Practice Problems

1. **[Queue/Stack Programming]** Suppose the only data structure currently implemented is the Stack (i.e., Lists don’t yet exist in our language). Define a Queue\(<E>\) class with \(E\) poll()\), offer(\(E\) item)\), and \(E\) peek() methods, using only Stacks\(<E>\) and its associated methods (push(\(E\) item)\), \(E\) pop()\), and \(E\) peek()). What are the complexities of \(E\) poll()\), offer(\(E\) item)\), and \(E\) peek() using such an implementation? Sol: See code file.

2. **[Queue Programming]** Write a method \(\text{interleave}\) that accepts a Queue of integers as a parameter and rearranges the elements by alternating the elements from the first half of the queue with those from the second half of the queue. For example, suppose a variable \(q\) stores the following sequence of values: \([4,5,6,8,9,10,11]\) (with 4 at the head of the queue). Then \(\text{interleave}(q)\) should return \([4,8,5,9,6,10,7,11]\). For full credit, you are only allowed to use one temporary queue in your solution, and you cannot use any methods besides those provided by java’s Queue interface. You should throw an IllegalArgumentException if the size of the given queue is odd. Sol: See code file.

3. **[List Complexity]** Consider the following two methods that reverses the contents in a List. Answer the following questions. (If this question appeared on the exam, I would not provide you with the SinglyLinkedList code).

```java
public void reverse1() {
    for (int i = 0; i < size(); i++) {
        add(i, remove(size()-1));
    }
}

public void reverse2() {
    for (int i = 0; i < size(); i++) {
        add(size()-1-i, remove(0));
    }
}
```

(a) If the class in which these methods were implemented is a SinglyLinkedList, which of the two methods would you prefer to run if iterator (location) caching and tail reference are both enabled? Sol: You should understand that both algorithms run in theoretical \(O(n^2)\) time. With that said, however, you would prefer reverse2 in practice. This is because \(\text{remove}(0)\) takes constant time, so you are only eating the cost of the \(\text{add()}\), which adds to the list in reverse order (so iterator caching doesn’t help). Therefore, the amount of steps done in iteration 1 of the outer loop is \(n-1\), followed by \(n-2\) in iteration 2, and so on, resulting in \(T(n) = (n-1) + (n-2) + \ldots + 2 + 1 = n(n-1)/2\) total steps.
Compare this result to reverse1(). The remove(size()-1) call always requires $n - 1$ steps to find the node that comes before the tail node (because you need to set its next pointer to null, and because you need to set the list’s tail to its). The iterator caching does however allow the add(i, ..) call to be constant time. Therefore, the number of steps performed in each iteration of the outer loop is $n - 1$, resulting in $T(n) = n(n - 1)$ total steps.

(b) Does your answer to the previous question change if the list was instead doubly linked, with iterator (location) caching and tail reference are both enabled? **Sol:** A doubly linked list makes a significant difference.

* For reverse1(), the complexity reduces because you no longer need an $O(n)$ time lookup to identify the node before the tail node. You can just follow the previous reference, which is an $O(1)$ operation.
* For reverse2(), the complexity also reduces because the iterator can now move in reverse, and therefore adding to the previous node can be done in $O(1)$ time.

You would expect that both algorithms now run in $T(n) = n$ steps due to the outer loop, which is $O(n)$.

(c) Would you prefer one over the other if the list was an ArrayList? **Sol:** Both algorithms would have the same theoretical complexity of $O(n^2)$ if run over an ArrayList. However, you would prefer reverse1() over reverse2() in practice because it would require fewer total steps.
4. **[Recursive Tracing]** Consider the following recursive method.

```java
public static int mystery(int[] list, int head) {
    if (head == list.length - 1) {
        if (list[head] % 2 == 0) {
            return 1;
        } else {
            return 0;
        }
    } else if (list[head] % 2 == 0) {
        return 1 + mystery(list, head+1);
    } else {
        return 0 + mystery(list, head+1);
    }
}
```

In a couple sentences, describe what this method does. What is the time complexity of this algorithm? Show your work. **Sol:** This method counts the number of odds in the given list starting from the head position. It is assumed that head is input as 0.

**Complexity analysis:** Let the problem size \( n \) denote \( \text{list.length} - 1 - \text{head} \).

\[
T(n) = \begin{cases} 
1, & n = 0 \\
\k + T(n-k), & n > 0 
\end{cases}
\]

where \( k > 0 \). We are interested in \( T(n) \) when \( n - k = 0 \) (i.e., reaches the base case). So if we let \( n - k = 0 \), then \( k = n \), giving us: \( T(n) = n + T(0) = n + 1 \). Therefore, \( T(n) = O(n) \).

5. **[Recursive Programming]** The exponentiation of \( a \) to the power of \( b \) can be expressed as follows:

\[ a^b = a \times a \times \ldots \times a \]  
\[ \text{b times} \]

Provide a recursive method, \( \text{exp()} \), that inputs two integers, \( a \) and \( b \), and returns the value of \( a^b \). You may assume that \( b \geq 0 \). Recall from algebra that when \( b = 0 \), then \( a^b = 1 \). What is the time complexity of this algorithm? Show your work. **Sol:** See code.

6. **[Recursive Programming]** Write a recursive method \( \text{repeat} \) that accepts a string \( \text{str} \) and an integer \( n \) as parameters and that returns a string consisting of \( n \) copies of \( \text{str} \). If \( n \) is negative, throw an IllegalArgumentException. For example:

- \( \text{repeat("foo ", 3)} \) returns "foo foo foo 
- \( \text{repeat("i love CS", 1)} \) returns "i love CS"
- \( \text{repeat("i love CS", 0)} \) returns "

**Sol:** See code.

7. **[Recursive Programming]** Define the method \( \text{buildOddStack} \) that creates and returns a stack full of odd numbers. It should take a single argument, the largest odd number to be placed in the stack,
and return a stack containing 1 through that value, with 1 at the bottom of the stack. For example
buildOddStack(11) would return a stack containing [1,3,5,7,9,11]. For full credit, define the method
recursively, and don’t use methods other than those defined in the Stack class. Sol: See code.

8. **[Recursive Programming]** Do the previous problem again, except that the method should return a
stack containing the numbers in reverse order. For example buildOddStack(11) would return a stack
containing [11,9,7,5,3,1]. For full credit, define the method recursively, and don’t use methods other
than those defined in the Stack class. Sol: See code.