Data Structures, Lab
CRN: 19897

Salvador Robles Herrera
ID: 80683116

Lab #3 Report

Problem 1

This problem consists on creating an ArrayList class that will work as a normal ArrayList, with the same functionalities like get and set an item given an index, get the length of the ArrayList, append and prepend values, insert a value at a given index, remove an element given a certain value and delete an element given an index. The ArrayList must be implemented with an Array (class already created).

Note that:

- Delete index functionality must be done in O(n), except when index = 0 or n-1.
- Remove value must be done in O(n), except when the value is the first element.
- Insert value at index must be done in O(n), and call append or prepend when index = 0 or n respectively.
- Prepend and append must be done in O(1), except when the array needs to be larger to fit more elements.
- Contains must be done in O(n)

The main idea of this problem is to "think circularly", so my first idea in approaching this problem is to prepend an element at the end of the inner Array, so if the inner Array is like: 1,2,3,x,x,x,x then to prepend "0" the Array would look like: 1,2,3,x,x,x,0. To append "4" into the previous array it would result the Array to 1,2,3,4,x,x,0. That's why my approach to this functionalities is to have think of the Array as in two mini arrays, the left part and the right part, where the right part are the elements that are being prepended. But there is a problem when the array has no elements on the right part (prepended elements) and the user wants to delete the first element in the array.

My approach to this problem was to have three pointers in the inner Array. The first pointer represents the index of the first element of the left part of the array (the first mini array). The second pointer represents the index of the element right next to the last element (that way you can see where to append a value). The third pointer is the element left next to the first element of the second mini array (the part where to prepend), that way it's easier to prepend an element.

When appending a value, the element is going to be placed on the index being pointed to by the second pointer, and then the second pointer will be updated by incrementing its value by one, observe that the size of the ArrayList must be incremented by one as well since an element has been added, this would result in O(1) since those are just constant operations.

When prepending a value, the element will be placed in the first mini array (left part) if there is space at the index pointed by the first pointer (this way the first pointer is decreased by one), otherwise it will be placed at the index pointed by the third pointer (this would be on the right

Salvador Robles Herrera
ID: 80683116

part of the array) and the pointer would be decreased by 1. After that the size of the ArrayList will decrease by one.

When appending a value, the element will be placed at the index the second pointer is pointing and the second pointer will be incremented by one. Note that when appending a value it is important to check if the inner Array is already filled, because then it will be need to create another inner Array with two times the size of the last one and passed all elements to the new one (starting with elements on the right mini Array and then the left mini Array).

When inserting a value at a specified index, it's necessary to shift all elements with a bigger index to the right. So, there are two scenarios: where the index is at the right part of the Array or at Left part of the Array. Note that if the index is at the right part, it is necessary to shift the elements of the right mini Array to the right (all indexes greater than), then shift all elements of the left part to the right, note that the last element from the first part will be placed as the first element on the left part of the Array (pointer and size are updated). Otherwise if the index is at the Left part of the Array the only elements needed to be shift to the right are those elements in that part being greater than the index. Note that when inserting a value, it is needed to check if the array is full, if needed the Array will duplicate its size.

Deleting an index has the same concept as inserting a value but now the elements with a greater index will be shifted to the left instead of to the right, the current size of the ArrayList will be decreased.

When removing a value, I used the deleting method, it will traverse the array until the value is found and then the deleting method will be called with the respective index, otherwise if it's not found then the method will raise an Exception.

When searching for a value with the "contains" method, it is needed to traverse all the array, the right part first and then the left part, and then returning true if its encountered or false if not.

I tried to test the algorithm with this instructions and this worked:

```
nums = ArrayList()
nums.append(100)
nums.append(150)
nums.append(200)
nums.preprend(50)
nums.preprend(0)
nums.insert(2,75)
nums.delete(2)
nums.remove(200)
```

Problem 2

Salvador Robles Herrera
ID: 80683116

Solution 1:

The second problem consists in shifting the elements of an array k spaces to the right and placing them circularly at the beginning. The array and k are passed as parameters, in this part it is just needed to solve the problem.

Without thinking on running time or space time complexity restrictions, I thought on just putting the last k elements of the array into a list by traversing those elements and appending it to the list. Then the other n – k elements, where n is the length of the array, should be shifted to the right so that those elements are the last elements in the original array. Finally, just update the first k elements of the original array with the values on the list created previously, those elements should be placed in order at the beginning. Also note that not doing any shifting is the same as shifting n spaces to the right, therefore k should be replaced with the value of k mod (n).

The runtime complexity would be O(n) since at most it is needed to traverse the array 3 times with constant time instructions. At most the list created for the last k elements is n-1 in length, which results in O(n) in space complexity.

The test cases I used for this problem were the following, which returned the expected outcome in every one of them. Some cases have k being less or more than the length of the array, as well when the array has 0 elements.

```
print("What I got: ", circular_shift_1([1,2,3,4,5], 7), "Expected: [4,5,1,
2,3]")
print("What I got: ", circular_shift_1([1,2,3], 1), "Expected: [3,1,2]")
print("What I got: ", circular_shift_1([], 1), "Expected: []")
print("What I got: ", circular_shift_1([1], 5), "Expected: [1]")
```

Solution 2:

This solution has some restrictions on how to solve the problem, now it has to be solved in place, meaning that there should be a constant space solution.

Seeing after a few tries on trying to find the solution that the array can be viewed in having two little arrays of k and n-k elements, since it is wanted to place the last k elements to the beginning of the array and the first n-k elements to the end of the array, that would the same as reverting the whole array (placing the last element to be the first, etc.) and that way the last k elements would be at the beginning and the first n-k elements would be at the end of the array, just not in place yet. Now remember that by viewing the array like two little arrays, we can appreciate that the two little arrays are just in not order, the little array are in place but need to be reversed. Then by reverting the little arrays you will get that all element are in order, note that this is done by reverting the first k elements and the reverting the last n-k elements.

The time complexity of this algorithm is O(n) where n is the number of elements in the array, this because at most the array is being traversed 3 times with constant time instructions. All is being

Salvador Robles Herrera
ID: 80683116

done by swapping elements on the array, so the space complexity is O(1) and therefore it satisfies with the restrictions of the solution.

I tested this solution with the following test cases, which had the expected output.

```
print("What I got: ", circular_shift_2([1,2,3,4,5], 7), "Expected: [4,5,1,
2,3]")
print("What I got: ", circular_shift_2([1,2,3], 1), "Expected: [3,1,2]")
print("What I got: ", circular_shift_2([5,3,1,7,9], 2), "Expected: [7,9,5,
3,1]")
print("What I got: ", circular_shift_2([1], 5), "Expected: [1]")
```

Problem 3

As the problem states, you need to compare two strings and determine if it's possible to edit a string to be exactly the same as the other string. You can only use one edit which consists on inserting a character, remove a character or replacing a character.

At first, I thought of doing a matrix of n x m where n is the length of the first string and m is the length of the second string. I remember seeing a similar problem, where every entry in the matrix is the least number of edits to transform the string. For example, the entry in the matrix "i, j " would be the least number of edits between the first "i" elements of String 1 and the first "j" elements of String 2. That way the entry of the matrix "n, m" would be the result, since that would be the least number of edits between String 1 and String 2, to accomplish this idea I needed to find a way of filling the entries one by one starting at "1,1" and then "1,2", etc. this by finding the number of edits by knowing previous entries on the matrix. Unfortunately, I couldn't find the correct way of continuing with that idea, so I tried something else.

Note that there are only three ways performing an edit: inserting, replacing or removing. I realized that inserting a character on the first string would be the same as removing a character from the second string, this would be the same the other way around, and that replacing a character from String 1 was the same as replacing a character from String 2. Therefore, it is only needed to prove if there is a way of making an edit on String 1 to get String 2, no need to edit String 2.

The first thing to check between the strings is the difference in length of the strings, if the difference is more than one, then there should be two edits at least to make String 1 = String 2, in other words the method would return false. If the length of String 1 is one more than the length of String 2 then the only possibility is to remove a character and then the remove method will execute. If the length of String 1 is one less than String 2 then the only possibility is to insert a character so the insert method will execute. If the length of the strings are the same then the only possibility is of replacing a character.

Salvador Robles Herrera
ID: 80683116

If there is needed to remove a character, and the length of String 1 is n, then there are only n possibilities of choosing which character to remove from String 1, so the method traverses through those possibilities and checks if those are equal to String2, when you find strings that are the same return true, otherwise return false. For inserting a character, there are n+1 possibilities since you need to place a character before the 1st character of String 1, or before the 2nd character of String 1, …, before the n character of String 1 or at the end of String 1, note that the character inserted would be the character at that place of String 2, this since we need both strings to be equals. For last, to replace a character it is needed to change every character from String 1 with its respective character on String 2.

The runtime complexity of the algorithm would be O(n) since in any of the three methods it is needed to recurse through at most n+1 possibilities that would perform a constant number of instructions. The space complexity would be O(n) since in each of the three possibilities on edits it is needed to at least create one string (you can't change a single character of a string, it has to be created a new one) where n is the length of String 1.

I tested my algorithm with these cases, where the method needed was remove, insert, replace, remove and replace respectively. All cases returned what expected.

```
print("What I got: ", one_edit_away("pale", "ple"), " Expected: True")
print("What I got: ", one_edit_away("pales", "pale"), " Expected: True")
print("What I got: ", one_edit_away("pale", "bale"), " Expected: True")
print("What I got: ", one_edit_away("pale", "bae"), " Expected: False")
print("What I got: ", one_edit_away("pale", "pale"), " Expected: True")
```