1. Rank the following functions from smallest to largest according to their big-O complexity. That is, order them based on their asymptotic growth rate. For example, you could write $n < n^3$ since $n \in O(n^3)$, but $n^3 \notin O(n)$. Some of the functions might be in the same big-O class. You do not need to do any formal proofs.

<table>
<thead>
<tr>
<th>Function</th>
<th>Growth Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n^{100}$</td>
<td>$\lg n$</td>
</tr>
<tr>
<td>$2^n$</td>
<td>$2^{\lg n}$</td>
</tr>
<tr>
<td>$4$</td>
<td>$\sqrt{n}$</td>
</tr>
<tr>
<td>$n!$</td>
<td>$100n$</td>
</tr>
</tbody>
</table>

**Solution:** $4 < \lg n < \sqrt{n} < 2^{\lg n} = 100n < n^{100} < 2^n < n!$

$4$ is a constant and does not grow as a function of $n$. All positive powers of $n$ grow faster than $\lg n$, which comes next. The square root of $n$ is the same as $n^{1/2}$, which is the lowest power of $n$ we see in the list. $2^{\lg n}$ is equal to $n$, which is asymptotically equivalent to $100n$. Both grow faster than $\sqrt{n}$ and slower than $n^{100}$. All of these are polynomial functions of $n$, and grow more slowly than $2^n$, which is exponential. Finally, $n!$ is the fastest-growing function.

2. Let $f(n) = 2n^3 + 4n - 17$. Prove that $f(n) \in O(n^3)$.

**Solution:**

**Proof.** Let $c = 3$ and $n_0 = 2$. We will show that for all $n \geq n_0$, $0 \leq f(n) \leq c \cdot n^3$.

For $n \geq 2$, we have $f(n) = 2n^3 + 4n - 17 \geq 2 \cdot 2^3 + 4 \cdot 2 - 17 = 24 - 17 = 7 \geq 0$, satisfying the first part of the desired inequality. We also have $f(n) = 2n^3 + 4n - 17 \leq 2n^3 + 4n \leq 2n^3 + n^3 = 3n^3 = c \cdot n^3$, satisfying the second part of the inequality.

We have shown that there exist a $c$ (namely $c = 3$) and an $n_0$ (namely $n_0 = 2$) such that for all $n \geq n_0$, $0 \leq f(n) \leq c \cdot n^3$. Hence by definition $f(n) \in O(n^3)$. 

3. Let $f(n) = \frac{1}{3}n \lg n$. Is $f(n) \in O(n)$? Prove or disprove.

**Solution:** The claim is false: $f(n) \notin O(n)$. We will prove this by contradiction.

**Proof.** Assume that $f(n) \in O(n)$. Then by definition there exist a $c, n_0 > 0$ such that for all $n \geq n_0$, $0 \leq f(n) \leq cn$. For the particular values of $c$ and $n_0$ satisfying this inequality, we have

\[
\frac{1}{3}n \lg n \leq cn \\
\frac{1}{3} \lg n \leq c \\
\lg n \leq 3c \\
n \leq 2^{3c}.
\]
At this point we have a contradiction because we have arrived at the conclusion that \( n \) must be upper bounded by \( 2^{3c} \), which is a constant. Clearly this cannot be true for all values of \( n \) that are greater than \( n_0 \), no matter what value we choose for \( n_0 \). Hence our original assumption must be incorrect, and \( f(n) \notin O(n) \).

4. Prove that \( O(n) \subseteq O(n^2) \). [Hint: what you are trying to show here is that for all functions \( f(n) \) that are in the set \( O(n) \), \( f(n) \) must also be in the set \( O(n^2) \).]

**Solution:** We are trying to prove a “for all” claim, so we will choose an arbitrary function \( f(n) \) that is in \( O(n) \) and show that it satisfies the properties for membership in \( O(n^2) \).

Proof. Let \( f(n) \) be an arbitrary function that is in \( O(n) \). All we know about \( f(n) \) is that, by definition, there exist constants \( c, n_0 > 0 \) such that for all \( n \geq n_0 \), \( 0 \leq f(n) \leq cn \). For the particular values of \( c \) and \( n_0 \) that satisfy this inequality, we have that for all \( n \geq n_0 \):

\[
0 \leq f(n) \leq cn
\]

\[
0 \leq f(n) \leq cn^2,
\]

where the second line results from multiplying the right-hand side of the inequality by \( n \). Since \( n \geq n_0 > 0 \) and \( n \) is an integer, we know \( n \geq 1 \) so this cannot decrease the right-hand side. But now we have shown that there exists a pair of constants \( c' \) and \( n'_0 \) such that for all \( n \geq n'_0 \), \( 0 \leq f(n) \leq c' \cdot n^2 \). This is exactly the definition of the class \( O(n^2) \). Hence \( f(n) \in O(n^2) \).

5. In class, we have discussed both an array-based implementation and a Vector-based implementation for the Queue ADT. Some advantages of the Vector implementation include that it’s easy to code up using the methods from the Vector class and that it allows us to use generics. We claimed that a disadvantage is that it’s slower. Your job in this problem is to use asymptotic analysis to justify this claim.

(a) Fill in the following table with the worst-case and best-case asymptotic runtime for both the array-based wraparound Queue implementation (the code is posted on the course web page) and the Vector-based Queue implementation (the code was provided as part of HW2, and the source code for the underlying Vector methods is at the end of this assignment for reference). Give a short (1-sentence) explanation of each entry.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Array: Worst Case</th>
<th>Array: Best Case</th>
<th>Vector: Worst Case</th>
<th>Vector: Best Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>enqueue</td>
<td>( O(n) )</td>
<td>( O(1) )</td>
<td>( O(n) )</td>
<td>( O(1) )*</td>
</tr>
<tr>
<td>dequeue</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
<td>( O(n) )</td>
<td>( O(n) )**</td>
</tr>
<tr>
<td>size</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>isEmpty</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
</tbody>
</table>

**Solution:** For the array-based implementation, unless a resize is needed each of the operations only involves a fixed number of constant-time operations. The size() and isEmpty() operations just involve looking up a field; dequeue() just involves removing the next element from a known location in the array, incrementing a pointer, clearing the array entry, and returning the element; and enqueue() just involves placing the new element into the known next available location in the array.
The exception is if a resize is needed: then we create a new array that is twice as large and copy all
\( n \) elements from the old array into the new array. Each copy takes constant time (it consists of two
array accesses and a variable assignment) and there are \( n \) of them, so this takes total time \( O(n) \). Resizes are only performed during the enqueue() operation, and only if we run out of space. So the worst-case runtime for enqueue() is \( O(n) \), but the best case is still \( O(1) \) (a resize might not be needed).

The analysis for the Vector-based implementation is very similar. The same analysis holds for
size(), isEmpty(), and enqueue(). The difference here is dequeue(): whenever an item is removed
from the Vector, all of the other elements are shifted over one position. There are \( n \) elements, each
“shift” takes constant time, so the total time for the shifting is \( O(n) \). This happens in both the
worst case and the best case.

*Note: The Vector implementation that I gave you copied over the entire array on every insert,
making the best case runtime also \( O(n) \). This is a silly thing to do, and is not what the Vector
class actually does—I revised some of the Vector code to make it easier to read, and in the process
inadvertently changed what it does. Vector actually resizes the array by doubling and only copying
over the elements when it runs out of space. For the best case analysis I accepted either \( O(1) \),
which is what Vector actually does, or \( O(n) \), which is what the provided code does.

**Note: \( O(1) \) is not a correct answer here. Many of you said that the best case runtime for the
Vector-based dequeue is \( O(1) \) because you might be removing the last element. Our Big-O anal-
ysis requires us to consider the asymptotic regime: that is, what happen as \( n \) approaches infinity.
Saying that the runtime is constant for a specific, finite value of \( n \) doesn’t tell us anything about
the asymptotic behavior of the operation.

(b) Use your answers from part (a) to either support or refute the claim that the Vector-based im-
plementation is slower than the array-based implementation.

Solution: The only difference between the runtimes in the array-based implementation and the
Vector-based implementation are for the dequeue operation. In the array-based implementation,
the worst case and the best case runtime for dequeue are both \( O(1) \) (i.e., constant time), but in the
Vector-based implementation, dequeue takes \( O(n) \) (i.e., linear time) in both the worst case and the
best case. A runtime of \( O(1) \) is asymptotically faster than a runtime of \( O(n) \), hence the array-based
implementation is more efficient.