Dahyun Eom 115943034

1(a).

$f(n) = 1 - 10000n^2 + 10^{-10}n^4$

O(n^4) because the highest order term in the function is $10^{-10}n^4$.

1(b).

$f(n) = \frac{\sqrt{n}}{n} + \sqrt{n} + \sqrt{log(n)}$

O($\sqrt{n}$) because the highest order term in the function is $\sqrt{n}$.

1(c).

```
public static int minimumElement(int[] sortedArr){
    return sortedArr[0];
}
```

O(1) because the method just needs to return the first index of the array since the array is already sorted.

Dahyun Eom 115943034

1(d).

```
for(int i=0; i*i < n/3; i++){          //1
    int[][] x = new int[2*n][n];       //2
    for(int j=0; j<x.length; j++){     //3
        for(int k=0; k < j; k++){      //4
            x[i][k] = i+1;             //5
        }
    }
}
```

$O(n^2\sqrt{n})$

Below is the number of times each line executes big-O complexity for each line.

Line 1: $\sqrt{\frac{n}{3}}$+1 times because the 2nd statement in the 1st for-loop examines $i^2 < \frac{n}{3}$ which equals to $i < \sqrt{\frac{n}{3}}$. Also, i starts from 0 and it will execute extra one more time after the last update of i.

Line 2: $\sqrt{\frac{n}{3}}$ times.

Line 3: $(2n + 1) \times \sqrt{\frac{n}{3}}$times. Since x.length is 2n, j goes from 0 until 2n-1 which will loop 2n times and extra +1 time of execution since after the last update of j, it gets checked one last time. And this repeats whenever the outer loop's variable i changes. So, overall it will be $(2n + 1) \times \sqrt{\frac{n}{3}}$.

Line 4: $(2n^2 + n) \times \sqrt{\frac{n}{3}}$times. Integer k starts with 0 and should be less than j. So, from the inner-loop(line3), when j=0, it goes into the inner-inner loop, then line 4 will only execute once since it does not meet the statement of k<j and j becomes 1. It will again pass through the inner-inner loop and this time it will execute twice when the first is k=0 and execute line5 and the 2nd is when k becomes1 and exit the for loop. It will iterate the same process until j=2n-1.

$1 + 2 + \ldots\ldots\ldots + 2n = \frac{2n(2n+1)}{2} = 2n^2 + n$

Also, since inner-inner loop is inside of the outer loop, it executes whenever the outer loop's variable i changes. So, overall it will be $(2n^2 + n) \times \sqrt{\frac{n}{3}}$.

Line 5: $\{n(2n - 1)\} \times \sqrt{\frac{n}{3}}$times. Same with Line 4 except Line 5 does not need to execute once more to check one last time after the k grows.

$0 + 1 + 2 + \ldots\ldots\ldots + (2n - 1) = \frac{(2n-1)2n}{2} = n(2n - 1)$

Also, same with Line 4, since inner-inner loop is inside of the outer loop, it executes whenever the outer loop's variable i changes. So, overall it will be $\{n(2n - 1)\} \times \sqrt{\frac{n}{3}}$.

Dahyun Eom 115943034

The time complexity of this method would be

$$\sqrt{\frac{n}{3}} + 1 + \sqrt{\frac{n}{3}} + (2n + 1)\sqrt{\frac{n}{3}} + (2n^2 + n)\sqrt{\frac{n}{3}} + \{n(2n - 1)\}\sqrt{\frac{n}{3}}$$

However, since we just need to compute the tightest order, following addition rule from Big-O rules, the highest term without the coefficient is $n^2\sqrt{n}$.

$O(n^2\sqrt{n})$

Dahyun Eom 115943034

2.

Let's denote
k = size of string array 'allPlayers'
x = row of the grid (size of the row in matrix)
y = column of the grid (size of the column in matrix)
n in big-O represents the input size. Regardless of whether it's k, x, or y.

```
public Location whereIs(String name) {
    if (occupiedGridsNum()==0){                 //1
        return null;                            //2
    }
    String[] allPlayers = getAllPlayers();      //3
    boolean checker = false;                    //4
    for (String searcher : allPlayers){         //5
        if (searcher.equals(name)){             //6
            checker = true;                     //7
        }
    }
    if (checker == false){                      //8
        return null;                            //9
    }

    Locations playerLocation = new Locations();                 //10
    for(int i=0; i<maxRow; i++){                                //11
        for(int j=0; j<maxColumn; j++){                         //12
            if(grid[i][j]!= null && grid[i][j].equals(name)){   //13
                playerLocation.setRow(i);                       //14
                playerLocation.setCol(j);                       //15
            }
        }
    }
    return playerLocation;                                      //16
}
```

(for simplicity, I will not consider the time complexity of occupiedGridsNum() method and getAllPlayers() method)
Below is the number of times each line executes and big-O complexity for each line.
Line1: 1time. O(1)
Line 2: 1 or 0 time. O(1)
Line 3: 1time. O(1)
Line 4: 1time. O(1)
Line 5: k times. O(n)
Line 6: k times. O(n)
Line 7: unknown. However, less than k times and it is not important when analyzing the method's big-O complexity.
Line 8: 1 time. O(1)
Line 9: 1 or 0 time. O(1)
Line 10: 1 time. O(1)

Dahyun Eom 115943034

Line 11: x+1 times in total because the variable 'i' increases up to x-1 and gets checked one last time. O(n)
Line 12: x(y+1) times in total. Because the variable 'j' increases up to y-1 and gets checked one last time and this repeats x times since the outer for-loop loops x times. O(n^2)
Line 13: xy times since inner for-loop will loop y times and outer for-loop will loop x times. O(n^2)
Line 14: 1 time. Because it's just once when the if statement is true. O(1)
Line 15: 1 time. Same reason with Line 14. O(1)
Line 16: 1 time. O(1)

Based on the big-O rules, for the big-O complexity of this method, we just need to find the highest order within the lines inside the method whereIs().
So, it is O(n^2).

Dahyun Eom 115943034

3.

```java
public static long a(int[] arr) {
    long start = System.nanoTime();                          //1
    int sum = 0;                                             //2
    for(int i = 0; i < CSE214.REP_LEN; i++) {                //3
        for(int j = 0; j < arr.length; j++)                  //4
            sum += arr[j];                                   //5
    }
    long end = System.nanoTime();                            //6
    long elapsed_time = (end - start) / 1000000; // In 'ms'  //7
    return elapsed_time;                                     //8
}
```

Denote n = arr.length (input array length).

Below is the number of times each line executes.
Line 1: 1 time.
Line 2: 1 time.
Line 3: 1001 times. Because CSE214.REP_LEN is fixed to 1000 and it will execute extra one more time after the last update of i but not goes into the inner loop.
Line 4: $1000 \times (n + 1)$ times. Because the variable 'j' increases up to n-1 and gets checked one last time and this repeats 1000 times since the outer for-loop loops 1000 times.
Line 5: 1000n times. Because the inner for-loop loops n times and this repeats 1000 times due to the outer for-loop.
Line 6: 1 time.
Line 7: 1 time.
Line 8: 1 time.

Sum of the number of times each line executes:
$1 + 1 + 1001 + 1000(n + 1) + 1000n + 1 + 1 + 1$
$= 2000n + 2006$

However, Big-O only considers the highest order term without the coefficient, so it can be presented as O(n).

Dahyun Eom 115943034

```java
public static long b(int[] arr) {
    long start = System.nanoTime();                          //1
    int dSum = 0;                                            //2
    for(int i = 0; i < arr.length; i++) {                    //3
        for(int j = 0; j < arr.length; j++)                  //4
            dSum += arr[j];                                  //5
    }
    long end = System.nanoTime();                            //6
    long elapsed_time = (end - start) / 1000000; // In 'ms'  //7
    return elapsed_time;                                     //8
}
```

Below is the number of times each line executes.
Line 1: 1time.
Line 2: 1time.
Line 3: n+1 times. Because the variable 'i' increases up to n-1 and gets checked one last time.
Line 4: $n(n+1)$ times. Because the variable 'j' increases up to n-1 and gets checked one last time and this repeats n times since the outer for-loop loops n times.
Line 5: n^2times. Because the inner for-loop loops n times and this repeats n times due to the outer for-loop.
Line 6: 1 time.
Line 7: 1 time.
Line 8: 1 time.

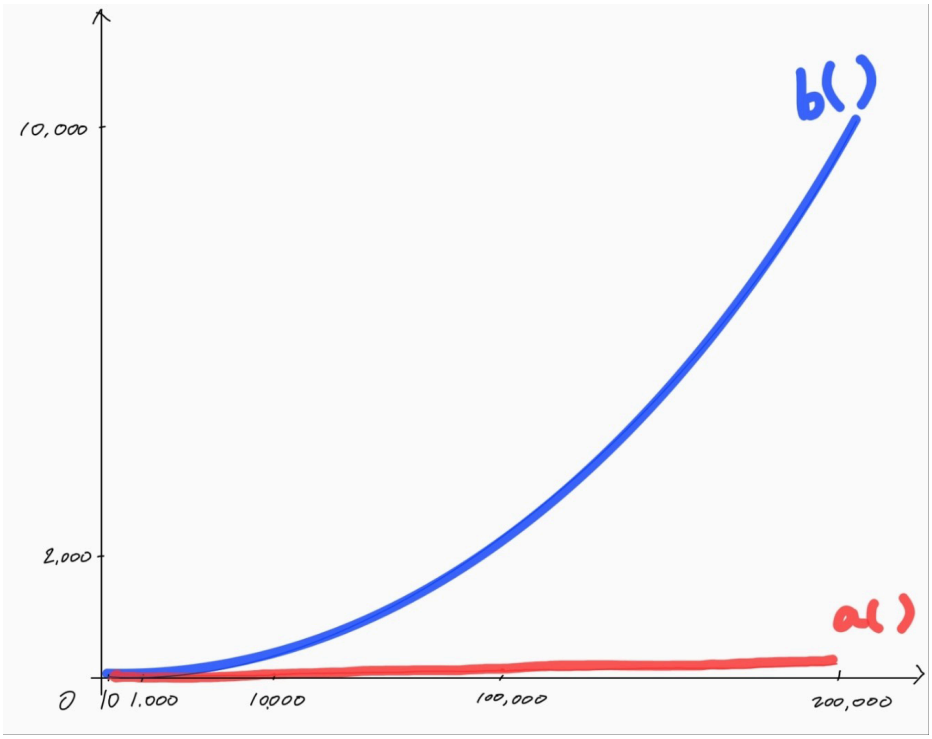
Sum of the number of times each line executes:

$$1 + 1 + (n+1) + n(n+1) + n^2 + 1 + 1 + 1$$
$$= 2n^2 + 2n + 6$$

However, Big-O only considers the highest order term without the coefficient, so it can be present as $O(n^2)$.

Dahyun Eom 115943034

4.



(y-axis: actual running time in milliseconds, x-axis: the size of the input)

Input data size	10	1000	10,000	100,000	200,000
Output of method a() (ms)	0	2	5	29	55
Output of method b() (ms)	0	2	28	2506	9995

From the plot, I found out that when the input size is small, then there is no such difference in running time between method a() and b(). However, since method b()'s time complexity is quadratic, method b()'s running time will be much larger than method a()'s as input size grows.

Dahyun Eom 115943034

5.

```
public static int count(int[] arr, int item){
    if(arr[0]<arr[arr.length-1]){ //ascending     //1
        if(item<arr[0]){                          //2
            return 0;                             //3
        }
        else if(item == arr[0]){                  //4
            int count =0;                         //5
            int i=0;                              //6
            while(arr[i] == item){                //7
                count++;                          //8
                i++;                              //9
                if(i == arr.length){              //10
                    break;                        //11
                }
            }
            return count;                         //12
        }
        else if (item > arr[0]){                  //13
            int count=0;                          //14
            int i=0;                              //15
            while(arr[i] <= item){                //16
                count++;                          //17
                i++;                              //18
                if(i == arr.length){              //19
                    break;                        //20
                }
            }
            return count;                         //21
        }
    }

    else if(arr[0]>arr[arr.length-1]){ //descending  //22

        if(item>=arr[0]){                             //23
            return arr.length;                        //24
        }
        else{                                         //25
            int countBig=0;                           //26
            int i=0;                                  //27
            while (arr[i]>item){                      //28
                i++;                                  //29
                countBig++;                           //30
                if(i == arr.length){                  //31
                    break;                            //32
                }
            }
            return arr.length-countBig;               //33
        }
    }

    else if(arr[0] == arr[arr.length-1]){             //34
        if(item>=arr[0]){                             //35
            return arr.length;                        //36
        }
        else{                                         //37
            return 0;                                 //38
        }
    }

    return 0;                                         //39
}
```

Dahyun Eom 115943034

Let n = arr.length (size of the input data)

Best case: O(1),
For example, **When the ascending order array is given and the item is smaller than the first index of the array.**
It goes into the first if statement, Line 1 because the array is in ascending order. Since it meets the condition item<arr[0] in Line2, it will immediately return 0.
In this case, the time complexity does not matter with the length of the array. It just executes Line 1, 2, 3 once each. So, the time complexity with big-O is O(1).

Worst case: O(n).
For example, **When the descending order array is given and the item is less than the last element(arr[arr.length-1]) of the array.**
It goes into the first if statement(Line 1) to check whether it meets the condition or not. However, since the array is descending order, it does not go into Line 2, but jumps to Line 22 to check whether the array is in descending order. Since it is, it executes Line 23, which this case does not meet the condition. So, it goes to the else part(26~33) and executes Line 26, 27. The case needs to run the while loop, and since the item is less than the last element of the array, the while loop has to loop n times. Because whenever the while-loop loops once, the ‘i’ will increase and this will repeat until the ‘i’ becomes the last index of the array and loops last once more. So, in this case, the time complexity matters with the length of the input array because the while-loop needs to loop as much as the number of elements in the array. So, the time complexity with big-O is O(n).