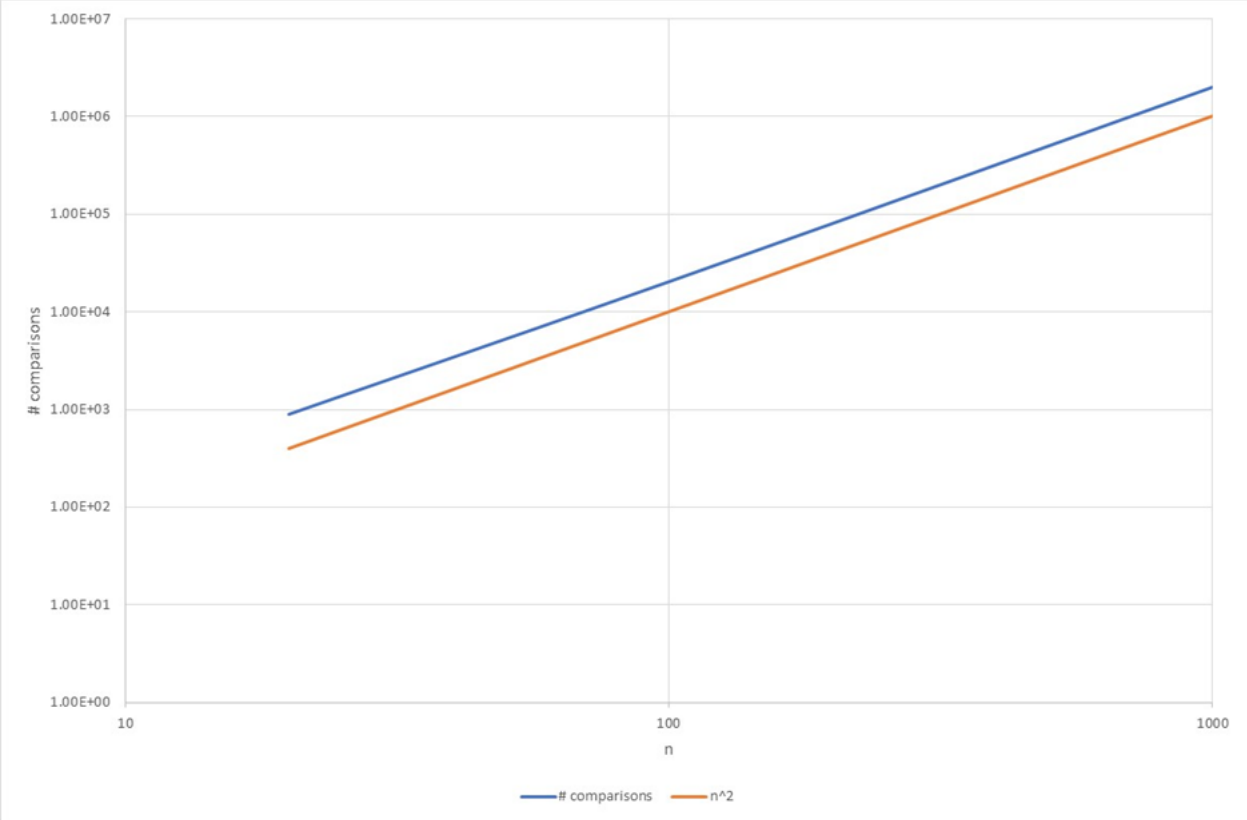

Figure 2: The counted number of comparisons for each s.length.