CMPS 2200 Assignment 1

Name:_______________________________________

In this assignment, you will learn more about asymptotic notation, parallelism, functional languages, and algorithmic cost models. As in the recitation, some of your answer will go here and some will go in `main.py`. You are welcome to edit this `assignment-01.md` file directly, or print and fill in by hand. If you do the latter, please scan to a file `assignment-01.pdf` and push to your github repository.

1. (2 pts ea) **Asymptotic notation**

- 1a. Is $2^{n+1} \in O(2^n)$? Why or why not? .
  .
  .
  .
  .
- 1b. Is $2^{2^n} \in O(2^n)$? Why or why not?
  .
  .
  .
  .
  .
- 1c. Is $n^{1.01} \in O(\log^2 n)$?
  .
  .
  .
  .
- 1d. Is $n^{1.01} \in \Omega(\log^2 n)$?
  .
  .
  .
  .
- 1e. Is $\sqrt{n} \in O((\log n)^3)$?
  .
  .
  .
  .
- 1f. Is $\sqrt{n} \in \Omega((\log n)^3)$?
  .

2. **SPARC to Python**

Consider the following SPARC code of the Fibonacci sequence, which is the series of numbers where each number is the sum of the two preceding numbers. For example, 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610 ...

$$
\begin{array}{l}
\mathit{foo}\ x = \\
\quad \texttt{if}\ \ x \leq 1\ \ \texttt{then} \\
\quad\quad x \\
\quad \texttt{else} \\
\quad\quad \texttt{let}\ \ (ra, rb) = (\mathit{foo}\ (x-1))\ \ ,\ \ (\mathit{foo}\ (x-2))\ \ \texttt{in} \\
\quad\quad\quad ra + rb \\
\quad\quad \texttt{end}.
\end{array}
$$

- 2a. (6 pts) Translate this to Python code – fill in the `def foo` method in `main.py`
- 2b. (6 pts) What does this function do, in your own words?

.
.
.
.
.
.
.
.

3. **Parallelism and recursion**

Consider the following function:

```
def longest_run(myarray, key)
    """
     Input:
       `myarray`: a list of ints
       `key`: an int
     Return:
       the longest continuous sequence of `key` in `myarray`
    """
```

E.g., `longest_run([2,12,12,8,12,12,12,0,12,1], 12) == 3`

- 3a. (7 pts) First, implement an iterative, sequential version of `longest_run` in `main.py`.
- 3b. (4 pts) What is the Work and Span of this implementation?

.
.
.

.
.
.
.
.
.

- 3c. (7 pts) Next, implement a `longest_run_recursive`, a recursive, divide and conquer implementation. This is analogous to our implementation of `sum_list_recursive`. To do so, you will need to think about how to combine partial solutions from each recursive call. Make use of the provided class `Result`.

- 3d. (4 pts) What is the Work and Span of this sequential algorithm?

  .
  .
  .
  .
  .
  .
  .
  .
  .
  .
  .

- 3e. (4 pts) Assume that we parallelize in a similar way we did with `sum_list_recursive`. That is, each recursive call spawns a new thread. What is the Work and Span of this algorithm?

.
.
.
.
.
.
.
.

Thứ
Ngày No

Intro to Algorithm

Assignment 1

①

1a. $2^{n+1} \in O(2^n)$

Because by definition, $f(n) \in O(g(n))$ if there exist constants $c > 0$ and n_0 such that $f(n) \le cg(n)$ for all $n \ge n_0$.

Let $f(n) = 2^{n+1}$ and $g(n) = 2^n$. Since $2^{n+1} = 2.2^n$, choose $c = 2$ and $n_0 = 0$ (any $n_0 \ge 0$ works). Then for all $n \ge n_0$,

$$2^{n+1} \le 2.2^n = cg(n)$$

Hence $2^{n+1} \in O(2^n)$.

1b. No. $2^{2^n} \notin O(2^n)$

Let $f(n) = 2^{2^n}$ and $g(n) = 2^n$. We consider the following quotient:

$$\frac{f(n)}{g(n)} = \frac{2^{2^n}}{2^n} = 2^{2^n - n}$$

Since $2^n - n \to \infty$, we have $2^{2^n - n} \to \infty$

Therefore the ratio $\frac{f(n)}{g(n)}$ is unbounded. By the definition of Big-O, there is no constant $c > 0$ and n_0 such that $f(n) \le cg(n)$ for all $n \ge n_0$.

Equivalently, assuming $2^{2^n} \le c2^n$ and taking $\log_2$ gives $2^n \le n + \log_2 c$, which fails for large n.

Hence 2^{2^n} grows strictly faster than 2^n, so it is not $O(2^n)$.

1.c. No. $n^{1.01} \notin O((\log n)^2)$.

Let $f(n) = n^{1.01}$ and $g(n) = (\log n)^2$ where the log base

Thứ . .
Ngày . . No.

is arbitrary since bases differ by a constant factor.

Consider:

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = \lim_{n\to\infty}\frac{n^{1.01}}{(\log n)^2}$$

Set $n = e^t$. Then the ratio becomes:

$$\frac{e^{1.01t}}{t^2} \xrightarrow[t\to\infty]{} \infty,$$

since an exponential dominates any polynominal. Hence $\frac{f(n)}{g(n)}$ is unbounded, so there are no constants $c > 0$ and n_0 with $n^{1.01} \le c(\log n)^2$ for all $n \geq n_0$.

Therefore $n^{1.01} = \omega((\log n)^2)$, not $O((\log n)^2)$.

1d. Yes. $n^{1.01} \in \Omega((\log n)^2)$.

By definition, $f(n) \in \Omega(g(n))$ if $\exists c > 0$ and n_0 such that $f(n) \geq c\,g(n)$ for all $n \geq n_0$.

Let $f(n) = n^{1.01}$ and $g(n) = (\log n)^2$ (with any log base; bases differ by a constant factor). We consider the fraction:

$$\lim_{n\to\infty}\frac{(\log n)^2}{n^{1.01}}$$

Set $n = e^t$. Then:

$$\frac{(\log n)^2}{n^{1.01}} = \frac{t^2}{e^{1.01t}} \longrightarrow 0 \; (t \to \infty),$$ since an exponential dominates any polynominal. Hence there exists n_0 with:

$$\frac{\log(n)^2}{n^{1.01}} \le 1 \text{ for all } n \geq n_0,$$

which implies $n^{1.01} \geq (\log n)^2$ for all $n \geq n_0$.

VIBOOK INTERNATIONAL

Taking $c = 1$ yields $n^{1.01} \in \Omega((\log n)^2)$. In fact, ~~$n^{1.01}$ -~~ $n^{1.01} = \omega((\log n)^2)$.

1e. No. $\sqrt{n} \notin O((\log n)^3)$.

To be in O, we would need constants $c > 0$ and n_0 with

$$\sqrt{n} \le c(\log n)^3 \text{ for all } n \geq n_0.$$

Let $n = e^t$. The inequality becomes $e^{t/2} \le ct^3$. We consider the ratio:

$$\frac{e^{t/2}}{t^3} \xrightarrow[t \to \infty]{} \infty,$$

so no such constant c can work. Hence $\sqrt{n}$ grows faster than any polylogarithm, in particular faster than $(\log n)^3$.

1f. Yes. $\sqrt{n} \in \Omega((\log n)^3)$.

Definition: $f(n) \in \Omega(g(n))$ if $\exists c > 0$ and n_0 such that $f(n) \geq c\, g(n)$ for all $n \geq n_0$.

Consider the fraction:

$$\frac{(\log n)^3}{\sqrt{n}}$$

Let $n = e^t$. Then the ratio becomes $\frac{t^3}{e^{t/2}} \to 0$ as $t \to \infty$.

Hence for some n_0,

$$\frac{(\log n)^3}{\sqrt{n}} \le 1 \text{ for all } n \geq n_0,$$

which is equivalent to $\sqrt{n} \geq (\log n)^3$ for all $n \geq n_0$.

Taking $c = 1$ satisfies the definition. Therefore $\sqrt{n} \in \Omega((\log n)^3)$ and, in fact, $\sqrt{n} = \omega((\log n)^3)$.

Thứ . .

Ngày . . No.

②

b. foo(x) returns the x-th Fibonacci number. It uses the standard recurrence

$F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$.

The base case returns x when $x \leq 1$. Otherwise, it computes the two previors Fibonacci numbers and add them up.

③

3b. • Work: $\Theta(n)$. We scan the list once with $O(1)$ work per element.

• Span: $\Theta(n)$. The computation is sequential.

3d. • Work: $\Theta(n)$ Recurrence $T(n) = 2T(n/2) + \Theta(1)$ gives linear work by the Master Theorem.

• Span: Θ. Calls run one after another in a single thread, so span equals total work.

3e. Work and span if we parallelize the two recursive calls

• Work: $\Theta(n)$. Parallelism does not change the total amount of work.

• Span: $\Theta(\log n)$. Critical path satisfies $S(n) = S(n/2) + \Theta(1)$ since the two halves run in parallel.