CSCI 2114: Tashfeen's Data Structures

# Homework 2

Let's remind ourselves of the asymptotic order notation[1]. Let $f(x)$ and $g(x)$ be functions of a positive $x$.

$$f(x) = O(g(x))$$

when there is positive constant $c$ such that,

$$f(x) \leq cg(x)$$

for all $x \geq n$. Function $f$ is then stated as *big oh* of $g$. Similarly, $f$ is *big omega* of $g$, i. e.,

$$f(x) = \Omega(g(x))$$

when there exist positive constants $c, n$ such that for all $x \geq n$ we have,

$$f(x) \geq cg(x)$$

Lastly, if $f$ is both big-$O$ and big-$\Omega$ of $g$,

$$\Omega(g(x)) = f(x) = O(g(x))$$

Then such a tight bound is stated as $f$ is big-$\Theta$ of $g$,

$$f(x) = \Theta(g(x))$$

**Question 1.** Consider the following Java subroutine,

```java
public static void conditionalWork(int n) {
    for (int i = 0; i < n; i++)
        if (Math.random() < 0.5)
            taskA()
        else
            taskB()
}
```

1) If on a certain machine the function `taskA()` takes 2 seconds on each call in the loop and the function `taskB()` takes 4 seconds then for $n = 10$, what is the total number of *expected* seconds taken by the subroutine `public static void conditionalWork(int n)`?
2) The subroutine `conditionalWork(int n)` is ran on a much faster machine reducing the time taken by `taskA()` on each call down to $\frac{1}{25}^{\text{th}}$ of a second and `taskB()` now takes $\frac{1}{5}^{\text{th}}$ of a second. For $n = 10$, what is the new total number of seconds *expected* by the subroutine `conditionalWork(int n)`?
3) Assume that for any arbitrary $n$, `taskA()` takes $\log(n)$ steps while `taskB()` takes $2n^2$ steps. Let $T(n)$ be the total number of steps *expected* by `conditionalWork(int n)`, what is $T(n)$?
4) What is the *best expected asymptotic* runtime complexity written as $T(n) = \Omega(g(n))$?
5) What is the *worst expected asymptotic* runtime complexity written as $T(n) = O(g(n))$?
6) If `taskB()` took $4\log(n)$ steps, what is the *tight expected asymptotic* runtime complexity written as $T(n) = \Theta(g(n))$?

**Question 2.** In the code snippet bellow, we see two ways of getting the tenth place digit of a Java `int` $n$. Give the runtime complexity of each method using the *big oh* notation. Justify your answer.

```java
System.out.println(n % 10);
System.out.println(String.valueOf(n).charAt(String.valueOf(n).length() - 1));
```

[1]Sometimes also referred to as the Bachmann-Landau notation.

**Question 3.** Consider the following functions,

$$\begin{aligned} a(x) &= 2(x^3+1)(x^2-1)+2 \\ b(x) &= 2x^2 \\ \alpha(x) &= x^2 \\ \beta(x) &= \pi x^2 \end{aligned}$$

1) Observe that $b(x) = O(a(x))$ because $b(x) \leq a(x)$ past a certain $x = n$. What is the value of $n$ in this case?
2) Observe that $\beta(x) = O(\alpha(x))$ because there exists a $c$ such that $\beta(x) \leq c\alpha(x)$ for all $x > 0$. What is the value of $c$ in this case?

**Question 4.** Given in listing 1 is a Java subroutine that sorts an array of non-negative Java integers in ascending order.

```java
    public static void sortIntegers(int[] toSort) {
        int i = 0, j = 0, k = 0, max = Integer.MIN_VALUE;
        for (i = 0; i < toSort.length; i++)
            max = toSort[i] > max ? toSort[i] : max;
        int[] counts = new int[max + 1];
        for (i = 0; i < toSort.length; i++)
            counts[toSort[i]]++;
        for (i = 0; i < counts.length; i++)
            for (j = 0; j < counts[i]; j++)
                toSort[k++] = i;
    }
```

Listing 1. Linear time algorithm for sorting unique integers.

Assume that the length of the input parameter `int[] toSort` is $n+1$, the $\max(\texttt{toSort}) \leq n$ and that `toSort` has unique elements. Let $T(n)$ be the worst and the average case complexity of the algorithm's runtime and $S(\texttt{toSort})$ be the worst case complexity of the memory-space.

1) What is $T(n)$?
2) What are $O(T(n)), O(S(\texttt{toSort}))$?
3) Look up and state the *big-O* of the average case complexity of Java's built-in `Arrays.sort(int[])`. Is this better than $O(T(n))$ from the previous step of this question?

**Question 5.** Given bellow is the Java implementation of the sieve of Eratosthenes. This particular implementation marks all the composite numbers as `true` since Java allocates `false` to all the elements of a newly created boolean arrays.

```java
public static void eratosthenes(boolean[] toSieve) {
    toSieve[0] = true;
    toSieve[1] = true;
    for (int i = 2; i < Math.sqrt(toSieve.length); i++)
        if (!toSieve[i])
            for (int j = i*i; j < toSieve.length; j += i)
                toSieve[j] = true;
}
```

Give an upper-bound on the runtime complexity of this particular implementation of the sieve of Eratosthenes. The better upper-bound you give, the more credit you get. Justify your answer.

## Submission Instructions

Submit a PDF file with your answers.

Oklahoma City University, Petree College of Arts & Sciences, Computer Science