Lab #2 Report

Problem 1:

Solution 1:

The first problem consists in sort the array of passwords objects using bubble sort, and then return the k most used passwords.

I used a subset of passwords since it was taking a lot of time compiling when testing, the number of passwords in the subset can be changed easily since I created a method that selects the elements to be sorted. Bubble sort is needed to sort the array by swapping adjacent elements if those are not in the correct position, this by traversing the array n times, where n is the number of elements to traverse. Note that I added a condition where k must be less than the number of elements to sort. Then the final step would be to create an array that will contain the k most used elements, which are going to be the last k elements in the original array of passwords, this since the array is sorted in an increasing order.

The run time would be of O(n^2) since it's using bubble sort and other operations are constant, and the space complexity is O(1) since all changes are being in-place and the array created would have k elements, where k is constant to O(1) in space complexity. I tested this case which worked:

```
Solution1Prob1(2)
```

Solution 2:

The second problem is almost the same as the first problem, but now to sort the array of passwords objects it is needed to use quicksort and then return the k most used passwords.

As the last solution I also used the same method for subset gathering of passwords. I used a recursive quicksort method, which includes two methods "quicksort" and "partition" where the first method calls the second method, which chooses a pivot and place all larger elements to the right of the pivot and place the smallest elements to the left of the pivot, then the first method calls recursively on the left and right side of the pivot. The final step as the last solution would be to create another array that will contain the k most used passwords.

The runtime would be of O(nlogn) since that is the time complexity of quicksort, and there are no other operations that would change form O(nlogn). I tested this case which worked where k is 2:

```
Solution2Prob1(2)
```

Solution 3:

The third mini-problem consists in implementing a modified version of quicksort that makes only one recursive call, and that as same as the last two mini-problems returns the k most used passwords.

I didn't have time to implement the solution of the problem, I spent a lot of time trying to solve solution 2 of problem 3, but I think I have the correct idea in how to solve the problem. To make just one recursive quicksort call recall that it's only necessary to return the k most used passwords, therefore it depends on the pivot and how many elements are to the right and to the left of it. In this case it is easier to apply quicksort to try and sort the array in a decreasing order, so when calling the "partition" method the elements that are bigger than the pivot should be on the left and the elements that are less than or equal to the pivot to the right.

If k is less than the numbers on the left of the partition, the recursion continues only on the left part, this since the top k passwords would be on that part, and there is no necessity to make a quicksort call in the right part of the partition. Else if k is bigger than the elements on the left side of the pivot, we add the left side on a new array that will contain the k most used passwords, and then it would take a recursion call on the right side of the pivot where now the k has changed to k – (number of elements on the left side). When k is 0 then we will have an array of the k most used passwords but not necessarily in order, now what's left is to sort the array of k elements and now it's done.

Problem 2:

Solution 1:

There is an array of color objects red, green, and blue, that are represented by integers 0, 1, 2 respectively. The problem is to sort the array.

The first thing that comes to mind, and obviously not the most optimal solution would be to implement the easiest sorting algorithm I know, and that would be the easiest to code and know with certainty that it works, which is to apply insertion sort. There is no other work in the method apart from insertion sort, so time complexity would be of O(n^2) and since it's in-place, it has O(1) of space complexity.

This are the examples I tested, where there are more two's than one's or more one's than two's etc. All tests gave the expected values.

```
print("What I Got: ", color_sort_1([2,1,1,0,1,2]), "  Expected: [0, 1, 1,
1, 2, 2]")
print("What I Got: ", color_sort_1([2,1,2,0,1,2]), "  Expected: [0, 1, 1,
2, 2, 2]")
print("What I Got: ", color_sort_1([2,1,2]), "  Expected: [1, 2, 2]")
```

Solution 2:

Now there are more prerequisites, the solution must have a O(n) run time and O(1) space complexity, therefore, it should be more optimal than the previous solution.

I noted that there are just three possible values that an element of the array would have which is 0, 1 or 2. It occurred to me that it would be faster to go through the array and count how many 0's, 1's and 2's the array contains and then just place those numbers in order. To place them in order I thought that it would be faster to create three for loops, each one to place one of the three possible numbers. This could be easily achieved since the 1's should be place after all the 0's have been placed, and all 2's should be placed after the 1's.

The time complexity would be of O(n). Note that the first step which is to count how many 0's, 1's, and 2's the array have is just a simple iteration through all the elements which is simple O(n). The next step is to place the 0's, 1's and 2's given the quantity of numbers counted previously. If the number of 0's is x, the number of 1's is y, and the number of 2's is z, then the algorithm would traverse the first x elements in the array and replace them with 0's, then it would traverse the next y elements and replace them with 1's and finally it would traverse the next z elements and replace them with 2's. That takes O(x) + O(y) + O(z) = O(x + y + z) = O(n), so the total time complexity would be O(n) and the space complexity is O(1) since we only use constant space.

I tested cases where there were only two different numbers, where there are more zero's than any other number, etc. Every test had the expected output.

```
print("What I Got: ", color_sort_2([0,1,2,0,1,1,1,2,0]), "  Expected: [0, 0, 0, 1, 1, 1, 1, 2, 2]")
print("What I Got: ", color_sort_2([2,1,1,0,0,0]), "  Expected: [0, 0, 0, 1, 1, 2]")
print("What I Got: ", color_sort_2([0,1,0,1,1,1,0]), "  Expected: [0, 0, 0, 1, 1, 1, 1]")
```

Problem 3:

Solution 1:

It is given an array in non-decreasing order of numbers, and the problem wants us to return the sorted array of those numbers but squared. If the array is [1, 2, 3], the returned sorted array will be [1, 4, 9].

The main problem is when the array has negative numbers, since when squaring a negative number will transform into a positive number and the position in the new sorted array will be different. Like in this example, if the array is [-3, -2, 1, 4] then the numbers squared are [9,4,1,16], but the array is unsorted. The first thing that comes to mind is to transform the negative numbers into its absolute value, since the order of the absolute values and the square of those numbers is the same. So, if the array is [-3, -2, 1, 4] the next step would be [3, 2, 1, 4], then I thought of the simplest sorting algorithm which is insertion sort, the array would result in

[1, 2, 3, 4], now the only step left would be to square every element in the array to finish with [1, 4, 9, 16]. Note that insertion sort has a time complexity of O(n^2) and O(1) in space complexity, the other steps takes at most n step which is O(n) and therefore the whole method would have time complexity of O(n^2).

I tested the method with these two examples, where there are negative and positive numbers, since that is where the main problem is, both cases had the expected values.

```
print(square_sort_1([-3,2,2,0,1,2,3]), "Expected: [0, 1, 4, 4, 4, 9, 9]")
print(square_sort_1([-7,-2,0,1,2,5]), "Expected: [0, 1, 4, 4, 25, 49]")
```

Solution 2:

The second solution needs a O(n) in time complexity and a O(1) in space complexity, therefore it is not possible to create another array of n size, just constant number of variables.

I noticed that there is a sorted array of the positive numbers from the previous array, and that there is a second array of previously negative numbers which would be -3 and -2 that when squared would be 9, 4 and therefore in a decreasing array order. This happen since -a < -b => a > b => a^2 > b^2. It is easy to transform an array from decreasing to increasing array, so it would result into two sorted arrays similar to merge sort. For the O(n) solution in time complexity I would use a kind of a merge sort, the first array would be from index 0 to the last previously negative number and the second array would be from the first previously positive number to the last element in the array. Unfortunately, the solution would have a O(n) in space complexity since it needs to create another array and have one pointer at every array and select the smallest and increment the pointer when necessary. I didn't find a O(1) solution and I don't think it is possible.

I tested the method but unfortunately, I forgot to the first step, which was to transform the array of negative numbers from a decreasing sorted array to an increasing sorted array.

```
print(square_sort_2([-3,2,2,0,1,2,3]), "Expected: [0, 1, 4, 4, 4, 9, 9]")
print(square_sort_2([-7,-2,0,1,2,5]), "Expected: [0, 1, 4, 4, 25, 49]")
```

Conclusions:

I think that in conclusion my favorite sorting algorithm is insertion sort, in this Lab I prefer that sorting algorithm when I try to sort something, and I don't want to think about time complexity or slower run times.

Something very interesting that I learned doing this Lab is that it's good to create a non-optimal solution and then trying to do the best solution possible. I think that in an interview is good to think first in any solution, it's better than nothing. Also I noticed that my first solution was very

similar to the second/optimal solution I implemented, so it's not a good idea to spend a lot of time thinking in the best possible solution when it's not so different from what I have in mind.

"I certify that this project is entirely my own work. I wrote, debugged, and tested the code being presented, performed the experiments, and wrote the report. I also certify that I did not share my code or report or provided inappropriate assistance to any student in the class."
-Salvador Robles Herrera