**Review and Research**

You can review the video of Session 6 here:

<https://blizzard.sharepoint.com/portals/hub/_layouts/15/PointPublishing.aspx?app=video&p=p&chid=8aa7fa80-bfa2-4021-bf44-543dba93f693&vid=c298282a-4bb9-4d68-93cd-e9be170105a3>

As always, email me at [semerson@blizzard.com](mailto:semerson@blizzard.com) if you have any questions!

**Problem Solving Techniques**

* Understand the problem before you begin planning a solution. Make sure you know what kind of input to expect and what should be done with that input. Think about possible edge cases and how you might deal with them. Most of all, make sure you’re solving the right problem!
* Plan before you code. It is tempting to jump straight into your development tools and begin writing code immediately, but a bit of planning will ultimately save you time, make your code less error-prone, and help prevent the frustration of not knowing how to proceed. If you find that you hit a wall when writing your code, don’t be afraid to stop and re-examine your plan (and revise it if necessary). When planning, you don’t necessarily need to focus on language features; think about the problem in real-life terms, and feel free to sketch out your ideas on paper to help make them more concrete.
* Break big problems down into smaller problems. When we implemented bubble sort in class, we first thought about how to sort a single number in an array rather than *all* the numbers. We then tackled how to repeat the process, how to manage the number of iterations in each loop, how to know when to stop, and how to swap values. By thinking about solutions to each of these smaller problems in their turn, we chip away at the larger problem until we have a workable solution.
* Use concrete examples when you plan. If you’re trying to devise a sorting algorithm, don’t just think about sorting in the abstract: Write down simple sets of actual numbers (or other values) to work with, and from those specific examples begin to create generalized solutions. Problem solving is almost always easier to do when you work with concrete values rather than jumping straight into purely abstract concepts!
* Consider extreme cases, i.e., extremely large cases or extremely small cases. This is useful for solving all kinds of problems! Consider this simple, classic riddle: You have a drawer with 10 red socks, 12 blue socks, and 5 yellow socks inside. If you begin pulling socks out of the drawer at random without looking at the color, how many socks must you take to guarantee that you grab at least one red sock? One way to easily come up with a general solution to this kind of riddle is to use an extreme case. Let’s go for an extremely small case and imagine we have exactly one sock of each color. It’s very easy to see that, in order to be sure you get the red sock, you have to take all three socks out of the drawer; until the yellow sock and blue sock have been removed you’re not guaranteed to get the red sock. Once we understand this, it’s easier to generalize for other cases: To be guaranteed a red sock, you must take the total number of yellow and blue socks, plus one more. Believe it or not, in software engineering we often solve very complex problems using just this kind of thinking! When choosing concrete examples to work with, don’t be afraid to start with extremely simple cases; the solution is usually easier to find, and will often scale up to work for more complex cases as well.

**Problem Solving Methodology**

1. Understand the problem. Make sure you’re clear on input, output, and expected results. Consider edge cases carefully. If the problem in complex, break it down into several smaller problems.
2. Plan your solution. Don’t assume you have to think in terms of language features at this point; just come up with a step-by-step process that will solve the problem. Test your solution on paper with concrete examples to demonstrate that it works correctly.
3. Translate your solution into language features. If you’re not sure which features to use, review the fundamentals and look up new functionality as necessary.
4. Write code that follows your plan.
5. Test your code with a number of examples to satisfy yourself that your solution is working correctly. This is a good time to try out edge cases!
6. Modify your program if necessary and test again until all your test cases pass.

**Practice Exercises**

NOTES: Although most of the below exercises instruct you to write a function, remember that you can split your solution into multiple functions if it makes your solution easier to code. In many cases you may be able to solve these problems very easily by using methods that are already available as part of the .NET framework; in these cases, try creating multiple solutions using both built-in methods *and* your own algorithms!

Ex. 1. Write a function that, given a List of integers, removes all copies of the highest and lowest value from the list. Challenge: Try to do this without using the List.Min() or List.Max() methods!

Ex. 2. Write a function that, given two strings as input, determines how many times the second string appears as a substring of the first string. For example:

Str1: “ababa” Str2: “aba” Result: 2

Str1: “aabaab” Str2: “ba” Result: 1

Str1: “abbab” Str2: “ab” Result: 2

Str1: “aaaaa” Str2: “ab” Result: 0

Ex. 3. Write a function that, given an integer as input, returns a List<int> of all the distinct prime factors of the input integers. For example:

Input: 10 Output List: 2,5

Input: 60 Output List: 2,3,5

Input: 19 Output List: 19

Ex. 4. Write a find-and-replace function which takes two strings and a character as input. The function should search the first string for all instances of the second string and, upon finding them, replace them with the character you provided. Return a new string with the completed replacements. Examples:

Input: “Hello, I was looking for Han Solo.”, “lo”, ‘\*’

Output: “Hel\*, I was \*oking for Han So\*.”

Input: “I am smitten”, “sm”, ‘k’

Output: “I am kitten”

Ex. 5. Write a function that function that, given an integer *n* as input, returns a list of all prime numbers from 2 through *n*, inclusive.

Ex. 6. Write a function that takes a string as input and returns a new string with the following properties:

1. Contains no non-alphabetic characters, including numbers and whitespace.
2. Converts all alphabetic characters to upper case.
3. Contains no duplicate characters.
4. Arranges all characters in alphabetical order.

Examples:

Input: “Sandwich!”

Output: “ACDHINSW”

Input “No, really?!?”

Output: “AELNORY”

Input: “The quick brown fox jumps over the lazy dog…”

Output: “ABCDEFGHIJKLMNOPQRSTUVWYZ”

Ex. 7. Write a function that, given an integer as input, determines whether or not the given integer is a Fibonacci Number. For the purposes of this exercise, please consider “0” to be a Fibonacci Number.

Ex. 8. Write a function that takes two strings of equal length, *s1* and *s2*, as input (assume for this exercise that all input is well-formed… i.e., that it follows the preceding definition). Return a new string that interleaves the characters of the first string with those of the second string. This should be done by alternating characters from *s1* and *s2* as follows: *s1[0], s2[0], s1[1], s2[1], s1[2], s2[2], s1[3], s2[3],* etc. Here is an example to illustrate the concept:

Input: “Apple”, “Bread”

Output: “ABprpelaed”

Ex. 9. Write a function that, given a List<float> as input, returns a List<int> that contains all the values of the input List, rounded to the nearest integer and sorted into reverse numerical order. For example:

Input: { 1.5, 7.92, -5.1, 2.7 }

Output: { 8, 3, 2, -5 }

Ex. 10: In class we examined the bubble sort algorithm and implemented it together. Try looking up and implementing two other sorting algorithms: “insertion sort” and “merge sort.” Test your algorithms with Lists of integers.