CMPS 6610 Problem Set 1 Solutions

1. **Asymptotic notation**

- 1a. Is $2^{n+1} \in O(2^n)$? Why or why not?

  True since $2^{n+1} = 2 * 2^n$, pick $c \geq 2$ and $n_0 = 0$

- 1b. Is $2^{2^n} \in O(2^n)$? Why or why not?

  False. We've shown that any polylogarithmic function grows slower than any polynomial. That is, $\log^i n \in O(n^j) \ \ \forall i, j > 0$ and vice versa: $n^j \in \Omega(\log^i n) \ \ \forall j, i > 0$. Here we can change variables (i.e., $m = 2^n$) and take logarithms to apply this "lemma". The problems below can be worked in the same manner. .

  .

  .

  .

- 1c. Is $n^{1.01} \in O(\log^2 n)$?

  No, see above. .

  .

  .

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

  Yes, see above.

.

.

.

- 1e. Is $\sqrt{n} \in O((\log n)^3)$?

  No, see above.

.

.

.

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

  Yes, see above.

.

.

.

- 1g. Consider the definition of "Little o" notation:

  $g(n) \in o(f(n))$ means that for **every** positive constant c, there exists a constant n_0 such that $g(n) \leq c \cdot f(n)$ for all $n \geq n_0$. There is an analogous definition for "little omega" $\omega(f(n))$. The distinction between $o(f(n))$ and $O(f(n))$ is that the former requires the condition to be met for **every** c, not just for some c. For example, $10x \in o(x^2)$, but $10x^2 \notin o(x^2)$.

Prove that $o(g(n)) \cap \omega(g(n))$ is the empty set.

By definition, we know that for any $c_1 > 0$, $c_2 > 0$, there exists $n_1 > 0$ such that $f(n) < c_1 g(n)$; and also that there exists $n_2 > 0$ such that $c_2 g(n) < f(n)$

If we pick $n_0 = \max(n_1, n_2)$, and $c_1 = c_2$, then we have that $c_1 g(n) < f(n) < c_1 g(n)$. There can be no $f(n)$ that satisfies this inequality. .

.
.

2. **SPARC to Python**

Consider the following SPARC code:

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

- 2a. Translate this to Python code – fill in the `def foo` method in `main.py`

```
def foo(x):
    if x <= 1:
        return x
    else:
        return foo(x-1) + foo(x-2)
```

- 2b. What does this function do, in your own words?

  It generates the xth element in the Fibonacci sequence.

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`

- 4a. First, implement an iterative, sequential version of `longest_run` in `main.py`.

  This should just be a single for loop that keeps track of the longest run seen so far. Whenever `key` matches the next element, increment total count. If it doesn't match, reset to 0. If the total count is greater than the longest seen so far, update the longest seen so far.

- 4b. What is the Work and Span of this implementation?

  $O(n)$ for both, since there is no parallelism being used.

.
.

- 4c. Next, implement a `longest_run_recursive` , a recursive, divide and conquer implementation. This is analogous to our implementation of `sum_list_recursive` (this should have been `parallel_sum_list` from Module 1). 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`.

We split the list in half and solve each recursively. For the base case, the list length is 1. We either return Result(1, 1, 1, True) or Result(0, 0, 0, False), depending on if the input is equal to the key. The main difficulty is in combining the results of the two recursive calls. To do so, we need to take the Result objects from each recursive call, and combine them together to make a new Result object. To do so, we have to take the max of the left side and right side. But, we also have to consider cases where the longest sequence spans the two sides. E.g., if the key is `12` and the two sides are `[6 12] [12 6] -> [6 12 12 6]`. Here, the longest sequence of 12s requires adding together `result1.right_side` and `result2.left_side`.

I'd recommend walking through a number of examples to work out the logic. A good test case to check is `assert to_value(longest_run_recursive([6, 12, 12, 12, 12, 6, 6, 6], 12)) == 4`

- 4d. What is the Work and Span of this sequential algorithm?

  Since we split into two problems, each half the original size, and it takes constant time to merge the results, we get work:

$W(n) = 2W(n/2) + 1$

We can solve with the brick method – it is leaf dominated. Height of tree is $\lg n$, number of leaves is $2^{\lg n}$, so $O(n)$.

Span is same as above, since it is *sequential*.

.
.
.
.
.

- 4e. Assume that we parallelize in a similar way we did with `sum_list_recursive` (this should have been `parallel_sum_list` from Module 1). That is, each recursive call spawns a new thread. What is the Work and Span of this algorithm?

The work is same as above. For the span, notice that we can count the presence of the key on the left and right sides in the base case and thus we don't need to ever do any "scanning" of the key at the end or beginning of any sub-lists.

Since the recursive calls can be executed in parallel, the span recurrence is

$S(n) = S(n/2) + 1$

Now, using the brick method, this is *balanced*. The first level has a cost of 1 and the number of levels is $\lg n$. Multiplying those together, we get $O(\lg n)$.

.
.
.
.
.
.