Balanced Brackets

```
import java.util.Stack;
public class BalancedBrackets {

   public static String isBalanced(String brackets) {
       // O(1) time and Space
       if(brackets.length() == 0) {
           return "NO";
       }

       // O(1) time and Space
       Stack<Character> bracketStack = new Stack<>(); //this line is O(1) time and space

       int listSize = 0; // O(1) Time

       // O(N) time where N is the number of characters in the brackets String because each
loop
       // iterations results in a constant number of operations being performed and
       // N loop iterations are performed
       for(int i = 0; i < brackets.length(); i++) {
           char bracket = brackets.charAt(i); // O(1) space (allocates every loop iteration)
           if(bracket == '{' || bracket == '[' || bracket == '(') {
               // O(1) time
               bracketStack.push(bracket);
               listSize++;
           } else if(listSize == 0) {
               return "NO"; // no open brackets indicating it can never be balanced
           } else if (bracket == '}' && bracketStack.peek() == '{') {
              // O(1) time
               bracketStack.pop();
               listSize--;
           } else if (bracket == ']' && bracketStack.peek() == '[') {
               // O(1) time
               bracketStack.pop();
               listSize--;
           } else if (bracket == ')' && bracketStack.peek() == '(') {
               // O(1) time
               bracketStack.pop();
               listSize--;
           } else {
               // O(1) time
               return "NO"; // if a non balanced bracket or non bracket character appears
           }
       }

       // Because we have to push brackets into the stack, which requires an increasing amount
       // of memory, which results in O(M)space complexity where M is the number of open
       // brackets.

       // O(1) time and space
       if (listSize == 0) {
           return "YES";
       } else {
           return "NO";
       }
   }
}
```

***Time Complexity*: O(N)** where N is the number of characters in brackets
***Space Complexity*: O(M)** where M is the number of open brackets in brackets

DNA to RNA

```
import java.util.Stack;

public class DNAtoRNA {
   public static String dnaToRna(String dna) {
       // O(1) time and space
       String rna = "";
       char base;

       // O(N^2) time, where N is the number of characters in the dna String, because
       // each string concatenation operation must copy the previous string character by
       // character which is O(N) time and the loop iterates N times.

       // O(N) space, where N is the number of characters in the dna String, because
       // the resulting RNA sequence will have a length of N after the outer loop
       // finishes. However, there are not N Strings being stored in memory, just one String
       // of length N, being continuously rewritten.
       for(int i = 0; i < dna.length(); i++){
           base = dna.charAt(i); // this line is O(1) time and space
           if(base == 'T') {
               rna = rna + "U"; // this line is O(N) and space
           } else {
               rna = rna + base; // this line is O(N) and space
           }
       }


       // O(1) time and space
       return rna;
   }
}
```

***Time Complexity*: O(N^2)** where N is the number of characters in the dna String
***Space Complexity*: O(N)** where N is the number of characters in the dna String

```java
import java.util.LinkedList;
import java.util.Queue;
import java.util.HashMap;

public class RnaToProteins {
   public static String rnaToProteins(String rna) {
       // O(1) time and space
       HashMap<String, Character> lookUpTable = createLookUpTable();
       Queue<Character> protQueue = new LinkedList<>();
       Queue<Character> rnaQueue = new LinkedList<>();

       // O(N) time because the loop iterates N times where N is the number of characters
       // in the rna string, and inside the loop O(1) operations occur.
       // O(N) space where N is the number of characters in the rna string. Worst case, if
       // rna.length % 3 == 0, then the size of the rnaQueue will be of size N, and the size
of
       // protQueue is N/3.

       for(int i = 0; i < rna.length(); i += 3) {
          // O(1) time because add() is O(1) and the loop always iterates 3 times
           if(i+3 > rna.length()) {
               protQueue.add('.');
               break;
           }
           for(int j = 0; j < 3; j++) {
              // O(1) time because add() is O(1) and the loop always iterates 3 times
              // O(1) space 3 elements are always added to the queue
               rnaQueue.add(rna.charAt(i + j));
           }
           char protein = protLookUpTable(rnaQueue, lookUpTable);
           protQueue.add(protein);

       }

       // O(N^2) time, where N is the number of characters in the rna String, because
       // each string concatenation operation must copy the previous string character by
       // character which is O(N) time. The loop iterates equal to the number of elements in
the
       // protQueue, but the size of the protQueue in the worst case would be N/3.

       // O(N) space, where N is the number of characters in the rna String, because
       // the resulting result String will have a length of N after the outer loop
       // finishes. However, there are not N Strings being stored in memory, just one String
       // of length N, being continuously rewritten.
       String protSeq = "";
       int len = protQueue.size();
       for(int i = 0; i < len; i++) {
           protSeq += protQueue.remove();
       }

       return protSeq;
   }

   private static char protLookUpTable(Queue<Character> rnaQueue, HashMap<String, Character>
lookUpTable) {
       // O(1) time and space, 3 loop iterations
       String rnaSeq = "";
       for(int i = 0; i < 3; i++) {
           rnaSeq += rnaQueue.remove();
       }

       // Retrieving from a hashmap has an average time complexity of O(1), but
```

```
        // there are only 64 key-value pairs in the hashmap, so worst case, given 63
collisions,
        // no more than 64 elements must be searched, so O(1) time and space.
        char protein = lookUpTable.get(rnaSeq);
        return protein;
    }
private static HashMap<String, Character> createLookUpTable() {
    // https://en.wikipedia.org/wiki/DNA_and_RNA_codon_tables
    // O(1) time and space, hashmaps always has 64 elements pushed to it irrelevant of any
input
    HashMap<String, Character> LookUpTable = new HashMap<>();
    LookUpTable.put("UUU", 'F');
    LookUpTable.put("UUC", 'F');
    LookUpTable.put("UUA", 'L');
    LookUpTable.put("UUG", 'L');
    LookUpTable.put("CUU", 'L');
    LookUpTable.put("CUC", 'L');
    LookUpTable.put("CUA", 'L');
    LookUpTable.put("CUG", 'L');
    LookUpTable.put("AUU", 'I');
    LookUpTable.put("AUC", 'I');
    LookUpTable.put("AUA", 'I');
    LookUpTable.put("AUG", 'M');
    LookUpTable.put("GUU", 'V');
    LookUpTable.put("GUC", 'V');
    LookUpTable.put("GUA", 'V');
    LookUpTable.put("GUG", 'V');
    LookUpTable.put("UCU", 'S');
    LookUpTable.put("UCC", 'S');
    LookUpTable.put("UCA", 'S');
    LookUpTable.put("UCG", 'S');
    LookUpTable.put("CCU", 'P');
    LookUpTable.put("CCC", 'P');
    LookUpTable.put("CCA", 'P');
    LookUpTable.put("CCG", 'P');
    LookUpTable.put("ACU", 'T');
    LookUpTable.put("ACC", 'T');
    LookUpTable.put("ACA", 'T');
    LookUpTable.put("ACG", 'T');
    LookUpTable.put("GCU", 'A');
    LookUpTable.put("GCC", 'A');
    LookUpTable.put("GCA", 'A');
    LookUpTable.put("GCG", 'A');
    LookUpTable.put("UAU", 'Y');
    LookUpTable.put("UAC", 'Y');
    LookUpTable.put("UAA", '.');
    LookUpTable.put("UAG", '.');
    LookUpTable.put("CAU", 'H');
    LookUpTable.put("CAC", 'H');
    LookUpTable.put("CAA", 'Q');
    LookUpTable.put("CAG", 'Q');
    LookUpTable.put("AAU", 'N');
    LookUpTable.put("AAC", 'N');
    LookUpTable.put("AAA", 'K');
    LookUpTable.put("AAG", 'K');
    LookUpTable.put("GAU", 'D');
    LookUpTable.put("GAC", 'D');
    LookUpTable.put("GAA", 'E');
    LookUpTable.put("GAG", 'E');
    LookUpTable.put("UGU", 'C');
    LookUpTable.put("UGC", 'C');
    LookUpTable.put("UGA", '.');
```

```
    LookUpTable.put("UGG", 'W');
    LookUpTable.put("CGU", 'R');
    LookUpTable.put("CGC", 'R');
    LookUpTable.put("CGA", 'R');
    LookUpTable.put("CGG", 'R');
    LookUpTable.put("AGU", 'S');
    LookUpTable.put("AGC", 'S');
    LookUpTable.put("AGA", 'R');
    LookUpTable.put("AGG", 'R');
    LookUpTable.put("GGU", 'G');
    LookUpTable.put("GGC", 'G');
    LookUpTable.put("GGA", 'G');
    LookUpTable.put("GGG", 'G');

    return LookUpTable;
}
```

***Time Complexity*: O(N^2)** where N is the number of characters in the rna String
***Space Complexity*: O(N)** where N is the number of characters in the rna String

Infix to Postfix

```
import java.util.Stack;

public class InfixToPostfix {

   public static String infixToPostFix(String infix) {
       // O(1) time and space
       Stack<Character> operationStack = new Stack<>();
       Stack<Character> postFix = new Stack<>();
       char character;

       // Loop iterates N times where N is the number of characters in the infix String
       for(int i = 0; i < infix.length(); i++) {
           character = infix.charAt(i);
           if(character == '+' || character == '-' || character == '*' || character == '/' ||
character == '(' || character ==')' || character =='^')  {
               // Check whether to push operation to result or add it to operation stack
               // Either O(M) or O(P) time which either is greater as defined below, and O(N)
               // time where N is the number of characters in the infix String
               updateStack(operationStack, character, postFix); // O(1)
           } else {
               postFix.push(character);
           }
       }

      // Loop iterates R times where R is the elements in the operationStack after the first
loop
      // Worst case R=N, if the infix string is all the characters + - ^ * / ( )
       // Add any operations that are not in the result already from highest precedence to
lowest
       while(!operationStack.isEmpty()) {
           postFix.push(operationStack.pop());
       }

       // O(N^2) time, where N is the number of characters in the infix String, because
       // each string concatenation operation must copy the previous string character by
       // character which is O(N) time. The loop iterates equal to the number of elements in
the
       // postFix stack, but the size of the postFix stack in the worst case would be N.

       // O(N) space, where N is the number of characters in the infix String, because
       // the resulting result String will have a length of N after the outer loop
       // finishes. However, there are not N Strings being stored in memory, just one String
       // of length N, being continuously rewritten.
       // Turn the stack to a string
       String result = "";
       for(char output : postFix) {
           result += output;
       }
       return result;
   }

   private static int quantifyOperation(char operation) {
       // O(1) time and space
       if(operation == '+' || operation == '-' ) {
           return 1;
       } else if(operation == '*' || operation == '/') {
           return 2;
       } else if(operation =='^') {
           return 3;
       } else {
```

```
            return 0;
        }
    }

    private static void updateStack(Stack<Character> operationStack, char operation,
Stack<Character> postFix) {
        if(operation == '('){
            // O(1) time and space
            operationStack.push(operation);
            return;
        } else if(operation == ')') {
          // O(M) time where M is the number of characters between the open and closed
          // parentheses
            while(!operationStack.isEmpty() && operationStack.peek() != '(') {
                postFix.push(operationStack.pop());
            }

            // O(1) time and space
            if(operationStack.peek() == '(') {
                operationStack.pop();
            }
            return;
        }
        // O(1) time and space
        if(operationStack.isEmpty()) {
            operationStack.push(operation);
        // O(1) time and space
        } else if(operationStack.peek() == '(') {
            operationStack.push(operation);
        // O(1) time and space
        } else if(quantifyOperation(operationStack.peek()) < quantifyOperation(operation)) {
            operationStack.push(operation);
        } else {
           // O(P) time where P is the number of operations in the stack that have a greater
           // than or equal to precedence as defined in quantifyOperation() and are not
           // succeeded with an open parenthesis.
            char currentChar = operation;
            while(!operationStack.isEmpty() && quantifyOperation(currentChar) <=
operationStack.peek() && currentChar != '(') {
                postFix.push(operationStack.pop());

                currentChar = !operationStack.isEmpty() ? operationStack.peek(): '_';
            }
            operationStack.push(operation);
        }
    }
        // O(N) space complexity where N is the number of characters in the infix String,
because
        // the total amount of elements in the operation stack and the postFix stack
        // will never exceed the number of characters in the infix String.

}
```

***Time Complexity*: O(N^2)** where N is the number of characters in the infix String
***Space Complexity*: O(N)** where N is the number of characters in the infix String