Problem #1 **Name:** Alexander Michelbrink

We can rank the functions by growth rate by looking at their Big-Oh notations…

$6nlog(n)$ = O($nlog(n)$)
2^{100} = O(1)
$log(log(n))$ = O($log(log(n))$)
$log^2(n)$ = O($log^2(n)$)
$2^{log(n)}$ = n = O(n) [by Theorem 1.14 in the textbook]
$3\sqrt{n}$ = O($\sqrt{n}$)
$n^{0.001}$ = O($n^{0.001}$)
$n^2log(n)$ = O($n^2log(n)$)
$5log(n)$ = O($log(n)$)
$5n$ = O(n)
n^3 = O(n^3)
n^2 = O(n^2)
$n!$ = O($n!$)

Although $2^{log(n)}$ and $5n$ are both O(n), $5n$ has a bigger growth rate due to its constant factor.

We can now rank the above functions (from slowest growing to fastest growing) according to "Functions Ordered by Growth Rate" table in the textbook…

1. 2^{100}
2. $log(log(n))$
3. $5log(n)$
4. $log^2(n)$
5. $n^{0.001}$
6. $3\sqrt{n}$
7. $2^{log(n)}$
8. $5n$
9. $6nlog(n)$
10. n^2
11. $n^2log(n)$
12. n^3
13. $n!$

Problem #2

Algorithms **Foo** and **Bar** both calculate and return the value a^n.

The worst-case running time of **Foo** is O(n).

All lines inside and outside of the while-loop only involve 1-2 primitive operations, therefore we can focus on how many while-loop executions occur. In this case, since the iterator variable k increments by one each loop from 0 to n-1, the loop executes n times. Therefore, the worst-case running time is O(n).

The worst-case running time of **Bar** is O($log(n)$).

Like in **Foo**, all lines inside and outside of the while-loop involve 1-2 primitive operations, meaning they all have a running time of O(1). This means we can once again just focus on how many times the while-loop executes.

In this case, how many times the while-loop executes depends on how many times the "if" executes and how many times the "else" executes. The worst-case resulting in the most amount of loops would occur when n starts as odd and when, for every "if" that executes, an "else" must execute as well (every division of an even number results in an odd number).

The amount of times n can be divided by 2 (the amount of "if" executions) is given by the rounded-down value of $log(n)$ (represented by $\lfloor log(n) \rfloor$). So, in the worst case, there would be $\lfloor log(n) \rfloor$ "if" executions and ($\lfloor log(n) \rfloor + 1$) "else" executions, with the "$+1$" coming from the initial odd value of n. Therefore, in the worst-case, the total number of loops would be ($2 \cdot \lfloor \log(n) \rfloor + 1$), giving **Bar** a worst-case runtime of O($log(n)$).

Problem #3

A.) _____

Stack:										
									Final Stack	
	B	→	B	→	B	→	B	→	B	
	A		A		A		A		A	
	T		Z		Z		Z		Z	
	I		A		I		I		I	
					N		N		N	
					L		G		G	
							A		A	← Top of Stack
							R			
							F			

B.) _____

Two stacks can be implemented using one array by dedicating half of the array to one stack and the other half to the second stack. If the array is of size n, elements 0 to $\frac{n}{2} - 1$ will belong to the first stack and elements $\frac{n}{2}$ to $n - 1$ will belong to the second stack. This means that the array would have to have an even number of fixed elements. In this implementation, there would also need to be two top-of-stack variables to keep track of the top of both stacks (tos1 for the first stack and tos2 for the second stack). tos1 would be initialized with the value 0 while tos2 would be initialized with the value $\frac{n}{2}$.

One way to implement the stack operations push, pop, and size would be to include an extra parameter *s* in each that clarifies which stack the operation is being performed on. So, push(*o*) becomes push(*o,s*), pop() becomes pop(*s*), and size() becomes size(*s*), where *s* can be either 1 (for stack 1) or 2 (for stack 2).

The algorithm of each method would almost match their original implementations, except with an if-else statement added that branches based on the value of *s*. For size, if *s* = 1, it would go down the branch that returns $tos1 + 1$, else, it would go down the branch that returns $tos2 + 1$. The push and pop operations would follow similarly, with the top-of-stack variable used, EmptyStackException check, and FullStackException check ($\frac{S.length()}{2} - 1$ for stack one, $S.length() - 1$ for stack two) varying depending on which stack is specified by *s*.

Since the inclusion of these if-else statements doesn't add any loops or include any operations with runtimes greater than O(1), all of the stack operations would continue to run in O(1) time as in their original implementations.

Problem #4

A.) _____

									Final Queue	
Queue:	B	→	T	→	I	→	Z	→	I	← Front
	A		I		Z		A		N	
	T		Z		A		I		L	
	I		A		I		N		G	
					N		L		A	
					L		G		R	
							A		F	← End
							R			
							F			

B.) _____

To reverse a queue, I will use an auxiliary data structure, specifically a stack called holderStack. A stack is useful since the order that objects are popped from the stack must be the reverse order of the way they were pushed into the stack. This is due to stacks being first-in-last-out data structures. Therefore, I can use an algorithm that dequeues the elements from Q and pushes them into a stack before popping them out of the stack (in reverse order) and enqueueing them back into Q.

Pseudocode:

```
function reverseQueue(Q)
    while not Q.isEmpty() do
        holderStack.push(Q.dequeue())
    if holderStack.isEmpty() then
        return Q
    while not holderStack.isEmpty() do
        Q.enqueue(holderStack.pop())
    return Q
```

All stack and queue methods used have a running time of O(1) and all other lines involve only one primitive operation. Therefore, we can focus on how many times both while-loops within the function loop to get the runtime. If n is the amount of elements in queue Q, then both while-loops will execute n times. The first loops until all n elements are dequeued from Q and pushed into the holderStack, and the second loops until all n elements are popped from the holderStack and enqueued into Q. This gives that runtime $T(n) \approx 2n$.

Therefore, the above pseudocode for the reverseQueue function above runs in linear-time O(n).

Problem #5

A.) _____

enqueue has a worst-case running time of O(1).

This fact is true because the method only includes a stack push operation, which runs in time O(1).

dequeue has a worst-case running time of O(n).

The if-statement (lines 7-8), assignment of the out_stack.pop() to return_obj (line 9), and the return statement (line 12), all run in time O(1) since any stack operations used also run in O(1) time. The first while-loop pushes all n elements into the out_stack from the in_stack and therefore executes n times.The second while-loop executes $n-1$ times since one element was popped from the out_stack. Since the stack operations inside of both while loops run in time O(1), both loops are O(n). Because neither while-loop is inside of the other, we can see that the **dequeue** method has a worst-case runtime of O(n).

B.) _____

First, we can find the worst-case running times of **enqueue** and **dequeue**.

- Once again, **enqueue** has a worst-case running time of O(1) since its algorithm remains completely unchanged.
- **dequeue** also keeps the same worst-case running time of O(n). Once again, all stack operations used within loops and if-statements run in time O(1), along with the return statement. The only while-loop executes n times since all n elements in the in_stack are popped and then pushed into the out_stack. Therefore, **dequeue** has a worst-case running time of O(n).

Next, we can use amortized analysis.

Since there are $2n$ **enqueue** operations and n **dequeue** operations, the total running time of a series of these operations in unspecified order is given by…

$$T(n) = (2n \cdot O(1)) + (n \cdot O(n)) = O(2n + n^2) = O(n^2)$$

We can look at the amortized time of this series of operations by finding the average time over the total amount of operations.

$$\frac{T(n)}{2n + n} = \frac{O(n^2)}{3n} = O(n)$$

Therefore…

The worst-case total running time of performing the series of operations is $O(n)$.

Problem #6

According to the table from the textbook…

Operation	Array Implementation Runtime	List Implementation Runtime
atRank	O(1)	O(n)
insertAtRank	O(n)	O(n)
remove	O(n)	O(1)

Array Implementation: $(\ n^2 \cdot O(1)\ ) + (\ 5n \cdot O(n)\ ) + (\ n \cdot O(n)\ )$

= $O(n^2) + O(5n^2) + O(n^2)$

= $O(7n^2)$

= $O(n^2)$ runtime.

List Implementation: $(\ n^2 \cdot O(n)\ ) + (\ 5n \cdot O(n)\ ) + (\ n \cdot O(1)\ )$

= $O(n^3) + O(5n^2) + O(n)$

= $O(n^3 + 5n^2 + n)$

= $O(n^3)$ runtime.

The student should use the Array Implementation of the Sequence ADT. They should choose this option because, when considering the type and number of operations being performed and their respective runtimes, the Array Implementation would run in $O(n^2)$ time and the Linked-List Implementation would run in $O(n^3)$ time. This means, for any n, the Array Implementation would have a faster runtime. The reason for this is that there are n^2 atRank operations (O(1) in the Array Implementation and O(n) in the List Implementation) and only n remove operations (O(n) in the Array Implementation and O(n) in the List Implementation).