Homework 01 — Solution

CS 624, 2024 Spring

1. Consider the following algorithm for calculating a number raised to a power. The input is a real number x and a nonnegative integer n. The output is a real number r.

Algorithm 1 Power(x, n)

```
r ← 1; y ← x; p ← n
while p > 0 do
  if p is odd then
    r ← r × y
  end if
  y ← y × y
  p ← ⌊p/2⌋
end while
return r
```

The correctness property for this algorithm is the following:

$$r = x^n$$

(a) State the loop invariant for the **while** loop.

The loop invariant is
$$r \cdot y^p = x^n$$

(b) Prove the correctness of the algorithm using the loop invariant.

Initialization: At the beginning of the loop, $r = 1$, $y = x$, and $p = n$, so

$$r \cdot y^p = 1 \cdot x^n = x^n$$

So the loop invariant holds at the beginning.

Maintanence: Assume the loop invariant holds at the beginning of the loop; we must show that the execution of the loop body makes the loop invariant hold for the "next" values of the variables.

Let r_0, y_0, p_0 refer to the values of the variables at the beginning of the loop iteration, and let r, y, p refer to the values of the variables at the end of the iteration. Then

$$\begin{aligned} r &= \begin{cases} r_0 y_0 & \text{if } p_0 \text{ is odd} \\ r_0 & \text{otherwise} \end{cases} \\ y &= y_0^2 \\ p &= \left\lfloor \frac{p_0}{2} \right\rfloor \end{aligned}$$

There are two cases: p_0 is odd, and p_0 is even.

- If p_0 is odd, then $p_0 = 2p + 1$ and $r = r_0 y_0$ (the odd branch). Then:

$$\begin{aligned}
x^n &= r_0 y_0^{p_0} && \text{(LI)} \\
&= r_0 y_0^{2p+1} && \text{(because } p_0 = 2p+1) \\
&= r_0 y_0 (y_0^2)^p && \text{(rewrite exponent)} \\
&= r (y_0^2)^p && \text{(because } r = r_0 y_0) \\
&= r y^p && \text{(because } r = r_0 y_0)
\end{aligned}$$

- If p_0 is even, then $p_0 = 2p$ and $r = r_0$. Then:

$$\begin{aligned}
x^n &= r_0 y_0^{p_0} && \text{(LI)} \\
&= r_0 y_0^{2p} && \text{(because } p_0 = 2p) \\
&= r_0 (y_0^2)^p && \text{(rewrite exponent)} \\
&= r_0 y^p && \text{(because } y = y_0^2) \\
&= r y^p && \text{(because } r = r_0)
\end{aligned}$$

So in either case, the loop invariant holds for the new values of the variables.

Termination: When the loop exits, we know $p = 0$. The loop invariant is $ry^p = x^n$. With $p = 0$, that simplifies to $r = x^n$, which is the goal.

(c) What is the running time of this algorithm? Justify your answer.

The **while** loop halves p each iteration, and p is initially n, so it takes up to $1 + \log_2 n$ iterations. The amount of work done within the loop is bounded above and below by constants. So the running time is $\Theta(\log n)$.

2. Consider the following algorithm for calculating the *cumulative sums* of an array. The input is an array of numbers, A. The output is a new array of number of the same length, R. (Array indexes start at 1.)

Algorithm 2 CumulativeSums(A)

```
R ← new array(length[A])
if length[A] > 0 then
    R[1] ← A[1]
end if
for j ← 2 to length[A] do
    R[j] ← R[j − 1] + A[j]
end for
return R
```

The correctness property for this algorithm is the following:

$$R[n] = \sum_{i=1}^{n} A[i] \qquad \text{for all } 1 \leq n \leq \mathit{length}[A]$$

(a) State the loop invariant for the **for** loop.

The loop invariant for iteration j is the following:

$$R[n] = \sum_{i=1}^{n} A[i] \qquad \text{for all } 1 \leq n < j$$

or, equivalently,

$$R[n] = \sum_{i=1}^{n} A[i] \qquad \text{for all } 1 \le n \le j-1$$

(b) Prove the correctness of the algorithm using the loop invariant.

If the input array A is empty, then R is empty, the **if** and **for** bodies never execute, and the result is trivially correct. The rest of the proof assumes $\mathit{length}(A) \ge 1$.

Initialization: We must show that the loop invariant holds at the beginning of the first iteration ($j = 2$). The range for n in the loop invariant is $\{1 \mathrel{..} 2-1\} = \{1\}$, and $R[1] = \sum_{i=1}^{1} A[i] = A[1]$ because of the assignment on the third line of the function body.

Maintenance: Assuming the loop invariant holds for j at the beginning of an iteration, we must show the invariant holds for $j+1$ at the end of the iteration.

Based on the loop body's assignment:

$$\begin{aligned} R[j] &= R[j-1] + A[j] && \text{(by line 6 of the algorithm)} \\ &= \sum_{i=1}^{j-1} A[i] + A[j] && \text{(by LI)} \\ &= \sum_{i=1}^{j} A[i] && \text{(absorb term into summation)} \end{aligned}$$

No other array slots are assigned, so the loop invariant equation still holds for $1 \le n \le j-1$ and the assignment extends it to $1 \le n \le j$ (the LI's range for the next value of j).

Termination: When the loop exits, the loop invariant holds for $j = \mathit{length}(A) + 1$, which simplifies to the desired correctness property.

(c) What is the running time of this algorithm? Justify your answer.

The running time is $\Theta(n)$ where $n = \mathit{length}(A)$.

The **for** loop body executes $n-1$ times and does a constant amount of work each time. There is additional work bounded above and below by constants, but that gets dominated by the linear term.

3. Problem 3-4 (a, b, c, d) in the textbook (page 62).

Let f and g be asymptotically positive functions. Prove or disprove each of the following:

(a) $f(n) = O(g(n))$ implies $g(n) = O(f(n))$.

False. Here is one counter-example: Let $f(n) = n$ and $g(n) = n^2$. We know that $f = O(g)$ but $g \ne O(f)$.

(b) $f(n) + g(n) = \Theta(\min(f(n), g(n)))$.

False. Here is a counter-example: Let $f(n) = 1$ and $g(n) = n$. Then for $n \ge 1$, $\min(f(n), g(n)) = f(n) = 1$, which cannot bound $n+1$ above.

(c) $f(n) = O(g(n))$ implies $\lg(f(n)) = O(\lg(g(n)))$, where $\lg(g(n)) \ge 1$ and $f(n) \ge 1$ for all sufficiently large n.

Proof: Since $f(n) = O(g(n))$, there are c_1 and n_1 such that for all $n \geq n_1$, $f(n) \leq c_1 g(n)$. The lg function is monotone: that is,

$$x \leq y \implies \lg x \leq \lg y$$

Therefore:

$$\lg(f(n)) \leq \lg(c_1 g(n)) = \lg(c_1) + \lg(g(n))$$

We are guaranteed that for "sufficiently large n" (call it $n \geq n_2$), $\lg(g(n)) \geq 1$, and thus $\lg(c_1) \leq \lg(c_1)\lg(g(n))$. So resuming:

$$\begin{aligned}
\lg(f(n)) &\leq \lg(c_1) + \lg(g(n)) \\
&\leq \lg(c_1)\lg(g(n)) + \lg(g(n)) \\
&= (1 + \lg(c_1))\lg(g(n))
\end{aligned}$$

So we choose $n_0 = \max(n_1, n_2)$ and $c = 1 + \lg(c_1)$.

(d) $f(n) = O(g(n))$ implies $2^{f(n)} \in O(2^{g(n)})$.

False. Here is a counter-example: Let $f(n) = 2n$ and $g(n) = n$.
But $2^{2n} \neq O(2^n)$. Suppose it were; then there would be c such that

$$\begin{aligned}
2^{2n} &\leq c2^n \\
2^{2n}/2^n &\leq c \\
2^n &\leq c
\end{aligned}$$

That is, the "constant" c would have to be larger than 2^n for all sufficiently large n, which is impossible.

4. Problem 4-1 (a, b, f, g) in the textbook (page 107).

Give asymptotic upper and lower bounds for $T(n)$ in each of the following recurrences. Assume that $T(n)$ is constant for $n \leq 2$. Make your bounds as tight as possible, and justify your answers.

(a) $T(n) = 2T(n/2) + n^4$.

Apply the master theorem, with $p = \log_2 2 = 1$. Then $n^4 = \Omega(n^{p+\epsilon})$ with $\epsilon = 3$. So the theorem tells us that $T(n) = \Theta(n^4)$.

(b) $T(n) = T(7n/10) + n$.

Apply the master theorem, with $p = \log_{10/7} 1 = 0$. Then $n = \Omega(n^{p+\epsilon})$ with $\epsilon = 1$. So the theorem tells us that $T(n) = \Theta(n)$.

(f) $T(n) = 2T(n/4) + \sqrt{n}$.

Apply the master theorem, with $p = \log_4 2 = \frac{1}{2}$. Then $\sqrt{n} = \Theta(n^{\frac{1}{2}})$. So the theorem tells us that $T(n) = \Theta(n^{\frac{1}{2}} \lg n)$.

(g) $T(n) = T(n-2) + n^2$.

$T(n) = \Theta(n^3)$.

For simplicity, let's assume n is even. Then

$$\begin{aligned}
T(n) &= \sum_{k=1}^{n/2} (2k)^2 \\
&= 4 \sum_{k=1}^{n/2} k^2 \\
&= \frac{4}{6}\left(\frac{n}{2}\right)\left(\frac{n}{2}+1\right)(n+1) && \text{(equation A.3, p1147)} \\
&= \frac{1}{6}\left(n^3 + 3n^2 + 2n\right)
\end{aligned}$$

We get a cubic polynomial whose n^3 coefficient is positive, so $T(n) = \Theta(n^3)$.

(Alternative answer)

$T(n) = \Theta(n^3)$. Another way is to guess and use induction to prove the bounds. For the upper bound, show $T(n) \le cn^3$, and for the lower bound show $T(n) \ge c_1 n^3 - c_2 n^2$.

5. Let a binary tree be either NIL or a node with left and right attributes whose values are also binary trees. Let the mindepth function be defined as follows:

$$\text{mindepth}(t) = \begin{cases} 0 & \text{if } t = \text{NIL} \\ 1 + \min(\text{mindepth}(\text{left}(t)), \text{mindepth}(\text{right}(t))) & \text{otherwise} \end{cases}$$

and let the countnil function be defined as follows:

$$\text{countnil}(t) = \begin{cases} 1 & \text{if } t = \text{NIL} \\ \text{countnil}(\text{left}(t)) + \text{countnil}(\text{right}(t)) & \text{otherwise} \end{cases}$$

Prove the following: If $\text{mindepth}(t) \ge n$, then $\text{countnil}(t) \ge 2^n$.

Hint: Use induction on n.

More precisely: $\forall n \in \mathbb{N},\ \forall t \in \textit{BinaryTree},\ \ \text{mindepth}(t) \ge n \Rightarrow \text{countnil}(t) \ge 2^n$.

Proof: by induction on n.

Base case ($n = 0$):
Goal: (for all t) if $\text{mindepth}(t) \ge 0$, then $\text{countnil}(t) \ge 2^0 = 1$.

It's obvious from the definition that $\text{countnil}(t) \ge 1$ for any tree, so the right side of the implication always holds. Done.

(Note: Similarly, $\text{mindepth}(t) \ge 0$ for every binary tree t.)

Inductive case (assume for $n = k$, prove for $n = k+1$):
The inductive hypothesis is: (for all t) if $\text{mindepth}(t) \ge k$, then $\text{countnil}(t) \ge 2^k$
Goal: (for all t) if $\text{mindepth}(t) \ge k+1$, then $\text{countnil}(t) \ge 2^{k+1}$.

Let t be an arbitrary binary tree, and assume that $\text{mindepth}(t) \ge k+1$.
We must show that $\text{countnil}(t) \ge 2^{k+1}$.

If $\text{mindepth}(t) \ge k+1$, then t cannot be NIL.
So we will use the non-NIL cases of the mindepth and countnil functions.

By case 2 of mindepth:

$$1 + \min(\text{mindepth}(\text{left}(t)), \text{mindepth}(\text{right}(t))) \ge k+1$$

Cancel out the (1+):

$$\min(\text{mindepth}(\text{left}(t)), \text{mindepth}(\text{right}(t))) \geq k$$

Facts about $\min(a, b)$: $a \geq \min(a, b)$ and $b \geq \min(a, b)$. So:

$$\begin{aligned} \text{mindepth}(\text{left}(t)) &\geq \min(\text{mindepth}(\text{left}(t)), \text{mindepth}(\text{right}(t))) \geq k \\ \text{mindepth}(\text{right}(t)) &\geq \min(\text{mindepth}(\text{left}(t)), \text{mindepth}(\text{right}(t))) \geq k \end{aligned}$$

Now we can apply the IH to the left and right children of t and get

$$\begin{aligned} \text{countnil}(\text{left}(t)) &\geq 2^k \\ \text{countnil}(\text{right}(t)) &\geq 2^k \end{aligned}$$

Now calculate

$$\begin{aligned} \text{countnil}(t) &= \text{countnil}(\text{left}(t)) + \text{countnil}(\text{right}(t)) \\ &\geq 2^k + 2^k = 2^{k+1} \end{aligned}$$

Done.