1 Announcements

- HW 2 is due next Monday. Keep in mind you only have half the time to complete this that you had to complete HW 1. It may not be half the length.
- Questions from lecture or homework?

2 Recurrences

Let’s first get some more practice with recurrences.

2.1 Example 1

\[ f(n) = f(n/4) + \Theta(\lg^2 n) \]

2.1.1 Brick Method.

At level \( i \):

<table>
<thead>
<tr>
<th>Problem Size</th>
<th>( n/4^i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node Cost</td>
<td>( \leq k_1 \lg^2(n/4^i) + k_2 )</td>
</tr>
<tr>
<td>Number of Nodes</td>
<td>1</td>
</tr>
</tbody>
</table>

This may look root-dominated: the level costs get smaller as we go down the tree. However, they don’t get smaller by a constant factor, since the \( 1/4^i \) is inside a logarithm. Thus, we are in a balanced situation and \( f(n) \) is \( O(d \cdot \lg^2 n) = O(\lg^3 n) \).
2.1.2 Tree Method.

From the chart above, we get this not-so-friendly looking summation:

\[ f(n) = k_1 \sum_{i=0}^{\lfloor \lg n \rfloor} (\lg^2(n/4^i)) + k_2 \lg n \]

Solving this summation exactly seems daunting, but we can make some headway using asymptotic approximations.¹ The second term, \( k_2 \lg n \), is clearly lower-order and so we can drop it. The highest-order term of the summand is going to be \( O(\lg^2 n) \), since \( \lg 4^i \) is a constant with respect to \( n \). Since we are summing \( \lg n \) terms, each on the order of \( \lg^2 n \), we can guess that the result will be \( \Theta(\lg^3 n) \). This doesn't seem sufficiently formal, so a good thing to do is check ourselves using the substitution method, now that we have a guess.

2.1.3 Substitution Method.

We will show that the solution to the recurrence is \( O(\lg^3 n) \). To do this, we wish to prove that there exist \( k_1, k_2 \) such that for all \( n > 1 \),

\[ f(n) \leq k_1 \lg^3 n + k_2 \]

**Base Case:** From the definition of \( \Theta \), we know that there exists \( c_1 > 0 \) such that \( f(1) \leq c_1 \). This proves the base case of our theorem as long as \( c_1 \leq k_2 \). Let’s remember that so we can choose an appropriate \( k_2 \).

**Inductive Case:** We know there exists \( c_2 > 0 \) such that

\[ f(n) \leq f(n/4) + c_2 \lg^2 n \]

Apply the induction hypothesis.

\[ f(n) \leq k_1 \lg^3 (n/4) + k_2 + c_2 \lg^2 n \]

\[ = k_1(\lg^3 n - 3 \lg 4 \lg^2 n + 3 \lg^2 4 \lg n - \lg^3 4) + k_2 + c_2 \lg^2 n \]

We rearrange this suggestively.

\[ f(n) \leq k_1 \lg^3 n + k_2 + k_1(-3 \lg 4 \lg^2 n + 3 \lg^2 4 \lg n - \lg^3 4) + c_2 \lg^2 n \]

We are done as long as

\[ k_1(-3 \lg 4 \lg^2 n + 3 \lg^2 4 \lg n - \lg^3 4) + c_2 \lg^2 n \leq 0 \]

¹If this seems a lot like the brick method, it is. The brick method is essentially a useful shortcut to the tree method that allows us to gain intuitive understanding of the behavior of the recurrence without explicitly solving the summation.
We need to find \( k_1 \) that makes this true, so let’s solve the above for \( k_1 \).

\[
k_1 \geq \frac{c_2 \lg^2 n}{3 \lg 4 \lg^2 n - 3 \lg^2 4 \lg n + \lg^3 4} = \frac{c_2}{3 \lg 4 - 3 \lg^2 4 \lg^{-1} n + \lg^3 4 \lg^{-2} n}
\]

We can satisfy these constraints by setting \( k_1 = c_2 \) and \( k_2 = c_1 \).

Thus, the recurrence is \( O(\lg^3 n) \). Indeed, it is also \( \Theta(\lg^3 n) \). We could show this by flipping around the theorem for the substitution method and proving that it is \( \Omega(\lg^3 n) \).

### 2.2 Example 2

\[
f(n) = 2f(\sqrt{n}) + \Theta(1)
\]

This is somewhat harder than the recurrences we’ve solved before, but let’s give it a try. Note that 1 won’t work as a base case here (why?), so we assume \( f(2) \in \Theta(1) \) as the base case.

#### 2.2.1 Brick Method.

This time, we can just draw the tree since it’s a familiar one:

```
+  
++  
++++  
++++++  
++++++  
++++++  
++++++  
++++++  
++++++  
++++++  
++++++  
++++++  
++++++  
++++++  
++++++  
++++++  
++++++  
++++++  
++++++  
++++++  
++++++  
++++++  
++++++  
++++++  
++++++  
++++++  
++++++  
++++++  
++++++  
++++++  
++++++  
++++++  
++++++  
++++++  
++++++  
```

The cost is dominated by the leaves, since the node cost is constant but the number of nodes grows exponentially. Thus, \( f(n) \) is \( O(\text{cost}_d) = O(2^d) \), where \( d \) is the number of levels in the tree.

But, how many levels are there? This is equivalent to asking how many times you can take the square root of a number before you get to 2.

\[
n^{1/2^i} = 2
\]

That’s a messy exponent, so let’s keep taking logs and hope it gets better.

\[
1/2^i \lg n = \lg 2
\]

\[
i \lg \frac{1}{2} + \lg \lg n = \lg \lg 2
\]

Noting that \( \lg 1/2 = -1 \) and \( \lg \lg 2 = 0 \), this gives

\[
i = \lg \lg n
\]

So, \( d \approx \lg \lg n \), and \( f(n) \in O(2^{\lg \lg n}) = O(\lg n) \).
2.2.2 Tree Method.

At level $i$:

<table>
<thead>
<tr>
<th>Problem Size</th>
<th>$n^{1/2^i}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node Cost</td>
<td>$\leq k$</td>
</tr>
<tr>
<td>Number of Nodes</td>
<td>$2^i$</td>
</tr>
</tbody>
</table>

Given the chart and our calculation of $d$, we can fairly easily get the summation

$$f(n) = k \sum_{i=0}^{\log \log n} 2^i$$

$$f(n) = k(2^{\log \log n + 1} - 1)$$

$$f(n) = k(2 \cdot 2^{\log \log n} - 1)$$

$$f(n) = k(2 \log n - 1) \in O(\log n)$$

2.2.3 Substitution Method.

We want to show that there exist $k_1, k_2$ such that

$$f(n) \leq k_1 \log n + k_2$$

**Base Case:** We know $f(2) \leq c_1$ for some $c_1$, so we need $c_1 \leq k_1 \log 2 + k_2 = k_1 + k_2$.

**Inductive Case:** Apply the inductive hypothesis to our assumption.

$$f(n) \leq 2(k_1 \log \sqrt{n} + k_2) + c_2$$

$$f(n) \leq 2\left(\frac{k_1}{2} \log n + k_2\right) + c_2$$

$$f(n) \leq k_1 \log n + 2k_2 + c_2$$

This works exactly if we set $k_2 = -c_2$. Does that work with our constraint from the base case?

$$c_1 \leq k_1 - c_2$$

$$k_1 \geq c_1 + c_2$$

We can satisfy this constraint by setting $k_1 = c_1 + c_2$, so this completes the proof.

Again, we have shown only that the recurrence is $O(\log n)$. We leave as an exercise the other direction required to show $\Theta(\log n)$. 
3 Scan

Yesterday, we covered the function \texttt{scan}. We'll recap the definition of \texttt{scan} briefly today, and show you how to solve interesting problems with it.

\texttt{scan} takes a function as one of its arguments. All of the text below makes the assumption that this function is \textit{associative}. Recall the mathematical definition that a function \( f \) is said to be associative if and only if

\[ \forall a \forall b \forall c. f(f(a, b), c) = f(a, f(b, c)) \]

We also make the assumption that the initial value is a \textit{left-identity} of the functional argument. Recall the mathematical definition that \( I \) is a left-identity of \( f \) if and only if

\[ \forall a. f(I, a) = a \]

We don't need these assumptions in general, and we'll come back to a version of \texttt{scan} later that doesn't have them, but it's a cleaner way to start thinking about \texttt{scan} with these properties.

With the assumption that \( f \) is associative, \((\text{scan } f \ b)\) is logically equivalent to \((\text{iterh } f \ b)\) in the same way that \((\text{reduce } f \ b)\) is logically equivalent to \((\text{iter } f \ b)\), but these functions differ in their span. Specifically, if \( f \) is a function that takes no more than a constant number of steps on all input, \((\text{iterh } f)\) and \((\text{iter } f)\) have both work and span in \( O(n) \), whereas \texttt{reduce} and \texttt{scan} both have work in \( O(n) \) and span in \( O(\lg n) \).

It's worth noting that while \texttt{reduce} and \texttt{scan} are highly parallel, unlike \texttt{iter} and \texttt{iterh}, they pay the price by having slightly less general types.

3.1 Note on Terminology

If \( f \) is a function and \( I \) is a relevant identity for \( f \), we'll often say “\( f \)-scan” to mean

\[ \texttt{scan } f \ I \]

For example, a “+\text{-scan}” is

\[ \texttt{scan} \ (\text{op +}) \ 0 \]

3.2 Recap

If \( s = \langle 1, 6, 3, -2, 9, 0, -4 \rangle \), then

\((\texttt{scan \ Int.min \ Int.maxInt \ s})\) yields the following:

\((\langle \text{Int.maxInt}, 1, 1, -2, -2, -2, -4 \rangle)\)

Remember that in the result, location \( i \) stores the “sum” of the values at locations \textbf{before} \( i \) in the original sequence. There is a variant of \texttt{scan} called \texttt{scanI} which sums the values at locations before and including \( i \).
3.3 Example Uses of Scan

At first glance, \texttt{scan} seems to offer not much that isn't already available through \texttt{reduce}. With clever choices of associative functions, though, \texttt{scan} can be used to compute some surprising things efficiently in parallel.

3.3.1 Histogram

Consider the following problem:

Given a histogram, if we were to pour water over it, how much water (in terms of area) would it hold? For simplicity we will represent a histogram as a sequence of non-negative integers. For example the histogram shown below is represented by the sequence $s = \langle 2, 3, 4, 7, 5, 2, 3, 2, 6, 4, 3, 5, 2, 1 \rangle$, and holds 15 units of water.

Any ideas on how we might solve this problem?

The idea is to single out one bar $b_i$. If we know the maximum of the bar heights to the left of $b_i$ ($\text{max}_l$) and the maximum of the bar heights to the right of $b_i$ ($\text{max}_r$), given that $\text{max}_l > \text{height}(b_i)$ and $\text{max}_r > \text{height}(b_i)$ then the water $b_i$ will hold above it is $\min(\text{max}_l, \text{max}_r) - \text{height}(b_i)$.

When confronted with a problem like this, a good technique is to divide up the problem into smaller subproblems, each of which can be easily solved with \texttt{scan}, \texttt{map} and/or \texttt{reduce}. Here's one way to divide it up, which directly follows the text in the previous paragraph.

1. For each bar $b_i$, calculate $\text{max}_l$ and $\text{max}_r$.
2. For each bar $b_i$, let $w_i = \min(\text{max}_l, \text{max}_r) - \text{height}(b_i)$ if $\text{max}_l > \text{height}(b_i)$ and $\text{max}_r > \text{height}(b_i)$, or $w_i = 0$ otherwise.
3. Sum all of the $w_i$.

Step 3 can be done with a \texttt{reduce}, and steps 1 and 2 can be done for each $b_i$ in parallel, but separately calculating, for example, $\text{max}_r$ for $b_i$ and $b_{i+1}$ will redo a lot of work, since these two bars share many
of the same bars to their right. How can we complete steps 1 and 2 in parallel without duplicating work? Let’s rearrange the above list slightly.

1. Calculate \( \text{max}_l \) for each \( b_i \).
2. Calculate \( \text{max}_r \) for each \( b_i \).
3. For each \( b_i \), find \( h_i = \min(\text{max}_l, \text{max}_r) \).
4. For each \( b_i \), let \( w_i = \max(h_i - b_i, 0) \).
5. Sum all of the \( w_i \).

Note that we have split the previous step 2 into 2 steps, 3 and 4. This is starting to look more tractable. Let’s take each step, assuming \( \text{hist} \) is a sequence of integers representing the histogram.

1. We’ve more or less already seen how to do this with \text{scan}. We just change the code above to use \text{Int.max} instead of \text{Int.min}:

   \[
   \text{val (lHeights, _) = scan Int.max 0 hist}
   \]

2. This is similar to step 1, except we want to take the max of all of the values to the right. We can still use \text{scan} for this, we just want to do a scan on the reversed list. Let’s assume we have a function \text{rev} that reverses a sequence:

   \[
   \text{val (rHeightsRev, _) = scan Int.max 0 (rev hist)}
   \]

3. The phrase “for each” should imply that a \text{map} is in order. But we want to map over two sequences, \( \text{lHeights} \) and \( \text{rev rHeightsRev} \) (note that we need to reverse \( \text{rHeightsRev} \) again since it was generated by a scan over a reversed sequence.) For this, we can use \text{map2}:

   \[
   \text{val heights = map2 Int.min lHeights (rev rHeightsRev)}
   \]

4. We define a function \text{nonNegative} as follows:

   \[
   \text{fun nonNegative (maxHeight, thisHeight) =}
   \text{Int.max (maxHeight - thisHeight, 0)}
   \]

   Step 4 can then be accomplished by mapping this function over \( \text{heights} \) and the original histogram:

   \[
   \text{map2 nonNegative heights hist}
   \]

5. Finally, we do a reduce to add all of these heights:

   \[
   \text{reduce op+ 0 (map2 nonNegative heights hist)}
   \]
Defining \texttt{rev} and putting it all together gives us the complete SML code for the histogram filling problem:

```sml
fun rev s = 
    let val n = length s 
    in tabulate (fn i => nth s (n - i - 1)) n 
    end

fun histogramFill (hist : int seq) = 
    let
        val (lHeights, _) = scan Int.max 0 hist 
        val (rHeightsRev, _) = scan Int.max 0 (rev hist) 
        val heights = map2 Int.min lHeights (rev rHeightsRev) 

        fun nonNegative (maxHeight, thisHeight) = 
            Int.max (maxHeight - thisHeight, 0) 
        in
            reduce op+ 0 (map2 nonNegative heights hist) 
        end
```

### 3.3.2 Matching Parentheses

We can use \texttt{scan} to solve the parenthesis matching problem that we went over two weeks ago. The idea is that we first map each open parenthesis to 1 and each close parenthesis to \(-1\). We then do a \texttt{+}-\texttt{scan} on this integer sequence. The elements in the sequence returned by \texttt{scan} exactly correspond how many unmatched parentheses there are in that prefix of the string. This is very much like the sequential algorithm we looked at in recitation, but \texttt{scan} lets us parallelize it! Recall that the parentheses are matched if and only if the counter never goes negative and is 0 at the end. We can check the first condition using a \texttt{reduce} over the sequence returned by \texttt{scan}, and the second by simply looking at the final value returned by \texttt{scan}.

For example:

\[
(\langle,\langle,\langle,\rangle\rangle,\rangle)\
\]

becomes

\[
(1,-1,1,1,-1,-1,-1)\
\]

and then \texttt{+}-\texttt{scan} gives

\[
(0,1,0,1,2,1,0),-1)\
\]

and then fails, because the counter went negative at some point indicating an imbalance.

```sml
fun match s = 
    let
        fun paren2int OPAREN = 1 
            | paren2int CPAREN = ~1 
```
val C = map paren2int s
val (S, total) = scan (op+) 0 C
val SOME(maxint) = Int.maxInt

in
  (reduce Int.min maxint S) >= 0 and also total = 0
end