This homework is out of 10 points and graded for completion. Working together on a homework is okay, copying answers is not. Some homeworks will include challenge extra credit problems that will be worth 2 points each.

1. Show that if \( f(n) \) is \( O(g(n)) \) then \( g(n) \) is \( \Omega(f(n)) \).

Because \( f(n) \) is \( O(g(n)) \), we know:

\[
\lim_{n \to \infty} \frac{f(n)}{g(n)} < \infty
\]

\[
\frac{1}{\lim_{n \to \infty} \frac{f(n)}{g(n)} > 0} \quad \text{Taking 1 over both sides}
\]

\[
\lim_{n \to \infty} \frac{1}{\frac{f(n)}{g(n)}} > 0
\]

\[
\lim_{n \to \infty} \frac{g(n)}{f(n)} > 0
\]

2. List several reasons you might choose to use a \( \Theta(n \log n) \) time algorithm over a \( \Theta(n) \) time algorithm for the same task.

- You are only running on small inputs and the \( \Theta(n \log n) \) algorithm is faster on them due to a better constant factor (1000\(n\) vs \(\frac{n \log n}{2}\))
- The \( \Theta(n \log n) \) algorithm uses much less memory and it is running in a memory constrained environment
- The \( \Theta(n \log n) \) algorithm is much simpler to implement and maintain
- The \( \Theta(n \log n) \) algorithm is easier to parallelize
- The \( \Theta(n \log n) \) algorithm is actually faster for the types of inputs you will give it

3. The array based list described in class has a flaw. If a large number of elements are inserted and then all removed, the underlying array will still be large even though there are no elements in the list. This problem can be fixed by copying into a smaller array when the number of elements goes below a threshold. In terms of \( n \) the number of elements currently in the list and \( N \) the size of the underlying array, when should a new smaller array be created and how big should it be?
We should resize when \( n = N/4 \) and we should resize to \( N/2 \). Other correct answers will resize at a point when \( n = \epsilon N \) for some \( \epsilon < 1/2 \), and should resize to something

**Extra credit:** Prove a lower bound on the time of any comparison based search algorithm for sorted lists. That is show that any algorithm that takes as input a sorted list and an element and outputs the index of the element in the list (if it appears) must take a certain number of steps.

We use a similar argument we used for the sorting lower bound. Given a comparison based search algorithm \( A \), run \( A \) on the following inputs:

\[
L_1 = 1, 2, 3, 4, 5, \ldots, n \text{ searching for } 1 \\
L_2 = 1, 2, 3, 4, 5, \ldots, n \text{ searching for } 2 \\
L_3 = 1, 2, 3, 4, 5, \ldots, n \text{ searching for } 3 \\
\vdots \\
L_n = 1, 2, 3, 4, 5, \ldots, n \text{ searching for } n
\]

When \( A \) is run on input \( L_i \), it needs to return index \( i - 1 \). So \( A \) should return a different answer for each input. Therefore if for two inputs \( A \) runs the same code and returns an answer, one of the answers is incorrect meaning \( A \) needs to perform a comparison that differentiates the two answers.

We can now employ the same comparison following strategy from class. We start by following all \( n \) runs of \( A \) on the \( L_i \). Whenever a comparison is performed on the current set we are following, we continue following whichever comparison value was larger (so if more than half of the comparisons were true we follow the runs that returned true). At each comparison the size of the set we are following shrinks by at most a factor of two. Because the algorithm needs to get down to one input in the set we are following to return an answer, it must perform \( \log_2 n \) comparisons. Thus any comparisons based search algorithm takes \( \Omega(\log n) \) time.