Name: Annica Chiu

Here are some basic rules for calculating the Big O for some $T(n)$ or an algorithm.

1. Only the highest degree of n matters. For example

$$T(n) = n^3 + 5n^2 + 10^7 \rightarrow O(n^3)$$

since once n becomes super-massively huge, the other terms just stop mattering.

2. Constant factors don't matter. $T(n) = 500n$ and $T(n) = 0.005n$ both $O(n)$. Again, as n becomes bigger, these constants stop mattering; what matters is the rate of growth. Example:

$$T(n) = 50n^3 - 2n^2 + 400 \rightarrow O(n^3)$$

3. Counting the number of nested loops usually works.

You can turn in this assignment physically to a TA or me. You can also scan and upload your answers.

 _____ 0 points

For each of the following $T(n)$, write the corresponding Big O time complexity. Some series may require research.

1. (2 points) $T(n) = n^2 + 3n + 2$

   1. n^2

2. (2 points) $T(n) = (n^2 + n)(n^2 + \frac{\pi}{2})$

   2. n^4

3. (2 points) $T(n) = 1 + 2 + 3 + \ldots + n - 1 + n$

   3. n

4. (2 points) $T(n) = 1^2 + 2^2 + 3^2 + \ldots + (n-1)^2 + n^2$

   4. n^2

5. (2 points) $T(n) = 10$

   5. 1

6. (2 points) $T(n) = 10^{100}$

   6. 1

7. (2 points) $T(n) = n + \log n$

   7. n

8. (2 points) $T(n) = 12\log(n) + \frac{n}{2} - 400$

   8. n

9. (2 points) $T(n) = (n+1) \cdot \log(n) - n$

   9. n

10. (2 points) $T(n) = \frac{n^4+3n^2+2n}{n}$

    10. n^3

_____ 20 points

11. (4 points) What is the time complexity to get an item from a specific index in an ArrayList?

It is O(1) because, to get an item from a specific index, Java adds the address of the ArrayList by the specific index to get the address of the item. Adding and accessing the item is O(1), so overall it is O(n).

12. (3 points) What is the time complexity remove an item in the middle of an ArrayList?

It is O(n).

13. (3 points) Why?

To remove an item, the program would remove the item and move every item after to the index before it. If an item were removed from the middle of an ArrayList, ever item after the middle would have to move back. The same function would occur on half the items in the list. O(n/2) simplifies to O(n).

14. (3 points) What is the **average** time complexity to add an item to the end of an ArrayList?

It is O(n) because adding to the end does not require the function to move any items, unless the ArrayList needs to reallocate, which is rare.

15. (3 points) What is the **worst case** time complexity to add an item to the end of an ArrayList? What if you have to or don't have to reallocate?

The worst case would be O(n), which when the ArrayList needs to be reallocated. This is because when the ArrayList needs more space, and the entire ArrayList needs to be copied into an array with more space. Each item would be copied, which would result in O(n).
If you do not need to reallocate, the time to add an item to the end of an ArrayList is O(1).

16. (4 points) Taking this all into account, what situations would an ArrayList be the appropriate data structure for storing your data?

An ArrayList would be best for when I need to use indexes to get items, only be adding to the end, and rarely removing items.

```
public static int[] allEvensUnder(int limit){
        if (limit <= 0){
                return new int[0];
        }
        if (limit < 2){
                return new int[1];
        }
        int[] vals =  new int[(limit+1)/2];
        for(int i  = 0;  i <(limit+ 1)/2  ; i++ ) {
                vals[i] = i*2;
        }
        return vals;
}
```

17. (5 points) What is the **time** complexity of the above algorithm?

O(n)
Because the two if statements take O(1) and the for loop executes for every index in vals.
The loop adds even numbers from 0 to the limit to the vals array, which would take O(n).

18. (5 points) What is the **space** complexity of the above algorithm? In other words, how much space is used up as a function if the input size? Think about it, we didn't cover this in the videos.

O(n)
Because the algorithm requires an array. The array will store n items.

```
/*
*  https://rosettacode.org/wiki/Sorting_algorithms/Insertion_sort#Java
*/
public static void insertSort(int[] A){
        for(int i = 1; i < A.length; i++){
                int value = A[i];
                int j = i - 1;
                while(j >= 0 && A[j] > value){
                        A[j + 1] = A[j];
                        j = j - 1;
                }
                A[j + 1] = value;
        }
}
```

19. (10 points) What is the time complexity of the above algorithm?

O(n^2) because in the worst case scenario, the while loop would have to execute every time the for loop executes. If the array to be passed in was numbers in descending order, the while loop would have to iterate on every value. Since the while loop is within a for loop, which is already iterating on every value, the algorithm is O(n^2).

`bogosort` attempts to sort a list by shuffling the items in the list. If the list is unsorted after shuffling, we continue shuffling the list and checking until it is finally sorted.

20. (5 points) What is the worst case run time for `bogosort`?

The worst case would be the program never ends.

21. (5 points) Why?

Because bogosort shuffles randomly every time, there is a possibility that the list is never sorted correctly for all of eternity. This is unlikely, but it is possible.

22. (5 points) What is the average case run time for `bogosort` (Hint: think about a deck of cards)?

The average case run time for bogosort would be O(n!).

23. (5 points) Why?

This is because the total possible arrangements of n items is calculated by n!.

 _____ 20 points

24. (20 points) For each of the methods you wrote in Lab 2, figure out the time complexity of the method you wrote. To turn in this portion, attach a printout of the code and specify the time complexity of each.

See next page...

_____ 20 points

```
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).