Benjamin Leibowitz
Lab 1

Description of data structures
This lab was meant to focus on stacks. Though I did not see the utility in stacks at the beginning of the module, this lab demonstrated their ease of use and ability to solve specific problems. To create a stack (of characters) for this lab, I created a CharStack class that has three class variables:

1. *MAX_LENGTH* - a static int representing the capacity of the stack (78)
2. *stack[]* - a character array of length *MAX_LENGTH* serving as the infrastructure for the stack
3. *top* - an int representing the top of the array, initialized to -1

The CharStack class has five methods:

1. *push()* - adds an element and adds it to the stack, and moves the top up one
2. *isEmpty()* - checks whether the stack has items yet
3. *pop()* - removes the top item from the stack and moves top down one
4. *peek()* - returns but does not remove the top of the stack
5. *print()* - prints the entire contents of the stack and was used for debugging purposes only

Justification of data structure choices and implementation
The use of stacks may not have been the most efficient way to solve this problem. It likely would have been much easier both in development and at runtime had the length of the input string been measurable, and if random access data structures were permitted. Had length measurement been permissible, I would likely have used an array to utilize random access and the length of strings and substrings. This would have cut the number of operations that had to be performed. For example, oftentimes the lengths of substrings had to be compared. Had an array implementation been used, that would have taken one operation - the comparison between the lengths of the two arrays. However, because this was not permitted, the stacks had to be emptied, then their contents compared to ensure they were empty. This is an O(n) solution for something that could have been O(1) if arrays were used. This would have made a trivially easy lab, though. Because the length measurement was not permissible, stacks provided an elegant solution.

Discussion of the appropriateness to the application
Upon reading the problem statement, I had initially thought that though the lab encouraged the use of stacks, there had to be a better solution. That is, until I tried for days to get the algorithm for L4 to no avail. On my way home from work one day, it clicked that stacks were the most logical and appropriate implementation. Their LIFO behavior provides the ability to test lengths of strings by popping all contents from the stack and comparing whether both are empty. This simple concept did not occur to me until I struggled through these problems only to realize that stacks are highly elegant data structures, mainly due to their simplicity. Given the problem statement, stacks seemed to be the most appropriate solution, though other approaches could have been taken, such as recursion.

Description and justification of your design decisions

Benjamin Leibowitz
Lab 1

Several design decisions had to be made - first and foremost was whether to use stacks or recursion. A recursive algorithm could certainly have been implemented for L4. A recursive call could be used to find the boundaries between *B*/*A* to ensure that the substring of the base case consisted only of a sequence of As then Bs. If it did not, then the recursive implementation would return false. If it did, then a recursive call could be made to the next substring of one sequence of As and Bs for the next *p*, and so on, to ensure that *m* and *n* were the same for each *p*. Essentially, rather than pushing and popping to stacks to test the lengths of substrings, the number of recursive calls for A or B could be counted to measure lengths instead. However, this likely would have been difficult to implement and interpret.

Efficiency with respect to both time and space
Nor would it likely be efficient from a run-time perspective. Iterating through a recursive algorithm would likely have taken longer due to the testing of all potential base cases each time the recursion is called. However, it is likely that recursion would have been more efficient from a storage perspective, as integers (counting) would be used to test the lengths of strings, rather than a stack of characters (array / list), which requires more data storage. As mentioned prior, had length measurement been permissible, testing lengths would have been O(1) rather than O(n) to pop all the stacks and test the emptiness of their contents.

Several other design decisions had to be made. One was how to know when the end of the string was reached. Because we couldn't test the length of the strings, I made use of the Character.MIN_VALUE character to ensure that I could control the end of the string and know when it was over, even if not being able to know the index. Another design decision for L4 in particular was how to know whether *m* and *n* were consistent each *p*, this one took some thinking, only to realize that this could be done by adding a condition for the first run, which I called *first_run*. The job of this case was not only to test any failure modes, but also to keep track of the true values for m and n going forward. Because they could not be stored as integers, they were stored in the form of stacks, *m1* and *n1*. These are some design decisions I thought of as "workarounds" to accommodate for the problem statement, that in hindsight were design decisions.

What you learned
Despite my qualms with making a simple problem very difficult by placing bounds around it (such as not being able to measure length of a string), I am very grateful for the learning opportunity. Had it not been for this lab, I would not have fully appreciated the utility and simplicity of stacks. Also, one valuable lesson learned was that just because you think of failure modes or test cases in a certain order, does NOT mean they should be coded as such. For example, several times throughout this lab, I wrote out conditionals as they popped into my head (pun intended). After debugging, I realized that I had basically entered two different test cases for the same iteration because I had flipped a boolean variable in a preceding test case within the loop. This got me into trouble, and many times I had to reorder logic

from how it first made sense to me. This may have been one of the most valuable lessons from this lab, despite the seeming triviality of it.

What you might do differently next time

Next time, I'm not sure if I would have coded anything differently, but how I went about the lab would be very different. Instead of trying to code everything in a few-day sprint, I am going to start the lab early and hit some roadblocks, and spend more time thinking about solutions in passing. I found during this lab that my most productive time spent was away from the computer thinking about the logic behind the different algorithms. Once it made sense in my head, I could open my computer and implement the logic with relative ease. I am pleased with the solution used given the parameters and constraints of the problem and not sure I would code anything differently given more time.

Discussion of anything you did as an enhancement

As an enhancement, I created a language that is met if no character is used twice consecutively. Though it is relatively simple, it can be quite useful if there are few characters (such as a password) and the goal is to create a unique string that is difficult to guess as a hacker. The more different the characters are, the less likely a hacker is to guess the password correctly. Forcing lack of consecutive identical characters could be useful in a use case such as this. Additionally, it allowed us to break out of the constraint placed in the first four languages, where the characters could only be binary (A or B).