

```

public static <E> boolean unique(List<E> list) {
    List<E> checkItems = new ArrayList<E>();
    for (int i = 0; i < list.size(); i++) {
        // add the key to the checkItems if the key isn't already there
        if (!checkItems.contains(list.get(i))) {
            checkItems.add(list.get(i));
        } else { // if the value is already there, they are not unique
            return false;
        }
    }
    // if the item is never repeated, all values are unique
    return true;
}

```

The unique function has time complexity **$O(n^2)$** because in the worst case scenario, the for loop operates on every value in the List that gets passed in, and iterates through the list again to check if it contains the current value. The for loop executes for every item in the list. Within the for loop, the contains method iterates through the list created within the function to check if the item has already been added. In the worst case scenario, the list created within the function would have the same length as the passed in list. Every item needs to be checked against every other item. This results in a runtime of $O(n^2)$.

the time complexity is $O(n^2)$.

```

public static List<Integer> allMultiples(List<Integer> list, int num) {
    List<Integer> multiples = new ArrayList<>();
    for (int i = 0; i < list.size(); i++) {
        // add the numbers that are divisible to the return list
        if (list.get(i) % num == 0) {
            multiples.add(list.get(i));
        }
    }
    return multiples;
}

```

The allMultiples function has time complexity **O(n)** because the for loop operates on every value in the list that gets passed in. Because the for loop operates on every item in the list and the time it takes to execute the function changes linearly with the number of items in the list, the time complexity is O(n).

```
public static List<String> allStringsOfSize(List<String> list, int length) {
    List<String> listOfStringsOfSpecLength = new ArrayList<String>();
    // go through the list and filter the words that have the length
    for (int i = 0; i < list.size(); i++) {
        // if the word is of the right length, add it to the return list
        if (list.get(i).length() == length) {
            listOfStringsOfSpecLength.add(list.get(i));
        }
    }
    return listOfStringsOfSpecLength;
}
```

The time complexity of the allStringsOfSize is **O(n)** because the for loop operates on every value in the list that gets passed in. The time it takes to complete the function changes linearly with the number of items in the list and every item gets operated on, so the time complexity is O(n).

```
public static <E> boolean isPermutation(List<E> listA, List<E> listB) {
    // if sizes are not the same, false
    if (listA.size() != listB.size()) {
        return false;
    }
    // for each item in A, count each time each item occurs
    // this is fine b/c the lists are the same length
    for (E item : listA) {
        int countA = 0;
        int countB = 0;
        for (E itemInA : listA) {
            // count the number of times item in A is equal to the item
            if (item == itemInA || item.equals(itemInA)) {
                countA++;
            }
        }
    }
}
```

```

        }
    }
    for (E itemInB : listB) {
        // count the number of times item in B is equal to the item
        if (item == itemInB || item.equals(itemInB)) {
            countB++;
        }
    }
    // if the count in each is every different, it is false
    if (countA != countB) {
        return false;
    }
}
// if the two lists pass, they are permutations of each other
return true;
}

```

The time complexity of the isPermutation function is $O(n^2)$. This is because in the worst case scenario, the two lists would be permutations of each other. Within the outer for loop, there are two for loops which do similar things. The inner for loops iterate through the length of the passed in lists, which would result in each having $O(n)$. Together, they would have $O(2n)$ which simplifies to $O(n)$. These for loops are within another for loop which also executes for every item in the list. The outer for loop would cause the inner loops to have to execute for the entire length of the list for every iteration of the outer for loop, which also executes for the entire length of the passed in lists. This results in $O(n^2)$.

```

public static List<String> stringToListOfWords(String string) {
    List listOfWords = new ArrayList<String>();
    String punctuation = ",./?;:~!@#$$%^&*()";
    String stringWithoutPunc = "";
    // EXTRA CREDIT
    // strips the punctuation
    // checks each character in the string
    for (int i = 0; i < string.length(); i++) {

```

```

        // if the string of punctuations contains the character in the
string, skip
        String current = string.substring(i, i+1);
        // if the current character is not punctuation, add it to the
string
        if (!punctuation.contains(current)) {
            stringWithoutPunc += current;
        }
    }
    // for each word in the string, separate it by the whitespace
    for (String word : stringWithoutPunc.split("\\s+")) {
        // add to the return array
        listOfWords.add(word);
    }
    return listOfWords;
}

```

The function `stringToListOfWords` has **$O(n)$** time complexity. This is because the time increases linearly as more characters are added to the string. The first for loop checks runs through the entire length of the string to check for punctuation. This operates on every item in the list, so the time complexity would be $O(n)$. The second for loop operates on every character of the string also to split the string by the whitespace. The add method has $O(1)$. This also operates on every character of the string and has time complexity $O(n)$. $O(n)$ and $O(n)$ together make $O(n)$.

```

public static <E> void removeAllInstances(List<E> list, E item) {
    int countOfItem = 0;
    // go through the list and count how many times the item pops up
    for (int i = 0; i < list.size(); i++) {
        if (list.get(i) == item || list.get(i).equals(item)) {
            countOfItem++;
        }
    }
    // for the number of times the item pops up, remove from list
}

```

```
        while (countOfItem > 0) {  
            list.remove(item);  
            countOfItem--;  
        }  
    }  
}
```

The `removeAllInstances` function has time complexity **$O(n^2)$** . The first for loop iterates through the entire passed in list to check if each item is the item to be removed. This results in time complexity of $O(n)$. The while loop iterates for every time the item pops up and removes the item. Within the while loop is the `remove` method. The `remove` method has time complexity of $O(n)$ because it iterates through the whole list to find the item to be removed. In the worst case scenario, every item in the list would have to be removed. This would require the while loop to operate on every item in the passed in list, then the list would be iterated through again to find the item. This would result in a time complexity of $O(n^2)$.