1. What does the following code fragment do to the queue q?

```java
Stack<String> stack = new Stack<String>();
while (!q.isEmpty()) {
    stack.push(q.dequeue());
}
while (!stack.isEmpty()) {
    q.enqueue(stack.pop());
}
```

**Solution:** This simply reverses the elements in the queue q. The $i$-th element removed from the queue is the $i$-th element added to the stack. Since stacks are FIFO, the $i$-th element added to the stack when emptying q is the $n - i + 1$-th element removed from the stack when re-populating q. Hence, for each index $1 \leq i \leq n$, we have that the $i$-th element in q before the above code runs becomes the $n - i + 1$-th element in q afterwards. That is, the queue is in reverse order.

2. Explain the difference between analyzing the worst case running time of an operation and amortized analysis.

**Solution:** Certain operations/function calls may have significantly different running times. The worst case running time of an operation is the maximum of these running times. For example, the worst case running time of the "push" operation in the resizing array implementation of the stack is $\Theta(n)$, which occurs when we have to resize the underlying array.

The amortized running time of an operation considers all executions of this operation, then finds an average per-operation cost. That is, the amortized running time of an operation is an average of the running times of all executions of that operation. For example, in the resizing array implementation of the stack, while some "push" operations take $\Theta(n)$ time, most only take $\Theta(1)$ time. Averaging these out yields an amortized $\Theta(1)$ cost per push operation.

3. Prove that, starting from an empty stack, the number of array accesses used by any sequence of operations of length M in the resizing array implementation of a stack is proportional to M.
Solution: We prove via induction that the total number of array accesses is at most 6M.

Base Case: For $M = 1$, the operations push and peek take 1 array access while pop takes 2. In all these cases, the number or array accesses is certainly less than $6 \cdot 1 = 6$.

Induction Hypothesis: Assume that for all $1 \leq k < M$ that any sequence of $k$ operations starting from an empty stack takes at most $6k$ array accesses.

Induction Step: We will let $T(i)$ denote the number of array accesses made by the $i$-th operation. The induction hypothesis can be rephrased as $\sum_{i=1}^{k} T(i) \leq 6k$ for all $1 \leq k < M$. Now, the $M$-th operation is either a push, pop, or peek.

Case 1: If it is a peek, then there is no resizing and the number of array accesses made by this step is $T(M) = 1$. Hence by the I.H.

$$\sum_{i=1}^{M} T(i) = \sum_{i=1}^{M-1} T(i) + T(M) \leq 6(M-1) + 1 < 6M$$

Case 2: The $M$-th operation is a push.

Subcase 1: If there is no resizing, this simply takes 1 array access, so by the I.H.

$$\sum_{i=1}^{M} T(i) \sum_{i=1}^{M-1} T(i) + T(M) \leq 6(M-1) + 1 < 6M$$

Subcase 2: If there is a resizing, then let $n$ be the number of elements in the array at step $M$. If this was the first resizing, then we know that each of the $M - 1$ previous operations took at most 2 array accesses, and moreover that there are at most $M$ elements currently in the array. Hence the resize copied at most $M$ elements and thus $T(M) \leq 1 + M$. We thus have

$$\sum_{i=1}^{M} T(i) = \sum_{i=1}^{M-1} T(i) + T(M) \leq 2(M-1) + 1 + M < 6M$$

If this was not the first resizing, then let $k$ be the most recent resizing operation. We know that the number of elements in the array at operation $k$ was $n/2$. We also know that since the $M$-th operation involved a resizing, $M - k \geq n/2$ since we must’ve pushed at least $n/2$ elements during this time. Now, since all the operations between $k$ and $M$ (of which there are $M - k - 1$) didn’t involve a resizing, we know that each took at most 2 array accesses. We know that $T(M) = 1 + 2n$ since it involved a resizing. By the I.H. the first $k$ operations each took at most 6 array accesses on average. The average number of array accesses for the remaining $M - k$ operations is

$$\frac{2(M - k - 1) + 1 + 2n}{M - k} = \frac{2(M - k) + 2n - 1}{M - k} = \frac{2(M - k) + n - 1}{M - k} \leq 2 + \frac{2n - 1}{n/2} \leq 6$$

Hence, the amortized running time of the last $M - k$ operations is at most 6, and the amortized running time of the first $k$ operations is 6. Thus the amortized running time of all operations is 6 (and the total running time is thus $\sum_{i=1}^{M} T(i) \leq 6M$)

Case 3: The $M$-th operation is a pop.

Subcase 1:
If there is no resizing, this simply takes 2 array accesses, so by the I.H.

\[ \sum_{i=1}^{M} T(i) = \sum_{i=1}^{M-1} T(i) + T(M) \leq 6(M - 1) + 1 < 6M \]

**Subcase 2:** If there is a resizing, then we follow the same methodology as above. If it is the first resizing, then we know that the first M - 1 operations took at most 2 array accesses each, the copying took 2n \leq 2M accesses, and the pop took 2 accesses. The total runtime is thus 2(M - 1) + 2n + 2 \leq 2M - 2 + 2M + 2 = 6M.

If this was not the first resizing, then let k be the most recent resizing operation. If there are n elements currently in the array, then we know there were 2n elements at the time k was performed (since we are downsizing now) and thus M - k \geq n. By the induction hypothesis, the first k operations took 6 array access each on average. The next M - k - 1 operations took at most 2 accesses each since they didn’t involve resizing. The last operation took 2 + 2n operations. The amortized cost of the last M - k operations is thus

\[ \frac{2(M - k - 1) + 2 + 2n}{M - k} \leq \frac{2(M - k) + 2n}{M - k} \leq 2 + \frac{2n}{M - k} \leq 2 + \frac{2n}{n} = 4 \leq 6 \]

Hence the amortized cost over all operations is at most 6. This completes the proof by induction. (Phew!)

4. Suppose we have a resizing array that increases in size by K entries when the array is full, and decreases in size by K entries when the array has K empty entries. Show that the push and pop operations each take amortized M time (where M is the number of operations) for some worst case sequence. Give an example of a worst case sequence. Observe that this results in \(M^2\) time for \(M\) operations.

**Solution:** The first thing to note is that regardless of the value of K, a worst case push or pop operation in the resizing array when it has \(x\) elements will require \(O(x)\) time to create the new array and copy the \(x\) elements over. Given the resizing conditions of the array, we can cause the array to resize with each operation in the following way:

1. Call the size of the current array \(s\). Fill all \(s\) spaces in the array.
2. Push one more element to the array to trigger a resize. The array now has \(s + 1\) elements in an array of size \(s + k\).
3. Pop one element from the array. The array now is of size \(s + k\) and has \(s\) elements, and thus must resize down.
4. There are now \(s\) elements in an array of size \(s\), and the cycle of pushing and popping can continue indefinitely.

To show that there exists a sequence where each push and pop operation takes amortized time proportional to \(M\), we can simply consider the case when the size of the array \(s\) at the start of the sequence defined above is proportional to \(M\). For each of the operations in the above sequence, the operation will take time proportional to \(M\) due to the resizing.
We can make this more concrete with an example of $M$ operations that take time proportional to $M^2$ in total. Start with an empty resizing array and push $\frac{M}{2}$ elements. Push a constant number of elements to the array to make it full. We now have remaining operations left proportional to $M$, each of which we can use in the way defined above to take time proportional to $M$. This will give us a total runtime proportional to $M^2$. 