Time Spent: 2 hours

Collaborators and Resources:

Problem 2

For this problem I wanted to create an efficient algorithm which had an asymptotically tight bound of n (which requires O(n) time). Because the algorithm needed to be in O(n) time, I could check the n elements in the array one by one. The rest of the operations (identifying $A[i]$ and swapping $A[i]$ with $A[j]$ or with $A[k]$) can be done in linear time.
Since the largest value (2) needs to be located at the end of the array and the smallest value (0) needs to be at the beginning, I iterate through the array and if necessary swap the current value to append it to either the sequence of 0s at the beginning or the sequence of 2s at the end ...

Below I've created the pseudo-code for my algorithm ...

Given an unsorted array $A[1, \ldots, n]$ of n integers, each from {0, 1, 2}

1. Initialize i = 0
2. Initialize j = 0
3. Initialize k = length(A) - 1
4. While i $\leq$ k:
   (a) If $A[i]$ == 2
      i. Swap $A[i]$ with $A[k]$
      ii. Decrement k by 1
   (b) Else if $A[i]$ == 1
      i. Increment i by 1
   (c) Else if $A[i]$ == 0
      i. Swap $A[i]$ with $A[j]$
      ii. Increment j by 1
      iii. Increment i by 1

Claim 1. The algorithm effectively sorts the array using only the lookup and swap operations.

Proof. Essentially, my algorithm is going to look through everything in the array one by one. It will go from i = 0 to i = length(A) - 1, looking up $A[i]$ at each iteration. If it encounters a 0, it will use the swap operation to place the 0 to the right of the last verified 0 (since we're limited in operations we have to essentially check everything regardless of whether it's already in the right place). If it encounters a 2, it will swap it with whatever is on the left of the most recently verified 2. It doesn't move on unless it is no longer looking at a 0 or 2, and so reconstructs the head and tail ends of the array ... □

Claim 2. The algorithm has O(n) worst-case running time.

Proof. Not only does the algorithm have O(n) worst-case time but also has Ω(n) time (best-case). Because either i or k is incremented/decremented respectively in each iteration, it runs the sequence of operations (depending on the if and elif statements) n times. This is because when i reaches k the algorithm will no longer run and the array will be sorted. The example (while not a proof) demonstrates 9 sequences of operations (which is the length of the array) ... □

For Example.

212100122

212100122

212100122

112100222

*1***1**00222

*1*1**0***0*222

*1***0**1*0*2222

0*1*1**1***0*2222

0*1*11***0***2222

00*1*1*1***2**222

001112222