CSCE 411
Asymptotic Analysis Practice

---

1. Prove that $\frac{\sqrt{n}-\sqrt[3]{n}}{2} \in \Omega(\sqrt{n})$ using the formal definition of big-Ω.

Solution: To show that $\frac{\sqrt{n}-\sqrt[3]{n}}{2} \in \Omega(\sqrt{n})$ we need to choose positive constants c and n_0 such that $\forall\, n \geq n_0$, $\frac{\sqrt{n}-\sqrt[3]{n}}{2} \geq c \log^2 n$. Let $n_0 = 64$ (the smallest number after 1 which is a perfect square and a perfect cube) and $c = 1/4$. Then for all $n \geq n_0$ we have

$$\begin{aligned}
\frac{\sqrt{n}-\sqrt[3]{n}}{2} &= \frac{n^{1/2} - n^{1/2}n^{-1/6}}{2} \\
&= \frac{n^{1/2}(1 - 1/n^{1/6})}{2} \\
&\geq \frac{n^{1/2}(1 - 1/64^{1/6})}{2} \\
&= \frac{n^{1/2}(1 - 1/2)}{2} \\
&= \frac{n^{1/2}}{4} \\
&= c\sqrt{n}
\end{aligned}$$

2. Analyze the runtime of this algorithm and give a big-Θ bound (in terms of n).

```
ShiftySort(A[1, ..., n]):
    For i from 1 to n − 1:
        For j from n − i to n − 1:
            If A[j] > A[j + 1]:
                Swap A[j] with A[j + 1]
            Else:
                Break
```

Solution: As always in this course, when we talk about the "runtime" of an algorithm without another qualifier, we mean the worst-case runtime. Let $T(n)$ be the runtime of this algorithm. We will prove that $T(n) \in \Theta(n^2)$. To prove this we need to show an upper bound and a lower bound. That is, we need to prove that $T(n) \in O(n^2)$ and $T(n) \in \Omega(n^2)$ (as we proved in the notes, showing both of these is enough to conclude that $T(n) \in \Theta(n^2)$).

- $T(n) \in O(n^2)$: We note that the outer for loop runs at most n times and the inner for loop runs at most n times per iteration of the outer for loop, so there are at most n^2 total iterations of the inner for loop. The steps inside of the inner for loop is all constant time, therefore the total number of steps taken by the algorithm is $O(n^2)$.
- $T(n) \in \Omega(n^2)$: To show that the worst case runtime is $\Omega(n^2)$, we need to find some class of inputs on which the algorithm runs in $\Omega(n^2)$ time. We note that if the input array is is in reverse sorted order (i.e. sorted from largest to smallest) then in iteration i of the outer for loop, when $A[n-i]$ is accessed for the first time, it will be larger than all of the previously considered elements (which are in $A[n-i+1, \ldots, n]$), so the algorithm will swap $A[n-i]$ all the way to position n, meaning that the inner for loop will run the full i times.

  Thus, on a reverse sorted array, the total number of iterations of the inner loop is $\sum_{i=1}^{n-1} i = (n-1)n/2 \in \Omega(n^2)$, so the worst case runtime is $\Omega(n^2)$.

3. Suppose that f, g, f', and g' are all functions from $\mathbb{Z}^+$ to $\mathbb{Z}^+$ with $f'(n) \in O(f(n))$ and $g'(n) \in O(g(n))$. For each of the following statements either prove it, or disprove it by giving a counterexample:

   (a) $\max(f'(n), g'(n)) \in O(\max(f(n), g(n)))$

   (b) $f'(n)^{g'(n)} \in O(f(n)^{g(n)})$

   (c) $f'(n)/g'(n) \in O(f(n)/g(n))$

   (d) $f'(g'(n)) \in O(f(g(n))$

Solution: (a) This is true. Let Let c_f and c_g be positive real numbers and n_f and n_g be positive real numbers such that $\forall\, n \geq n_f$, $f'(n) \leq c_f f(n)$ and $\forall\, n \geq n_g$, $g'(n) \leq c_g g(n)$ (such constants must exist by the definition of big-O). To show that $\max(f'(n), g'(n)) \in O(\max(f(n), g(n)))$ we choose the constants $c = 2(c_f + c_g)$ and $n_0 = \max(n_f, n_g)$. Then, by our choice of c_f, c_g, n_f, and n_g, we have that for every $n \geq n_0$

$$\begin{aligned}\max(f'(n), g'(n)) &\leq f'(n) + g'(n) \\ &\leq c_f f(n) + c_g g(n) \\ &\leq (c_f + c_g)(f(n) + g(n)) \\ &\leq (c_f + c_g)(2\max(f(n), g(n))) \\ &= c\max(f(n), g(n)).\end{aligned}$$

So by the definition of big-O, $\max(f'(n), g'(n)) \in O(\max(f(n), g(n)))$.

(b) This is false. Consider $f(n) = n$, $f'(n) = n$, $g(n) = 2$, $g'(n) = 3$. Then $3 \in O(2)$, and $n \in O(n)$, but $n^3 \notin O(n^2)$.

(c) This is false. Consider $f(n) = n$, $f'(n) = n$, $g(n) = n$ and $g'(n) = 1$. Then $n \in O(n)$ and $1 \in O(n)$, but $n/1 \notin O(n/n)$.

(d) This is false. Consider $f'(n) = 2^n$, $f(n) = 2^n$, $g'(n) = 2n$, $g(n) = n$. Then $f'(g'(n)) = 2^{2n} = 4^n$, while $f(g(n)) = 2^n$, so $f'(n) \in O(f(n))$ and $g'(n) \in O(g(n))$, but $f'(g'(n)) \notin O(f(g(n))$.

4. Let $f : \mathbb{Z}^+ \to \mathbb{Z}^+$ and $g : \mathbb{Z}^+ \to \mathbb{Z}^+$ be two functions. Prove that if $f(n) = O(g(n))$ and $g(n) = O(f(n))$ then $f(n) = \Theta(g(n))$

Solution: This is true. Suppose $f(n) = O(g(n))$ and $g(n) = O(f(n))$. Let c_f, c_g, n_f, n_g be constants such that $\forall\, n \geq n_f$, $f'(n) \leq c_f f(n)$ and $\forall\, n \geq n_g$, $g'(n) \geq c_g g(n)$ (such constants must exist by the definition of big-O). We choose $n_0 = \max(n_f, n_g)$, $c_1 = 1/c_g$ and $c_2 = c_f$. Then we have that for every $n \geq n_0$

$$c_1 g(n) \leq f(n) \leq c_2 g(n),$$

so $f(n) = \Theta(g(n))$, by definition.

5. Analyze the runtime of this algorithm and give a big-Θ bound in terms of the input length n. Assume that x and y are given in binary (How long do the arithmetic operations inside the loop take?)

```
RPM(x, y)
  r ← 0
  a ← x
  b ← y
  While b > 0:
      If b mod 2 = 1 :
          r ← r + a
      a ← 2a
      b ← ⌊b/2⌋
  Return r
```

Solution: To start we note that when a number x is written in binary, the length is $\lfloor \log_2 x + 1 \rfloor \in \Theta(\log x)$. Thus, the length of our input is $n \in \Theta(\log x + \log y)$.

We make the following observations:

- Initially $b = y$
- Each iteration of the while loop, b halves (rounded down)
- The loop ends when $b = 0$

Therefore, the while loop runs for $\Theta(\log y)$ iterations.

Inside the loop we are doing arithmetic on r, a, and b. We note that a initially has $\Theta(\log x)$ digits, and that each time we double it the number of binary digits increases by 1, so after $\Theta(\log y)$ doublings it has $\Theta(\log x + \log y)$ digits. Similarly, r has $\Theta(\log x + \log y)$ digits by the end of the while loop. b initially has has $\Theta(\log y)$ digits and only decreases during the loop.

We can perform arithmetic in constant time if the numbers we are working with are at most $2^{cw} = 2^{c \log(n)}$ for some constant c. A number with value at most $2^{c \log(n)}$ has $O(\log(n))$ digits. The numbers that we are adding and multiplying have $\Theta(\log x + \log y) = \Theta(n)$ digits, so we *cannot* preform this arithmetic using a constant number of RAM model instructions.

Instead we can perform the addition $r \leftarrow r + a$ by looping through the digits and adding the corresponding digits of r and a, keeping track of a carry as necessary. This approach takes time linear in the length of the numbers that we're adding. Since we have already concluded that r and a have $O(\log x + \log y)$ digits and $\Omega(\log x)$ digits (except for the very first loop iteration where r is 0), the time to perform this addition is $O(\log x + \log y)$ and $\Omega(\log x)$. Since our numbers are written in binary, we can perform $a \leftarrow 2a$ by simply adding a 0 to the end of a. This can be done in $O(\log x + \log y)$ time (at most we need to shift over all of the digits of a). Similarly, $b \leftarrow \lfloor b/2 \rfloor$ is just removing the last digit of b (which again may require shifting the remaining digits to fill in the gap). Since b has $O(\log y)$ digits, we can perform $b \leftarrow \lfloor b/2 \rfloor$ in time $O(\log y)$. Thus, the total runtime for each iteration of the loop is $O(\log x + \log y)$ and $\Omega(\log x)$.

Since the loop runs $\Theta(\log y)$ times, the total runtime is $O((\log y)(\log x + \log y)) = O(n^2)$ (since $n \in \Theta(\log x + \log y)$) and $\Omega((\log y)(\log x))$. If x and y each have $n/2$ digits, then we have that $\Omega((\log y)(\log x)) = \Omega(n^2)$, so in the worst case our lower bound matches our upper bound and the algorithm runs in $\Theta(n^2)$ time.