Homework 1: CS 141
(Asymptotic notations)

Intermediate Data Structures and Algorithms (Spring 2022)
University of California, Riverside
Due: April 15^{th}

Brandon Paulsen

Problem 1: Let n be a positive integer. Partition the following set of 15 functions into equivalence classes such that $f(n)$ and $g(n)$ are in the same equivalence class if and only if $f(n) = \Theta(g(n))$. (If a base of a logarithm is not specified, it is assumed to be equal 2.)

$$9^{\log_3 \sqrt{n}} \qquad n^n \qquad 8e^2 + \frac{1}{e^n} \qquad 4^{\log 8 + 2^n} \qquad \log(4^n)$$

$$n \log(n!) \qquad 4^{(\frac{n}{2})} \qquad n^2 \qquad 4^n + n^5 \qquad (0.5)^n$$

$$n! \qquad n^n + n! \qquad 2^{2n} \qquad n^2 \log n \qquad \log_5(n^{\frac{1}{3}})$$

Then arrange the equivalence classes into a sequence

$$C_1, C_2, C_3 \cdots$$

such that for any $f(n) \in C_i$ and $g(n) \in C_{i+1}$ $f(n) = O(g(n))$.

Solution 1: Note: $f(n) \in \Theta(g(n)) \iff f(n) \in O(g(n)) \wedge f(n) \in \Omega(g(n))$

$$9^{\log_3 \sqrt{n}} = (3 \cdot 3)^{\log_3 \sqrt{n}} = (3^{\log_3 \sqrt{n}}) \cdot (3^{\log_3 \sqrt{n}}) = \sqrt{n}^2 = n \in \boldsymbol{\Theta(n)} \quad \forall n \geq 0$$

$$n^n \in \boldsymbol{\Theta(n!)}$$

$$8e^2 + \frac{1}{e^n} \in \Theta(1) \text{ since } 8e^2 \text{ is constant and } \lim_{n \to \infty} \frac{1}{e^n} = 0 \in \boldsymbol{\Theta(1)}$$

$$4^{\log 8 + 2^n} = 4^{\log 8} \cdot 4^{2^n} = 64 \cdot 16^n \in \boldsymbol{\Theta(c^n)}$$

$$\log(4^n) = \log(4) \cdot n = 2n \in \boldsymbol{\Theta(n)}$$

$$n \log(n!) \in \boldsymbol{\Theta(n^2 \log(n))} \text{ by logarithm rules}$$

$$4^{(\frac{n}{2})} = 4^{\frac{1}{2}^n} = 2^n \in \boldsymbol{\Theta(c^n)}$$

$$n^2 \in \boldsymbol{\Theta(n^2)}$$

$$4^n + n^5 \in \boldsymbol{\Theta(c^n)}$$

$$(0.5)^n \in \boldsymbol{\Theta(1)} \text{ since } \lim_{n\to\infty} 0.5^n = 0$$

$$n! \in \boldsymbol{\Theta(n!)}$$

$$n^n + n! \in \boldsymbol{\Theta(n!)}$$

$$2^{2n} = 4^n \in \boldsymbol{\Theta(c^n)}$$

$$n^2 \log n \in \boldsymbol{\Theta(n^2 \log(n))}$$

$$\log_5(n^{\frac{1}{3}}) = \frac{1}{3} log_5(n) \in \boldsymbol{\Theta(\log(n))}$$

Then, we can partition these functions into equivalence classes based on their Θ - classes and order them as follows:

$$C_1 = \{8e^2 + \frac{1}{e^n}\} \in \boldsymbol{\Theta(1)}$$

$$C_2 = \{\log_5(n^{\frac{1}{3}}\} \in \boldsymbol{\Theta(\log(n))}$$

$$C_3 = \{9^{\log_3 \sqrt{n}}, \log(4^n)\} \in \boldsymbol{\Theta(n)}$$

$$C_4 = \{n^2\} \in \boldsymbol{\Theta(n^2)}$$

$$C_5 = \{n^2 \log n\} \in \boldsymbol{\Theta(n^2 \log(n))}$$

$$C_6 = \{4^{\log 8 + 2^n}, 4^{(\frac{n}{2})}, 4^n + n^5, 2^{2n}\} \in \boldsymbol{\Theta(c^n)}$$

$$C_7 = \{n^n, n!, n^n + n!\} \in \boldsymbol{\Theta(n!)}$$

Problem 2: An elevator brought you to the ground floor of a building. You want to exit the building. You don't know where the exit from the building is. There are 5 different paths that you can take. The problem is that all five paths are infinitely long, and only one of them has a door that you can use to exit the building. The distance between the elevator and the door is no less than 30 steps. (Make sure to use this information!) You don't know which of the four paths has the door, but you will see it once you are in front of it. Design 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?

Solution 2: To begin, let's try the naïve method - let's try to check if the door is at a given position down each hallway, and if not, check the next position until it is found. The recursive equation that models this is as follows:

$$S(n) = \begin{cases} c = 300 & \text{if } n \leq 30 \\ S(n-1) + 10n & \text{if } n > 30 \end{cases}$$

For each hallway that the door is not down, we must walk down and back, and if the door is not at position n down any of the hallways, we must walk a total of $10n$ steps. It can be seen through iterative substitution that this solution is $O(n^2)$, which is, quite obviously, not $O(n)$.

Now, let's consider a slightly more optimal approach. The problem with the first option is that per step - that is per distance walked down the hall - only one step is a new spot the door could be, whereas $n-1$ of the n steps are places that we have already been. To rectify this, let's try to keep the ratio of new steps to

previously discovered steps at 1/2. We can do this by walking twice as far down the hallway every time - starting with 30 steps. The recurrence equation defining this is as follows:

$$S(n) = \begin{cases} c = 300 & \text{if } n \leq 30 \\ S(n/2) + 10n & \text{if } n > 30 \end{cases}$$

We can then perform iterative substitution as follows, of course assuming that n is sufficiently large:

$$S(n) = S(n/2) + 10n$$

$$S(n) = S(n/4) + \frac{10n}{2} + 10n$$

$$S(n) = S(n/8) + \frac{10n}{4} + \frac{10n}{2} + 10n$$

$$\cdots$$

$$S(n) = S(n/2^i) + \sum_{k=0}^{i-1} \frac{10n}{2^k}$$

This iterative scheme stops for $n/2^i \leq 30 \Rightarrow n \leq 30 \cdot 2^i \Rightarrow \log_2(n/30) \leq i$, where $S(n/2^i)$ becomes a constant. We can then rewrite $\sum_{k=0}^{i-1} \frac{10n}{2^k}$ as $10n \cdot \sum_{k=0}^{i-1} \frac{1}{2^k}$, since $10n$ is constant with regards to the summation variable. We know that $\lim_{i \to \infty} \sum_{k=0}^{i} \frac{1}{2^k} = 2$, so we get that, for large n, $S(n) = 20n \in O(n)$.

Problem 3: Use the method of iterative substitution to solve the following recurrence. Give the asymptotic solution as well. Assume that $T(n)$ is constant for $n \leq 5$.

$$T(n) = \begin{cases} c & \text{if } n \leq 5 \\ T(n-4) + \log(n) & \text{if } n > 5 \end{cases}$$

Solution 3: To begin, we can write out the first few iterations, assuming n is sufficiently large:

$$T(n) = T(n-4) + \log(n)$$

$$T(n) = T(n-8) + \log(n-4) + \log(n)$$

$$T(n) = T(n-12) + \log(n-8) + \log(n-4) + \log(n)$$

$$\cdots$$

$$T(n) = T(n-4i) + \sum_{k=0}^{i-1} \log(n-4k)$$

$$T(n) = T(n-4i) + \log(\prod_{k=0}^{i-1} (n-4k))$$

$$T(n) = T(n-4i) + \log(\Theta(n^i))$$

Where i is the number of iterations. Ws can then solve for i by solving the equation $n - 4i \leq 5$, which gives $i \geq \frac{n-5}{4}$. The $T(n-4i)$ term, when $i \geq \frac{n-5}{4}$ is a constant (from the definition of $T(n)$). We can then write:

$$T(n) = c + \log(\Theta(n^{\frac{n-5}{4}})) \in \Theta(n \log(n))$$

Problem 4:

(a) Use the method of iterative substitution to solve the following recurrence.

$$T(n) = \begin{cases} 2 & \text{if } n \leq 7 \\ T(n^{1/3}) + \Theta(\log\log n) & \text{if } n > 7 \end{cases}$$

(b) Use mathematical induction to prove that your asymptotic solution for (a) is correct.

Solution 4:

(a) We can write out the first few iterations, assuming n is sufficiently large, as:

$$T(n) = T(n^{1/3}) + \Theta(\log\log(n))$$

$$T(n) = T(n^{1/9}) + \Theta(\log\log(n^{1/3})) + \Theta(\log\log(n))$$

$$T(n) = T(n^{1/27}) + \Theta(\log\log(n^{1/9}) + \Theta(\log\log(n^{1/3})) + \Theta(\log\log(n))$$

$$\cdots$$

$$T(n) = T(n^{1/3^i}) + \Theta(\log\log(n)) + \Theta(\log 1/3\log(n)) + ... + \Theta(\log 1/3^{i-1}\log(n))$$

$$T(n) = T(n^{1/3^i}) + \Theta(\log\log(n)) + \Theta(\log(1/3) + \log(n)) + ... + \Theta(\log(1/3^{i-1}) + \log(n))$$

$$T(n) = T(n^{1/3^i}) + \Theta(\log\log(n)) + \Theta(\log(n) - \log(3)) + ... + \Theta(\log(n) - \log(3^{i-1}))$$

$$T(n) = T(n^{1/3^i}) + \Theta(\log\log(n)) + \Theta(\log(n) - 1\cdot\log(3)) + \Theta(\log(n) - 2\cdot\log(3)) + ... + \Theta(log(n) - (i-1)\log(3))$$

$$T(n) = T(n^{1/3^i}) + \Theta(\log\log(n)) + \Theta(\log(n) - \log(3)\cdot\sum_{k=0}^{i-1} k)$$

This recursion stops when $n^{1/3^i} \leq 7 \Rightarrow i = \log_3(\log_7(n))$, for which $T(n^{1/3^i})$ is a constant. We can then write $i - 1 = \log_3(\log_7(n)) - 1 = \log_3(\log_7(n)) - \log_3(3) = \log_3(\frac{\log_7(n)}{3})$

(b)

Problem 5: Euclid's algorithm uses the following simple formula, which leads to the algorithm below.

Euclid's rule If x and y are positive integers with $x \geq y$, then $\gcd(x, y) = \gcd(x \mod y, y)$.

procedure EUCLID(a, b)
input: a, b are integers with $a \geq b \geq 0$
output: $\gcd(a, b)$

if $b = 0$: return a
return EUCLID(b,$a \mod b$)

Here, we will look at an alternative algorithm based on divide and conquer.

(a) Show that the following rule is true.

$$\gcd(a,b) = \begin{cases} 2\gcd(a/2, b/2) & \text{if } a, b \text{ are even} \\ \gcd(a, b/2) & \text{if } a \text{ is odd, } b \text{ is even} \\ \gcd((a-b)/2, b) & \text{if } a, b \text{ are odd} \end{cases}$$

(b) Give an efficient divide-and-conquer algorithm for greatest common divisor. (You can write it in plain English or use pseuodocode.) Argue the correctness and running time of your algorithm.

Solution 5:

(a)

a & b are even: If a and b are even, that means that they share a factor of (at least) 2 - so therefore, if we remove that factor from both (by dividing), then the greatest common divisor of a and b is 2 times that of the remaining numbers (that have been divided by 2). Therefore, this shows that if a and b are even, $gcd(a, b) = 2gcd(a/2, b/2)$.

a is odd & b is even: If one of a and b are even and the other odd, we can swap them such that the first is odd and the second is even, so without loss of generality, it is enough to consider if only the second is even. Since a is odd, it has no factors of two, so $gcd(a, b)$ must itself be odd. Therefore, we can remove as many 2's as necessary from b to make it odd, and so if a is odd and b is even, $gcd(a, b) = gcd(a, b/2)$

a & b are odd: Without loss of generality, assume that a is greater than b (otherwise, $a - b < 0$ - though not necessarily required, this makes the argument nicer). If a and b share a common factor, then we can write $a = \alpha \cdot n$ and $b = \beta \cdot n$ for some α and β. Then $a - b = (\alpha - \beta) \cdot n$, so the difference of a and b shares any common factor that a and b might have. Since a and b are both odd numbers, α and β must be odd, and their difference must therefore be even, so $\alpha - \beta$ must have a factor of 2. Therefore, for all odd a and b, $gcd(a, b) = gcd((a - b)/2, b)$.

(b) To implement this algorithmically, it is fairly simple. We need only add a few base cases and then this implemenation of gcd is almost ready to go. The first base case is when one of the numbers is 1 - for which the gcd is also 1. The other base case is when one of the numbers is 0 - in which case, the gcd is the other number itself. We can then write an updated version of the gcd above with base cases as follows (again assuming without loss of generality that a is greater than b).

$$\gcd(a,b) = \begin{cases} 0 & \text{if } b \text{ is 1 and } a \text{ is not} \\ 1 & \text{if } b \text{ is 0 and } a \text{ is not} \\ a & \text{if } a = b \\ 2\gcd(a/2, b/2) & \text{if } a, b \text{ are even} \\ \gcd(a, b/2) & \text{if } a \text{ is odd, } b \text{ is even} \\ \gcd((a-b)/2, b) & \text{if } a, b \text{ are odd} \end{cases}$$

We can then choose to implement this recursively or regularly (with a while loop), and both implementations are fairly similar. The recursive method goes something like this:

```
gcd(a,b) {
    a = max(a,b);
    b = min(a,b);
    if(a == b) {
        return a;
    else if(b == 0) {
        return 1;
    else if(b == 1) {
        return 0;
    }   if(a mod 2 === 0) {
        a /= 2;
        if(b mod 2 == 0) {
            b /= 2;
            return 2*gcd(a,b);
        } else {
            return gcd(a,b);
        }
    } else { if(b mod 2 == 0) {
            b /= 2;
            return gcd(a,b);
        } else {
            a = (a-b)/2;
            return gcd(a,b);
        }
    }
}
```

First, we check if the given inputs (a and b) satisfy one of the base cases. If so, the recursion stops. Otherwise, we follow the definition of gcd above and continue to iterate. Now, for the runtime analysis. Every time we call gcd, one, if not both, of the inputs decrease in size by a factor of 2. This means that we will iterate approximately $\log_2(n)$ times, where n is the greater of a and b. Each loop, we must divide by 2 once, and so we have a total of $O(\log_2(n))$ divisions. Depending on the implementation of the division in particular, this could lead to different runtimes for the algorithm as a whole.

Submission. To submit your homework, you need to upload the pdf file (one per group!) to Gradescope by 11:59pm of Friday, April 15^{th}.

Reminders. Remember that only LaTeX papers are accepted.