CMPS 6610 Problem Set 01

Lei Yang

August 31, 2023

1 Asymptotic notation

1a. True, since $2^{n+1} = 2 \times 2^n$. Pick $c \geq 2$ and $n = 0$. We get $2 \times 2^n = 2^{n+1}$ for all $n > 0$. Hence, $2^{n+1} \in O(2^n)$.

1b. False, $\lim_{n\to\infty} \frac{2^{2^n}}{2^n}$ approaches infinity. Thus, we cannot find a constant c that satisfies the requirement for sufficiently large n.

1c. False, $\lim_{n\to\infty} \frac{n^{1.01}}{\log n^2}$ approaches infinity. Thus, we cannot find a constant c that satisfies the requirement for sufficiently large n.

1d. True, as $n^{1.01}$ grows faster than $\log^2 n$. Pick $c \geq 1$ and $n = 1$. We get $n^{1.01} \geq \log^2 n$ for all $n \geq 1$. Hence, $n^{1.01} \in \Omega(\log^2 n)$.

1e. False, $\lim_{n\to\infty} \frac{\sqrt{n}}{\log^3 n}$ approaches infinity. Thus, we cannot find a constant c that satisfies the requirement for sufficiently large n.

1f. True, as $\sqrt{n}$ grows faster than $\log^3 n$. Pick $c \geq 1$ and $n = 1$. We get $\sqrt{n} \geq \log^3 n$ for all $n \geq 1$. Hence, $\sqrt{n} \in \Omega(\log^3 n)$.

1g. Let's assume that there exists some function $f(n)$ that belongs to both $o(g(n))$ and $\omega(g(n))$.

(a) For $o(g(n))$: For every positive constant c_1, there exists n_{01} such that $f(n) \leq c_1 \times g(n)$ for all $n \geq n_{01}$.

(b) For $\omega(g(n))$: For every positive constant c_2, there exists n_{02} such that $f(n) \geq c_2 \times g(n)$ for all $n \geq n_{02}$.

Let c_1 and c_2 be any two positive constants. Now, let $n_{\max} = \max(n_{01}, n_{02})$. For $n \geq n_{\max}$, both conditions must hold:

(a) $f(n)/g(n) \leq c_1$

(b) $f(n)/g(n) \geq c_2$

This leads to a contradiction. Therefore, our initial assumption is false, and $o(g(n)) \cap \omega(g(n)) = \emptyset$.

2 SPARC to Python

1. The function `foo` calculates the n-th Fibonacci number using recursion. For inputs 0 and 1, it returns the value itself. Otherwise, it calculates the sum of the $(x-1)$-th and $(x-2)$-th Fibonacci numbers by recursively calling itself.

3 Parallelism and recursion

1. Work: Since we explore each element of the array at least once, the work done is $O(n)$, where n is the length of the array. Span: The algorithm is implemented in an iterative, sequential way, so its span is $O(1)$.

2. Work: $O(n)$
   Span: At each level of recursion, the algorithm splits the problem into two independent subproblems. Thus, the depth of the recursion tree is $\log n$, and the span is $O(\log n)$.

3. Work: The algorithm explores each element of the array at least once, making the work $O(n)$, where n is the length of the array.
   Span: At each level of recursion, the algorithm splits the problem into two independent subproblems. Thus, the depth of the recursion tree is $\log n$, and the span is $O(\log n)$.