Solution Set 2

Solutions to problem set 2.

Solution 1

Big O notation denotes an upper bound but is silent about the lower bound. Treat it like a cap on the worst that can happen. So when someone says that an algorithm has a lower bound of $O(n^2)$ that translates into saying the lower bound can at worst be quadratic in complexity. By definition, the lower bound is the best an algorithm can perform. Combine the two statements and your friend is saying the algorithm he found in the best case can perform in quadratic time or better. It can also perform in linear time or constant time, we just don't know. So your friend is effectively not telling you anything about the lower bound on the performance of his new found algorithm.

Solution 2

We can employ the Big O definition we learnt earlier to determine if $O(2^{2n})$ and $O(2^n)$ are equivalent. Intuitively, it would feel 2^{2n} will yield a greater number than 2^n for values of $n > 0$. According to Big O definition:

$$0 \leq f(n) \leq cg(n)$$

For $O(2^{2n})$ and $O(2^n)$ to be equivalent, we need to prove that the following inequalities would hold:

$$0 \leq 2^{2n} \leq c2^n$$

$$0 \leq 2^n \leq c2^{2n}$$

Let's consider the first inequality

$$0 \leq 2^{2n} \leq c2^n$$

Divide both sides by 2^n will give

$$0 \leq 2^n \leq c$$

It's easy to see that once we fix the constant *c*, we can vary the *n* to make the inequality false. Therefore, $O(2^{2n})$ and $O(2^n)$ are not equivalent.

Solution 3

Counting Sort and ***Radix Sort*** are two algorithms which have the same best, worst and average case complexities.

Solution 4

Below is the code-snippet for which we want to find the complexity

```
1.     void averager(int[] A) {
2.
3.         float avg = 0.0f;
4.         int j, count;
5.
6.         for (j = 0; j < A.length; j++) {
7.             avg += A[j];
8.         }
9.
10.        avg = avg / A.length;
11.
12.        count = j = 0;
13.
14.        do {
15.
16.            while (j < A.length && A[j] != avg) {
17.                j++;
18.            }
19.
20.            if (j < A.length) {
```

```
21.                 A[j++] = 0;
22.                 count++;

23.             }
24.         } while (j < A.length);
25.     }
```

The for loop from **lines 6-8** calculates the average of the contents of the array and is O(n) in complexity. The `do-while` loop from **lines 14-24** is tricky. A novice may wrongly conclude the complexity of the snippet to be $O(n^2)$ by glancing at the nested while loop within the outer do-while loop.

There are two possibilities, one the average calculated for the given input array also occurs in the array and second the average is a float and doesn't appear in the array.

If the average is a float then the nested while loop on **lines 16-17** will increment the value of the variable *j* to the size of the input array and the do-while condition will also become false. The complexity in this case will be O(n)

If the average does appear in the array, and say in the worst case the array consists of all the same numbers then the nested while loop of **lines 16-17** will not run and the if clause of **lines 20-23** will kick-in and increment j to the size of the array as the outer do-while loop iterates.

Hence the overall complexity of the snippet is O(n).

Solution 6

To the untrained eye, it may appear that since there's a single loop, the runtime complexity would likely be O(n), which is incorrect. If you look at the snippet again:

```
1.     void complexMethod(int[] array) {
2.         int n = array.length;
3.         int runFor = Math.pow(-1, n) * Math.pow(n, 2);
4.         for (int i = 0; i < runFor; i++) {
5.             System.out.println("Find how complex I am ?")
6.         }
7.     }
```

the worst case happens for even sizes of the input array. The loop doesn't run when the size of the array is an odd number. Next, the loop runs a quadratic

number of times the length of the array when the length is even. Ask yourself,

does a bigger array result in the loop running for a longer period of time? Yes, it does. The bigger the array size and provided it is an even number, the single loop runs as if it were a nesting of two loops. Therefore the complexity of this snippet of code is $O(n^2)$;

Solution 7

Let's examine the snippet again:

```
void complexMethod(int n, int m) {

   for (int j = 0; j < n; j++) {
      for (int i = 0; i < m % n; i++) {
          System.out.println("I am complex")
      }
   }
}
```

- Let's start with the case when n equals m. In that case, **m%n** will equal 0 and the inner loop will not run. So complexity will be **O(n)**.
- If **n** > **m** then m%n will equal **m**, so now the inner loop will run for m times, giving us a total complexity of **O(m*n)**.
- The last case is when **n** < **m** then m%n will equal values ranging from **1 to n-1**. So the inner loop in the worst case would run for **n-1** times. The complexity in the last case would then be O(n*(n-1)) which is **$O(n^2)$**.

Note that **O(m*n)** is a tighter bound for the second case, but since we are talking in big O terms, we can say for the second case, when **n** > **m**, then **O(m*n)** < **$O(n^2)$** so for the second case, we can say the complexity will be **$O(n^2)$**.

So in the worst case, the complexity would be **$O(n^2)$**.

Solution 8

It may seem that the complexity of the snippet is cubic since we have 3 loops. But the second loop runs for a constant number of times.

```
void someMethod(int n, int m) {

   for (int j = 0; j < n; j++) {
      for (int i = 0; i < 3; i++) {
         for (int i = 0; i < n; i++) {
             System.out.println("I have 3 loops");
         }
      }
   }
}
```

One way to think about the above snippet is to unroll the second loop. We can say the above code is equivalent to the following

```
void someMethod(int n) {

   for (int j = 0; j < n; j++) {
      for (int i = 0; i < n; i++) {
          System.out.println("I have 3 loops");
      }
      for (int i = 0; i < n; i++) {
          System.out.println("I have 3 loops");
      }
      for (int i = 0; i < n; i++) {
          System.out.println("I have 3 loops");
      }
   }
}
```

The output of both the snippets would be the same. From the unrolling, it is evident that the three inner loops contribute $n + n + n = 3n = O(n)$ and the outmost loop runs for n too therefore the overall complexity is $O(n^2)$.

Solution 9

```
void someMethod(int n, int m) {

   for (int j = 0; j < n; j++) {
      for (int i = 0; i < m; i++) {
          System.out.println("I have 2 loops");
      }
   }
```

```
    }
}
```

The above snippet is a traditional example of nested loops. Whenever you get nested loops, each running for a variable number of times, the complexity of the entire snippet is the product of the variables controlling the repetition of each loop. The string message would be printed for a total of ***m*n*** times and thus the overall, complexity will be ***O(m*n)*** = ***O(mn)***.