*Please enter your name and uID below.*

Name: Jason Crandall

uID: u0726408

**Submission notes**

- **(PS 1 specific) This is a warm-up assignment. Full points will be given for answers that have some correct elements and that clearly been given a legitimate effort.**

- *Solutions must be typeset* using one of the template files. For each problem, your answer must fit in the space provided (e.g. not spill onto the next page) *without* space-saving tricks like font/margin/line spacing changes.
- Upload a PDF version of your completed problem set to Gradescope.
- Teaching staff reserve the right to request original source/tex files during the semester, so please retain the original files.
- Please remember that for problem sets, collaboration with other students must be limited to a high-level discussion of solution strategies. If you do collaborate with other students in this way, you must identify the students and describe the nature of the collaboration. You are not allowed to create a group solution, and all work that you hand in must be written in your own words. Do not base your solution on any other written solution, regardless of the source.

1.

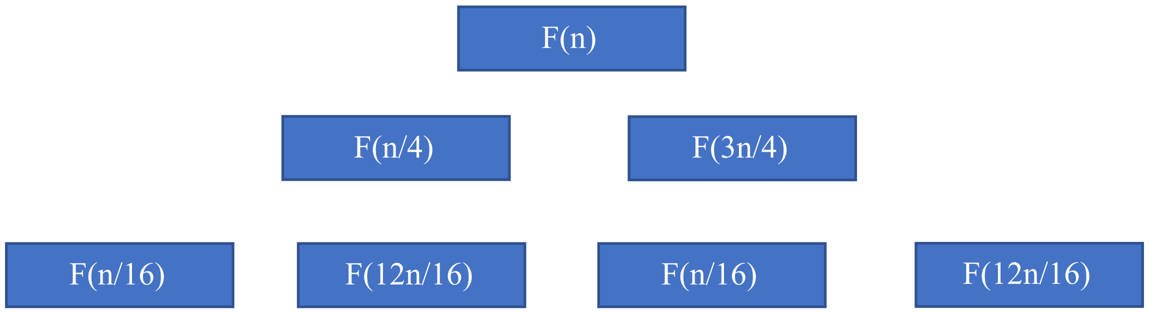
The key in analyzing the runtime of the modifiedMergesort function revolves around the bug found within the code. I will thus prove that the run time of modifiedMergesort to be $O(n^2)$. The bug in question is found within the alreadySorted in the for loop. Rather than having the for loop move from the left input to the right input, it starts from 0 going up to the right. As this moves through the entirety of the array, that gives a runtime value of $n$. Now, as this function resides within the parent modifiedMergeSort function, we can use recursion tree logic to determine the overall runtime of the function. The root node of the recursion tree will consist of the length of the entire array, leaving its value as $n$. The next level down consists of the array split in 2, but as shown by the run time of the alreadySorted function of $n$, that makes the total cost of the second level to be $2n$. Knowing that the depth of mergesort to be $\log_2 n$, we can thus conclude that the summation of each level of the recursion tree to be $n\sum_{i=0}^{\log_2 n} 2^i$. Following the rule of summations, we know this equation to equal $n(\frac{1-2^{(\log_2 n+1)}}{1-2})$. Reducing this further gives us $n(\frac{1-2n}{1-2})$ which when taking the priority of n as the highest power variable gives us $n^2$. Now in order to show this to be bounded by $O(n^2)$, for some c and k, then $n^2 \leq cn^2$, which for this to be true, c can be equal to 2. As shown here, this thereby proves the modifiedMergeSort function to be bound above by $O(n^2)$.

2.

The solution for solving the counting of inversions found in an array can be done by adding some additional checks within the existing mergesort functionality. We know that the time complexity of mergesort is $O(n \log_2 n)$, which is precisely the complexity we need for the solution. As such, if we add some non expensive logic to the existing function, we can achieve the same results. By keeping the mergesort function the same, we can add checks within the internal merge function. The merge function is where all of the comparisons occur, and it is here that we can add our logic to count the number of inversions. When we are comparing the locations in the array, we know that we are looking to order the array, meaning if we notice that the current item is less than the next one, then we know that the number of inversions would be equal to the found inversions, plus the remaining number of elements in the array. This would therefore account for all inversions while keeping the same complexity as mergesort.

3.

To reduce this recursion relation into a closed form, we can more easily represent this with a recursion tree as follows:



Each row of the tree add up to a sum of n, as we are evenly splitting each sub tree an equal amount. As we progress further down the recursion tree, we know that the distance of the tree is log(n), and by adding up the total rows found within the tree, we can conclude that the closed form of the recursion relation is O(n log(n)).