Link To GitHub repo:
https://github.com/Connor-Cash532/CSC3130-Assignment4-Connor_Cash

Problem 1-Stacking

```
Stack<Integer> s = new Stack<>();
s.push(8)
s = [8]
s.push(2)
s = [8, 2]
s.pop
s = [8]
s.push(s.pop()*2) //s.pop() == 8 so s.push(16)
s = [16]
s.push(10)
s = [16, 10]
s.push(pop()/2) //s.pop() == 10 so s.push(5)
s = [16, 5]
```

Problem 2-Queueing

```
Queue<Integer> q = new Queue<>();
q.push(4)
q = [4]
q.push(q.pop()+4) //q.pop() == 4 so q.push(8)
q = [8]
q.push(8)
q = [8, 8]
q.push(q.pop()/2) //q.pop() == 8 so q.push(4)
q = [8, 4]
q.pop()
q = [4]
q.pop()
q = []
```

Problem 3-Find In Deque

```
class findInDeque<T>{
   public int findEInDeque(T y, Deque<T> c){
       if(c.isEmpty())
           return -1;
       int size = c.size()/2;
       for(int i = 0; i < size; i++){
           if(c.peekFirst().equals(y)){
               return i;
           }
           if(c.peekLast().equals(y))
               return ((size*2)-i)-1;
           c.removeFirst();
```

```
            c.removeLast();
        }

        return -1;
    }
}
```

Problem 7-Algorithm Analysis

Analysis of isBalanced

```
Time Complexity
n = s.length()

O(1) + O(n/2) + O(1) + O(n/2) + O(1) + O(n/2) + O(n/2) + O(1)
O(2n)=O(n)

Space complexity
O(n/2)=O(n)
```

The time complexity is O(n) since there are 4 total loops in the algorithm that all run n/2 times. This simplifies to O(2n) where the constant can be ignored and simplified to O(n) time complexity.

The space complexity is O(n) since the only space that is being allocated that changes as the input changes is the stack which will have at most n/2 values in it. This is O(½*n) space complexity which simplifies down to O(n).

Analysis of decodeString

```
Time Complexity
n = s.length()
k = max # of letters enclosed in a single bracket in the encoded string ex
"3[a]2[bcef]" k=4
l = max # of digits in string ex "23[a]2[bc]", l=2
p = max number that is being multiplied in the string ex. "3[a]2[bc]" p=3
m = max length of string that is multiplied ex. "3[a2[c]]2[ef]"
m="accaccacc".length()==9
c = length of decoded string
Note I use the maxes here since the loops that run through the temporary
variables that hold the
string that is multiplied, the multiplied string, and the number the string is
multiplied with will
```

```
at most runs these max times in a given run of the outer loop

O(1)+O(1)+O(1)+O(1)+O(1)+O(1)+O(1)+O(n(k+l+p+m) + O(c)
Which simplifies to O(nk+nl+np+nm) + O(c) = O(nk+nl+np+nm+c)

Space Complexity
No new variables are initialized in the loop, so the
space complexity would be
O(c) + O(7) = O(c), which represents how the stack is the part of memory that
changes as input changes.
```

The time complexity is O(nk+nl+np+nm+c) because the nested loops run at most k,l,p, and m times which is not the tightest bound, but is true for any given input making it an upper bound. Then k, l, p, and m are multiplied by n since they are nested by an outer loop running n times which makes the inner loops run n times. The algorithm adds all parts of the decoded string into the stack then loops through the length of the decoded string concatenating it to an empty string. This loop runs O(c) times, so the total time complexity is O(nk+nl+np+nm+c). The space complexity of the algorithm is O(c) since the algorithm only initializes variables that take up constant memory and initializes a stack. C is the maximum size of the stack, so as the input changes the algorithms spaces grows at O(c).

Analysis of infixToPostfix

```
Time Complexity
n = expression.length()
m = # of operators and parentheses in expression
O(1) + O(1) + O(n(m+m)) + O(m)=O(2+n*2m+m)=O(2n*m+m)=O(n*m+m)

Space Complexity
The only space that changes as the input changes is the stack which will
contain at most
m characters
O(m)
```

The time complexity of the infixToPostfix algorithm is O(n*m+m). The nested loops add all parentheses and operators to the stack exactly once and pop each parentheses and operators exactly once, so they will never run more than m times. This makes the total time complexity of the first loop body is O(n(2m)) since for each time the outer loop runs the inner loops run at most m times. Then all other operators that were not nested in parentheses are popped from the stack and added to the postfix string. This is also bounded by m, since the stack's size will never be greater than m for all input values. Then the total time complexity is O(n(2m)+m)=O(2m*n+m)=O(m*n+m).
The space complexity of the algorithm is O(m) since the stack is the only variable or data structure that changes its memory allocation as the input changes. It is specifically O(m) since only parentheses and operators are added to the stack so the maximum size the stack can be is m.