CS 141, Spring 2019 Homework 1

Problem 0. (20 points) Gradescope/Piazza

Problem 1. (20 points) Order the following list of functions by the big-Oh notation, i.e., rank them by order of growth. Group together (for example, by underlining) those functions that are big-Theta of one another. Logarithms are base two unless indicated otherwise.

$3n+2$	n^3+n^2	$\log^2 n$	$\log\log n$	2^{2^n}	4^{n-1}
$2^{\log n}$	$2^{\sqrt{2\log n}}$	$\sqrt{n}$	n^3	1	$3^{n/3}$
$10000n^2$	$2^{2^{n+1}}$	e^n	$n^{\log\log n}$	$\log n$	$(\log n)^2$
2^n	$n3^n$	$4^{\log n}$	$4n^2/\sqrt{n}$	$\sqrt{\log n}$	$n\log n$

Answer: Here are the function in decreasing order of asymptotic growth (fastest growing to slowest growing)

$2^{2^{n+1}}$	2^{2^n}	4^{n-1}	$n3^n$	e^n
2^n	$3^{n/3}$	$n^{\log\log n}$	$\{n^3+n^2, n^3\}$	$\{10000n^2, 4^{\log n}\}$
$4n^2/\sqrt{n}$	$n\log n$	$\{3n+2, 2^{\log n}\}$	$\sqrt{n}$	$2^{\sqrt{2\log n}}$
$\{\log^2 n, (\log n)^2\}$	$\log n$	$\sqrt{\log n}$	$\log\log n$	1

Functions in $\{\}$ are Θ of each other.

Problem 2. (20 points)

Give a tight bound (using the big-theta notation) on the time complexity of following method as a function of n. For simplicity, you can assume n to be a power of two.

Algorithm WeirdLoop (n : **integer**)

```
i ← n
while i ≥ 1 do
    for j ← 1 to i do
        k ← 1
        while k ≤ n do
            k ← 2k
    i ← i/2
```

Answer: First note that the innermost loop (**while** k) takes always $O(\log n)$, since it does not depend on i and j. When $i = n$, the **for** j loop is executed n times, where each iteration costs $\log n$, for a total of $n \log n$. When $i = n/2$, the **for** j loop is executed $n/2$ times, where each iteration costs $\log n$-time, for a total of $(n/2) \log n$. ... When $i = 1$, the **for** j loop is executed once, for a total of $\log n$ time. In summary, the total complexity is $(n+n/2+n/4 \ldots +2+1) \log n \leq 2n \log n \in \Theta(n \log n)$.

Problem 3. (20 points)

You are facing a high wall that stretches infinitely in both directions. There is a door in the wall, but you don't know how far away or in which direction. It is pitch dark, but you have a very dim lighted candle that will enable you to see the door when you are right next to it. Show that there is an algorithm that enables you to find the door by walking at most $O(n)$ steps, where n is the number of steps that you would have taken if you knew where the door is and walked directly to it. What is the constant multiple in the big-O bound for your algorithm?

Answer: First, note that even if we knew the true distance n, we would need $3n$ steps in the worst case. Can linear-time be achieved and how bad is the constant when we do not know n?

Consider the following algorithm.

1. $k \to 1$
2. take k steps on left
3. if door found then STOP
4. take k steps on the right (back to the origin)
5. $k \to 2 \cdot k$
6. take k steps on the right
7. if door found then STOP
8. take k steps on the left (back to the origin)
9. $k \to 2 \cdot k$
10. goto 2.

Analysis: First, assume that $n = 2^q$. Note that the worst case is when you travel the first time 2^q steps left and the door is 2^q steps on the right of the origin. In that case you will travel 2^q to the left, come back to the origin, walk another 2^q to the right, and finally reach the door. Then, the number of steps would is

$$(1 + 1 + 2 + 2 + ... + 2^q + 2^q) + 2^q = 2\left(2^{q+1} - 1\right) + 2^q = 5 \cdot 2^q - 2 = 5n - 2$$

In general, however, n is not a power of two. Let us set $q = \lfloor \log_2 n \rfloor$. Note that $2^q < n < 2^{q+1}$. Because we did not find the door at 2^q, the algorithm will go back to the origin, go in the other direction 2^{q+1} steps, come back, and finally travel n positions to the door. Therefore, we have

$$1 + 1 + 2 + 2 + ... + 2^q + 2^q + 2^{q+1} + 2^{q+1} + n = 2\left(2^{q+2} - 1\right) + n = 8 \cdot 2^q - 2 + n = 9n - 2$$

So the constant is 5 when n is a power of two, 9 otherwise (worst-case).

The complexity of the algorithm is $O(n)$, although different algorithms may result in different constants.

Problem 4. (20 points) Consider the following "proof" that the solution $T(n)$ to the following recurrence relation

$$T(n) = \begin{cases} 1 & n = 1 \\ T(n-1) + n & n > 1 \end{cases}$$

is $O(n)$.

"Proof": *Base case* ($n = 1$)*:* $T(1) = 1$ which is $O(1)$.
Induction step ($n > 1$)*:* Assume that the claim is true for $n' < n$. Consider the recurrence for n, $T(n) = T(n-1) + n$. By induction hypothesis $T(n-1) \in O(n-1)$. Also, $n \in O(n)$. Then, $T(n) \in O((n-1)+n)$ by the properties of the big-Oh. Therefore, $T(n) \in O(n)$, since $O((n-1)+n)$ is $O(n)$.

We know however that this recurrence relation has solution $T(n) \in \Theta(n^2)$ (see slides). What is wrong with this "proof"?

Answer: In order to prove that $T(n) \in O(n)$, we have to find constants C and n_0, such that when $n \geq n_0$, we have $T(n) \leq Cn$. In the induction step, we assume the claim to be true for $n-1$, that is $T(n-1) \leq C(n-1)$. In the induction step we have

$$T(n) = T(n-1) + n \leq C(n-1) + n = (C+1)n - C.$$

Remember that we need to prove that $T(n) \leq Cn$. If we try to solve $(C+1)n - C \leq Cn$, we get $C \geq n$, which implies that C cannot be a constant. The proof is wrong because we cannot find a constant to satisfy the definition of big-Oh.