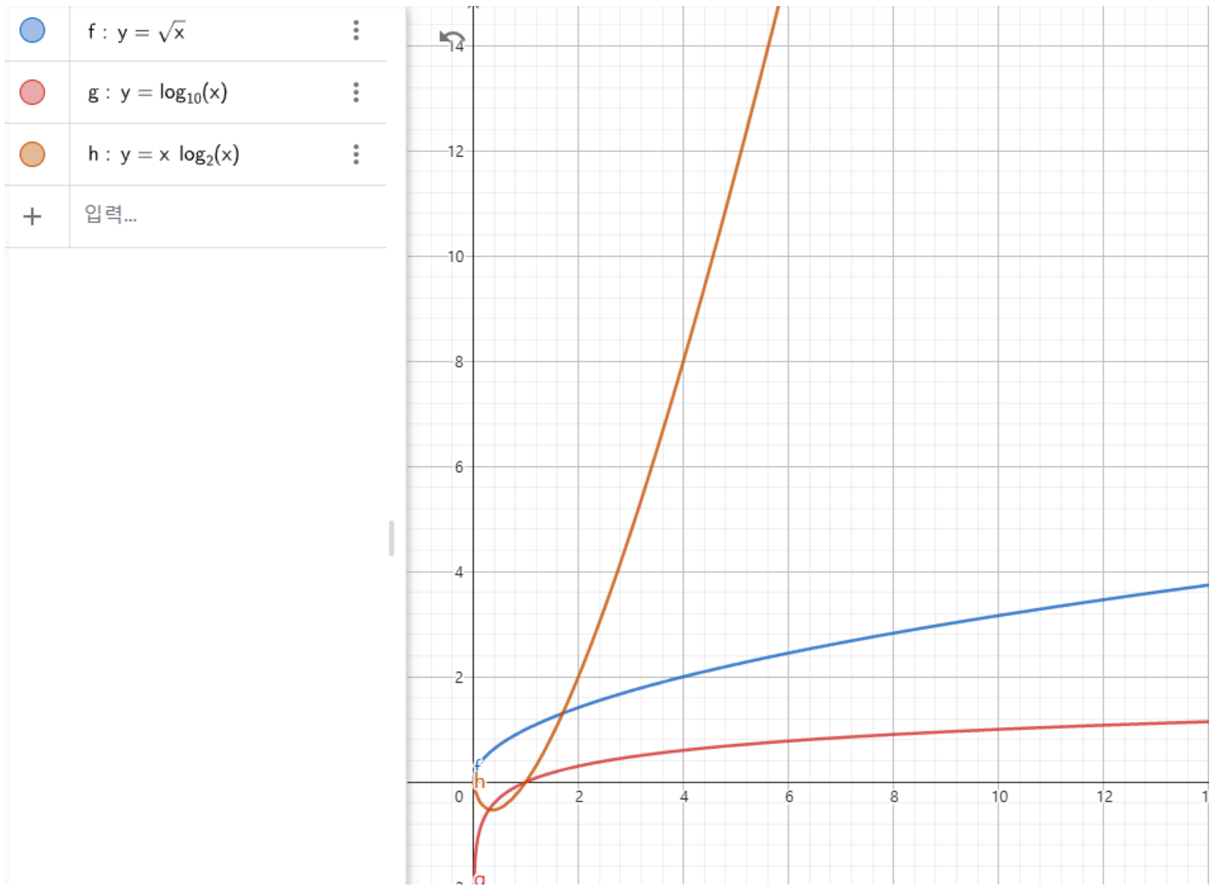
Name: Donghun Yoo
SBU ID: 114000660

1. Compute the tightest order, in big-O notation, of the following functions and codes. You must follow the rigorous derivation steps we did in class for A and B. For C and D, first write the code before providing an analysis (5 pts x 4 = 20 pts).
   A.
   f(n)=sin(n)−10000n^3+10^(−10)n^4
   sin(n) is a periodic function that exists between 1 and -1. So, sin(n) is O(1).
   -10000n^3 is O(n^3).
   10 is O(1).
   10^(-10)n^4 is O(n^4).
   Thus, the time complexity of f(n) is O(n^4)

   B.
   f(n)=√n + *n*log2 *n* + log10 *n*^2 (It's okay to use graphing software to find where the logarithm and the square root functions cross each other. Just don't use the graph as the entire justification for your answer)



   √n is O(n).
   nlog2 n is O(n logn)
   log10 n^2 is O(log n)
   O(log n) < O(n) < O(n logn). So, the time complexity of f(n) is O(n logn).

C.
A Java method that takes an integer array ‘x’ as input and returns the second-smallest element, which is guaranteed to exist in ‘x’.

```java
public static int secondSmallest(int[] x){
   int value;
   for (int i = 0; i < x.length - 1; i ++){
       for (int j = i + 1; j < x.length; j ++){
           if (x[i] > x[j]){
               value = x[i];
               x[i] = x[j];
               x[j] = value;
           }
       }
   }
   return x[1];
}
```

The first for loop is O(n) and the second for loop is O(n). So, the time complexity of the method is O(n) * O(n) = O(n^2)

D.
A method that takes two 2D arrays as inputs and returns another 2D array that contains the element-wise sums of the two inputs.

```java
public static int[][] addArrays(int[][] array1, int[][] array2){
   int[][] newArray = new int[array1.length][array1[0].length];
   for (int i = 0; i < newArray.length; i++) {
       for (int j = 0; j < newArray[0].length; j++){
           newArray[i][j] = array1[i][j] + array2[i][j];
       }
   }
   return newArray;
}
```

The first for loop is O(n) and the second for loop is O(n). So, the time complexity of the method is O(n) * O(n) = O(n^2).

2. Attach your code for HW1's moveTo() implementation. Based on YOUR code, analyze the big-O complexity of that method. The style of analysis should be similar to what we did for the analysis of InsertionSort. Please attach the code!!! We will NOT grade your work if you don't. By 'analysis', I mean accounting for every single line of code you have, not just the overall description (20 pts).

```java
public Piece moveTo(Location from, Location to) {
   Piece piece = getAt(from); /*C1*/
   if (chessBoard[from.getRow()][from.getCol()] != null &&
piece.isMoveAllowed(from, to)) { /*C2*/
      chessBoard[to.getRow()][to.getCol()] = piece; /*C3*/
      return piece; /*C4*/
   }  else { /*C5*/
      return null; /*C6*/
   }

}
```

C1 is O(1), C2 is O(1), C3 is O(1), C4 is O(1), C5 is O(1), C6 is O(1).
In my moveTo() method, I called each piece's isMoveAllowed method, so I also calculated each method.

```java
class BlackKing extends Piece{

   public BlackKing(String name, Location initLoc) {
      super(name, initLoc);
   }

   @Override
   boolean isMoveAllowed(Location source, Location dest) {
      boolean isValidDestination = false;
      Location[] locations = {new Location(0,1), new Location(1,1),
new Location(1,0), new Location(1, -1), new Location(0, -1),
            new Location(-1,-1), new Location(-1,0), new Location(-1,
1)};
      Location sourceToDest = new Location(dest.getRow() -
source.getRow(),dest.getCol() - source.getCol());
      for(int i = 0; i < locations.length; i++){
         if (locations[i].getRow() == sourceToDest.getRow() &&
locations[i].getCol() == sourceToDest.getCol()){
            isValidDestination = true;
         }
      }
      return isValidDestination;
   }
}
```

In BlackKing, it has for loop and its time complexity is O(n)

```java
class BlackPawn extends Piece{
   public BlackPawn(String name, Location initLoc) {
      super(name, initLoc);
   }

   @Override
   boolean isMoveAllowed(Location source, Location dest) {
      boolean isValidDestination = false;
      Location[] locations = {new Location(1,0)};
      Location sourceToDest = new Location(dest.getRow() -
source.getRow(),dest.getCol() - source.getCol());
      for(int i = 0; i < locations.length; i++){
         if (locations[i].getRow() == sourceToDest.getRow() &&
locations[i].getCol() == sourceToDest.getCol()){
            isValidDestination = true;
         }
      }
      return isValidDestination;
   }
}

class WhiteKnight extends Piece{
   public WhiteKnight(String name, Location initLoc) {
      super(name, initLoc);
   }

   @Override
   boolean isMoveAllowed(Location source, Location dest) {
      boolean isValidDestination = false;
      Location[] locations = {new Location(2,1), new Location(1,2),
new Location(-1, 2), new Location(-2, 1), new Location(-2, -1),
            new Location(-1,-2), new Location(1, -2), new
Location(2,-1)};
      Location sourceToDest = new Location(dest.getRow() -
source.getRow(),dest.getCol() - source.getCol());
      for(int i = 0; i < locations.length; i++){
         if (locations[i].getRow() == sourceToDest.getRow() &&
locations[i].getCol() == sourceToDest.getCol()){
            isValidDestination = true;
         }
      }
      return isValidDestination;
   }
}

class WhiteBishop extends Piece{
   public WhiteBishop(String name, Location initLoc) {
      super(name, initLoc);
   }

   @Override
   boolean isMoveAllowed(Location source, Location dest) {
      boolean isValidDestination = false;
      Location[] locations = {new Location(1,1), new Location(2,2),
new Location(3, 3), new Location(4, 4), new Location(5, 5),
            new Location(6,6), new Location(7, 7), new Location(-1,1),
new Location(-2,2), new Location(-3, 3), new Location(-4, 4),
            new Location(-5, 5), new Location(-6,6), new Location(-7,
7),new Location(-1,-1), new Location(-2,-2), new Location(-3, -3),
            new Location(-4, -4), new Location(-5, -5), new
Location(-6,-6), new Location(-7, -7),new Location(1,-1), new
Location(2,-2),
            new Location(3, -3), new Location(4, -4), new Location(5,
-5), new Location(6,-6), new Location(7, -7)};
      Location sourceToDest = new Location(dest.getRow() -
source.getRow(),dest.getCol() - source.getCol());
      for(int i = 0; i < locations.length; i++){
         if (locations[i].getRow() == sourceToDest.getRow() &&
locations[i].getCol() == sourceToDest.getCol()){
            isValidDestination = true;
         }
      }
      return isValidDestination;
   }
```

Also, other pieces have isMoveAllow method and all of the methods have for loop and its time complexity is O(n).

Thus, the overall time complexity of moveTo() method is O(n).

3. Open Complexity.java. Analyze the time complexities of the methods a() and b(). (10 pts)

```
public static long a(int[] arr) {
   int REP_LEN = 1000; /*C1*/
   long start = System.nanoTime(); /*C2*/
   int sum = 0;/*C3*/
   for(int i = 0; i < REP_LEN; i++) { /*C4*/
      for(int j = arr.length - 1; j > 0; j /= 2) /*C5*/
         for(int k = 0; k < arr.length; k++) { /*C6*/
            sum += (arr[j] * arr[k]); /*C7*/
         }
   }
   long end = System.nanoTime(); /*C8*/
   long elapsed_time = (end - start) / 1000000; // In 'ms' C9
   return elapsed_time; /*C10*/
}
```

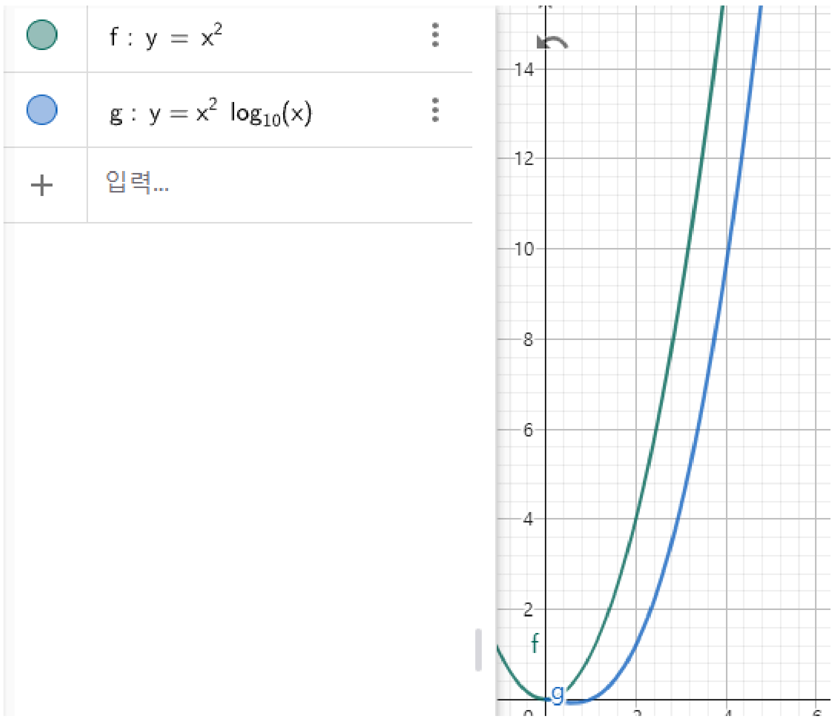
C1 = O(1), C2 = O(1), C3 = O(1), C4 = O(n), C5 = O(n logn), C6 =O(n^2 logn) C7 = O(n^2 logn), C8 = O(1), C9 = (1), C10 = O(1)
Thus, the time complexity of a(int[] arr) is O(n^2 logn).

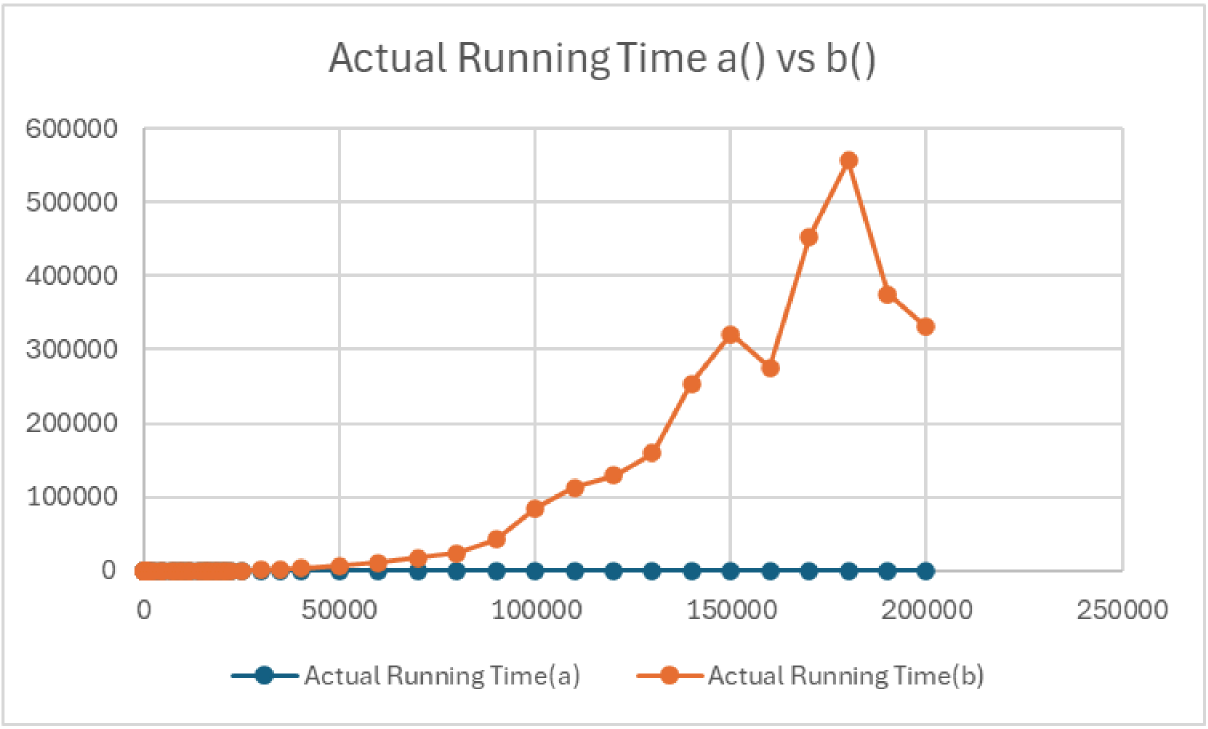
```
public static long b(int[] arr) {
    long start = System.nanoTime(); /*C1*/
    int dSum = 0; /*C2*/
    for(int i = 0; i < Math.max(0, arr.length - 10); i+=20) { /*C3*/
        for(int j = 0; j < arr.length; j+=40) /*C4*/
            dSum += c(arr); /*C5*/
    }
    long end = System.nanoTime(); /*C6*/
    long elapsed_time = (end - start) / 1000000; // In 'ms'C7
    return elapsed_time; /*C8*/
}
```

C1 = O(1), C2 = O(1), C3 = O(n), C4 = O(n^2), C5 = O(n^2), C6 = O(1), C7 = O(1), C8 = O(1)
Thus, overall time complexity of b(int[] arr) is O(n^2)

4. Now execute Complexity.java multiple times with varying values for IN_SIZE to collect the running times of methods a() and b(). Draw a plot containing two curves, one for a() and one for b(), that depicts the actual running time (y-axis) in milliseconds vs. the size of the input (x-axis). What kind of trend can you find, and how does it compare to the theoretical analysis you gave in #3? Don't just say 'They're the same' or 'They're different' but try to describe the qualitative properties of the plots. Attach the plot as well. CAUTION: The overall running time will get larger as you increase IN_SIZE. Be patient! (10 pts)

   Theoretically, the time complexity of a is O(n^2 logn) and the time complexity of b is O(n^2). As seen in the graph, f increases faster than g.





In the actual running time graph in a and b, there are little differences in that some of the values fluctuate but b(O(n^2) increases faster than a(O(n^2 log(n)).

5. Implement the following count() method that performs as follows: Given two arrays ‘a’ and ‘b’, this algorithm should return the number of items in ‘a’ whose value is larger than the sum of elements in ‘b’. (e.g., If a = {1, 12, 3, 6, 10, -1, 5} and b = {1, 2, 3}, then count(a, b) should return 2, because only 12 and 10 are greater than 6=1x2x3). Provide two versions of your algorithm: One running in linear time, and the other running in quadratic time. Justify your time complexities by analyzing your codes (20 x 2 = 40 pts).

```java
public static int linearCount(int[] a, int[] b){
    int sumb = 0; /*C1*/
    int count = 0; /*C2*/
    for(int i = 0; i < b.length; i++){ /*C3*/
        sumb += b[i]; /*C4*/
    }
    for (int i = 0; i < a.length; i++){ /*C5*/
        if (sumb < a[i]){ /*C6*/
            count ++; /*C7*/
        }
    }
    return count; /*C8*/
}

public static int quadraticCount(int[] a, int[] b){
    int count = 0; /*C9*/
    for (int i = 0; i < a.length; i++){ /*C10*/
        int sumb = 0; /*C11*/
        for(int j = 0; j < b.length; j++){ /*C12*/
            sumb += b[j]; /*C13*/
        }
        if (sumb < a[i]){ /*C14*/
            count ++; /*C15*/
        }
    }
    return count; /*C16*/
}
```

In linearCount(), C1 = O(1), C2 = O(1), C3 = O(n), C4 = O(n), C5 = O(n), C6 = O(n), C7 = O(n), C8 = O(1). C3 and C5 are parallel, thus the time complexity of linearCount() is O(n);

In quadraticCount(), C9 = O(1), C10 = O(n), C11 = O(n), C12 = O(n^2), C13 = O(n^2), C14 = O(n^2), C15 = O(n^2), C16 = O(1). Thus, the time complexity of quadraticCount() is O(n^2)