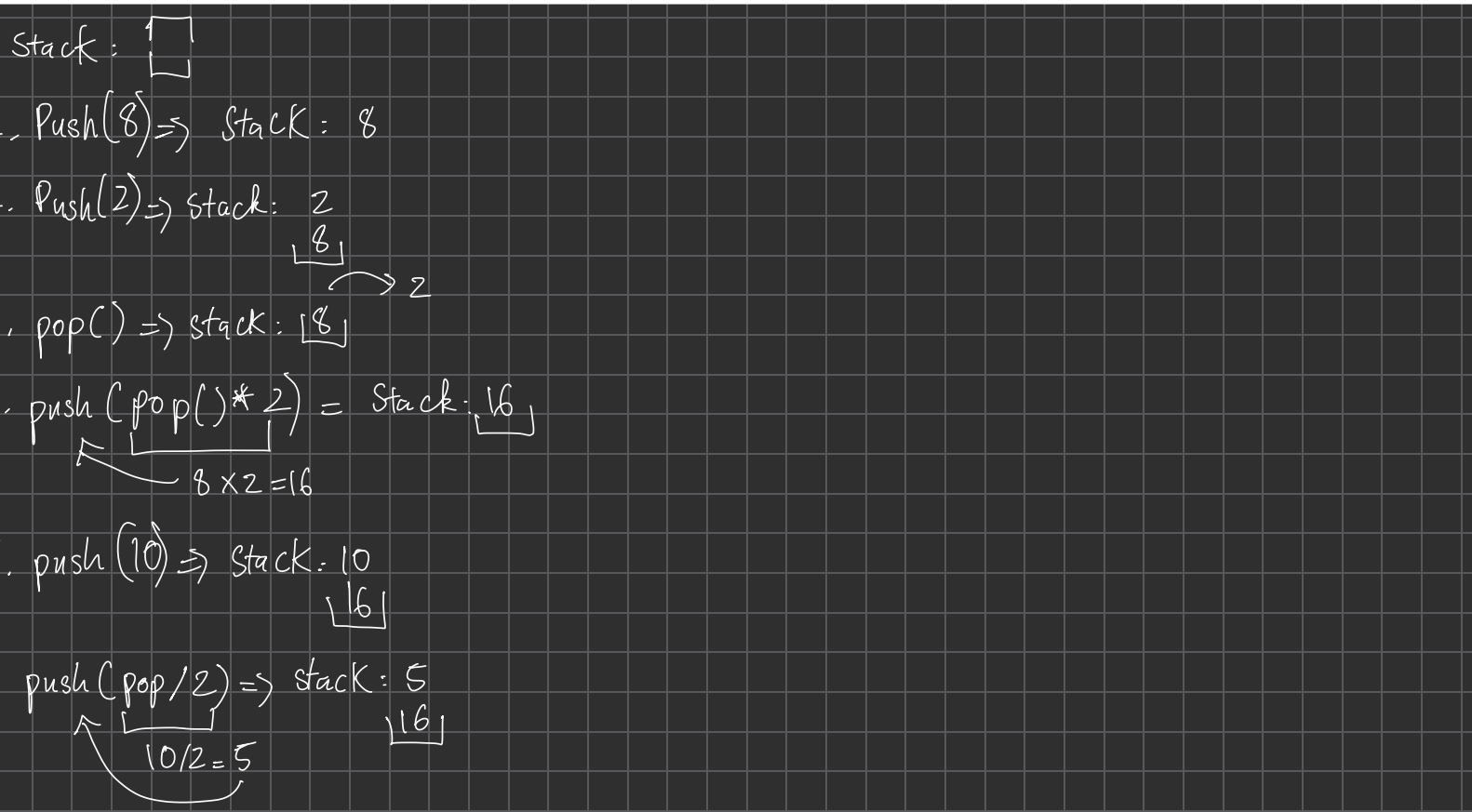
Problems:

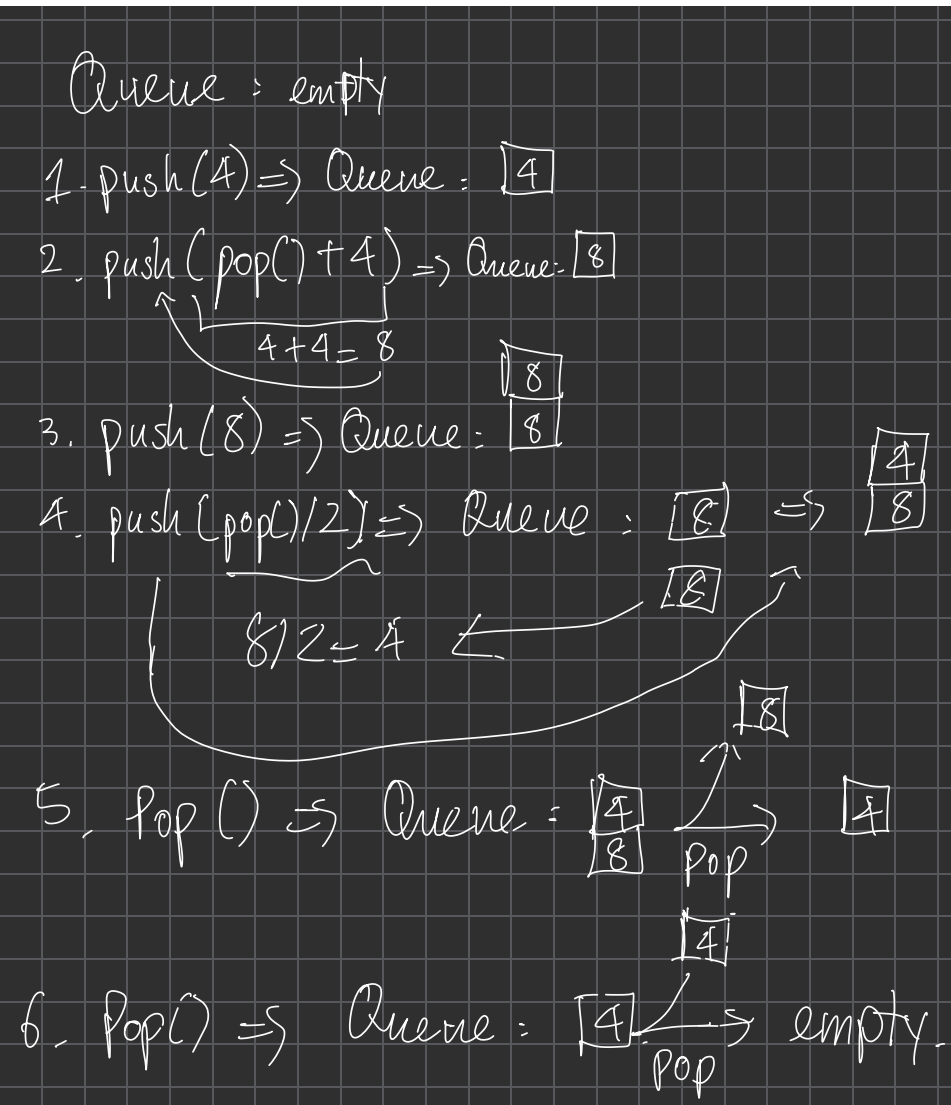
1 (text) Stacking [10 points] Given an empty stack, what will be the contents of the stack after the following operations? (Hint: When you have a method inside another method call (e.g., this(that()), that() will be executed first, afterwards this() will be executed)

- push(8)
- push(2)
- pop()
- push(pop()*2)
- push(10)
- push(pop()/2)



2 (text) Queueing [10 points] Given an empty queue, what will be the contents of the stack after the following operations? (Hint: When you have a method inside another method call (e.g., this(that()), that() will be executed first, afterwards this() will be executed)

- push(4)
- push(pop()+4)
- push(8)
- push(pop()/2)
- pop()
- pop()



3 (text) Find in deque [10 points] We discussed that using a doubly-linked list, you are able to search an element in $O(\frac{n}{2})$, the same is true for a deque.
Given a Deque q and an element y, provide an algorithm that finds the position in the deque in which element x is stored in $O(\frac{n}{2})$.

Hint: The i-th element from the right it a position i, whereas the i-th position from the left it a position n-i

1. Create a method that take a deque and a target element T.

2. We create a deque call q and store the total of elements from the deque to a variable n.
int n = q.size()

3. We then create leftIndex = 0 (first element) and rightIndex = n - 1 (Last element).

4. Create a forward iterator to move from the front to the back, and another iterator that goes backward to move from back to the front.

5. Use a while loop (leftIndex ≤ rightIndex) to compare elements at both ends simultaneously.
- If the left most element matches T, result its position (left index).
- If the rightmost element matches T, result its position (right index).
- If neither match, move the left index forword (+1) and right index backward (-1)
- If the left and right indices cross each other, meaning the element is not in the deque

In conclusion, it traverses front and back simultaneously to search. Since it searches both ends simultaneouly, it reduces the number of comparison by half. (Left doesn't go to n-1, Right doesn't go to index 0. Which makes this O(n/2) or simplify to O(n).

7 (text) Algorithm Analysis [10 points]
For each of the algorithms you wrote for problems 4-6, explain their time complexity and space complexity using Big-O notation. Explain how you arrived at your answer.

```java
public class balancedBracket {  2 usages  Peter
    public boolean isBalanced(String s) {  1 usage  Peter
        Stack<Character> stack = new Stack<>();
        for (char c : s.toCharArray()) {
            if (c == '(' || c == '[' || c == '{') {
                stack.push(c); // Push opening brackets
            }
            else {
                if(stack.isEmpty()){
                    return false;
                } else if ((c == ')' && stack.pop() != '(')) {
                    return false;
                } else if ((c == ']' && stack.pop() != '[')) {
                    return false;
                } else if ((c == '}' && stack.pop() != '{')) {
                    return false;
                }
            }
        }
        return stack.isEmpty();
    }
}
```

#4.

Time Complexity: O(n) where n is the length of string s as the input. This is because I'm traversing the whole string and look at each characters one by one.

Space Complexity: O(n) where n is the length of the string. This is because in the worst case, we would push n elements in the stack.

```java
public String decodeString(String s){  3 usages  Peter
    Stack<Integer> stackNum = new Stack<>();
    Stack<String> stackString = new Stack<>();
    int Num = 0;
    String tempString = "";

    for (char c : s.toCharArray()){
        if(Character.isDigit(c)){
            Num = Num * 10 + (c - '0');
        } else if (c == '['){
            stackString.push(tempString);
            stackNum.push(Num);
            tempString = "";
            Num = 0;
        } else if (c == ']') {
            int mulCount = stackNum.pop();
            String prevString = stackString.pop();
            String result = "";
            for (int i = 0; i < mulCount; i++) {
                result += tempString;
            }
            tempString = prevString + result;
        }else {
            tempString += c;
        }
    }
    return tempString;
}
```

#5.

Time Complexity: $O(n^2)$

Because the first for loop run n-times where n is the length of string s. Inside, there's a inner loop run K times where k is number of repetitions after every ']'.

We have 2 cases where n = K and n > K

$n = K \Rightarrow O(n) \cdot O(n) \Rightarrow O(n^2)$ $\quad n > K = O(n) \cdot O(K)$.

Space: O(n) (n length of the string input.) because stackNum stores numbers O(n) stackstring stores substrings at most O(n) space

Variables like tempString, PrevString, result being created n-times and reuse inside the second for loop. We can say it has O(2n) but Chop off constant $\Rightarrow$ O(n).

6.

```java
public static int getPrecedence(char op){ 2 usages  Peter
    if(op == '+' || op == '-'){
        return 1;
    }else if(op == '*' || op == '/'){
        return 2;
    }else if(op == '^'){
        return 3;
    }
    return 0;
}
```

O(1) atomics

```java
public String infix2Postfix(String infix){ 2 usages  Peter
    StringBuilder result = new StringBuilder();
    Stack<Character> stack = new Stack<>();

    for (char c : infix.toCharArray()) {
        if (Character.isLetterOrDigit(c)) {
            result.append(c);
        } else if (c == '(') {
            stack.push(c);
        } else if (c == ')') {
            while (!stack.isEmpty() && stack.peek() != '(') {
                result.append(stack.pop());
            }
            stack.pop(); // Remove '('
        } else {
            while (!stack.isEmpty() && getPrecedence(c) <= getPrecedence(stack.peek())) {
                result.append(stack.pop());
            }
            stack.push(c);
        }
    }
    while (!stack.isEmpty()) {
        result.append(stack.pop());
    }
    return result.toString();
}
```

O(n)
O(1)
O(1)
O(1)
Atomics

getPrecedence is O(1) because

it only has comparisons and return which is atomic operation. => O(1).
The time execute the code won't grow as the input grows.

Time Complexity.

O(n), where n is the length of the string. This is because we only traverse through the input string once. processing each character at O(1)
Stack operations (pop, push) is O(1).

Space: O(n), where n is the length of the string input.
This is because even though we may not store everything in the stack but in the worst case, the stack will hold n elements in memory.