CS 3100 / In-class Activity 1, Warmup and Review

Name	Computing ID
Your Name:	

In class: You must work in teams of 2, 3 or 4. Each person writes answers and turns in sheet at end of class..
Missed class? Work alone and answer to the best of your ability. Submit to GradeScope by 9am on the 2nd day after in-class activity.

1. If $f(n) = n^{1.5}(\log n)^2$ and $g(n) = n^2 \log n$, which grows more quickly? (That is, which would be a "worse" time-complexity.) Express your answer in terms of one of the order-classes we've studied, i.e. something in a form like $f(n) = \Theta(g(n))$ but with something other than Big-Theta. Write that in the box below:

Also, think and talk about what definition you'd use to prove your answer mathematically. (You don't have to actually show that work on this sheet!)

2. Consider the following problem. It can be solved using search algorithms you've studied, perhaps using sort algorithms you've studied. In the box below, write the Big-Oh order class (in terms of n, m and k) for a solution that uses sorting. (You can assume worst-case behavior if that helps you think about this.)

A. In a list of n items, you execute m searches (queries), where each query looks for a target value t_i in the list. If it is, we'll store the location p_i where it was found in the list that's searched. Assume $m < n$.
B. At the end of the m queries, some new value n_i replaces each of the target values that were found, i.e. `list[`p_i`]` = n_i for all the t_i targets that were found. (In other words, each successful query causes an update to the list.)
C. The previous steps are repeated k times.

3. In CS2100 you saw lower-bounds proofs about sorting where a logical argument (proof) was used to show that it was impossible for any sorting algorithm to do fewer operations than some lower bound value. (E.g. no comparison sort can do fewer than $\Theta(n \log n)$ comparisons.) Let's do a simpler lower bounds proof!
The problem is: Find the largest value in an unsorted list with no duplicate values. Give a logical argument that no algorithm that compares pairs of list-items can correctly solve this problem in fewer than $n - 1$ comparisons. (*Hints:* A single comparison produces a "winner" and a "loser". What does it mean for an item to be the largest in a list?) *Write your answer here and continue on the top of the next page if you need more space!*

4. Talk among your group to remind yourself how **proof by induction** works in general. Next, we'll use this technique to argue that Quicksort is correct.

We'll let you assume that the partition operation works correctly. Recall that `partition(list,first,last)` returns the location p of an item in the sublist `list[first:last]`, where all items in positions before p are < `list[p]`, and all items after position p are > `list[p]`. The item at location p is the pivot-value, and partition puts it into its correct position but doesn't sort what's before it or after it. After partition is done, Quicksort is called recursively on the sublists before and after the pivot-value.

Part A. Let's help you through this. In induction we start with **the base case**. If n=1 (i.e. there's one item in the list), explain why partition and quicksort produce the correct answer for that list of size 1.

Part B. For the inductive step, we will assume the *inductive hypothesis* that Quicksort works correctly for any list that has size less than n. Now we need to use this assumption to make an argument that, for a list of size n, the algorithm produces a list that's sorted. Assuming that partition works correctly and the inductive hypothesis is true, write an argument below that shows, for a list of size n, the call to partition and the recursive calls to Quicksort correctly sorts that list.