# Recitation 3 — HW1 Debrief

Parallel and Sequenctial Data Structures and Algorithms, 15-210 (Fall 2011)

*14th September 2011*

## 1 Announcements

- HW3 will come out today. It will be due on 22 September, which is a Thursday.
- Questions from lecture, homework, etc.?
- Today we'll solve the harder recurrence and go through a careful presentation of the proof.

## 2 Recurrence in HW1

In assignment one, we had the following recurrence:

$$W(n) = 2W(n/2) + O(\lg n) + O(1)$$

This turned out to be hard, and you should not be shamed if you did not get it. The another recurrence was much easier, and I almost everyone got the correct answer $O(n \lg n)$. It had the familiar structure of merge sort, for example.

Most typical *incorrect* answer for the above recurrence was $W(n) \in O(\lg^2 n)$. Let's first see why that does not work, using the induction (substitution) method.

### 2.1 Showing $W(n) \notin O(\lg^2 n)$

We start with the induction hypothesis (IH) that $W(n) \in O(\lg^2 n)$. This is equal to saying that there exists $c$ such that $W(n) \leq c \lg^2 n$ when $n$ is above some minimum value.

That is, we assume that $W(n/2) \leq c \lg^2 n$ and then substitute this to the recurrence on $W(n)$. We also replace the O(..) from the recurrence with corresponding definition and use constants $k_1$ and $k_2$ for them:

$$\begin{aligned} W(n) &\leq 2W(n/2) + k_1 \lg n + k_2 \\ &\leq 2c \lg^2 (n/2) + k_1 \lg n + k_2 \\ &= 2c(\lg n - \lg 2)^2 + k_1 \lg n + k_2 \end{aligned}$$

Now $\lg = \log_2$ so $\lg 2 = 1$:

$$\begin{aligned} &= 2c(\lg n - 1)^2 + k_1 \lg n + k_2 \\ &= 2c(\lg^2 n - 2\lg n + 1) + k_1 \lg n + k_2 \\ &= 2c \lg^2 n + \textit{(lower degree terms)} \end{aligned}$$

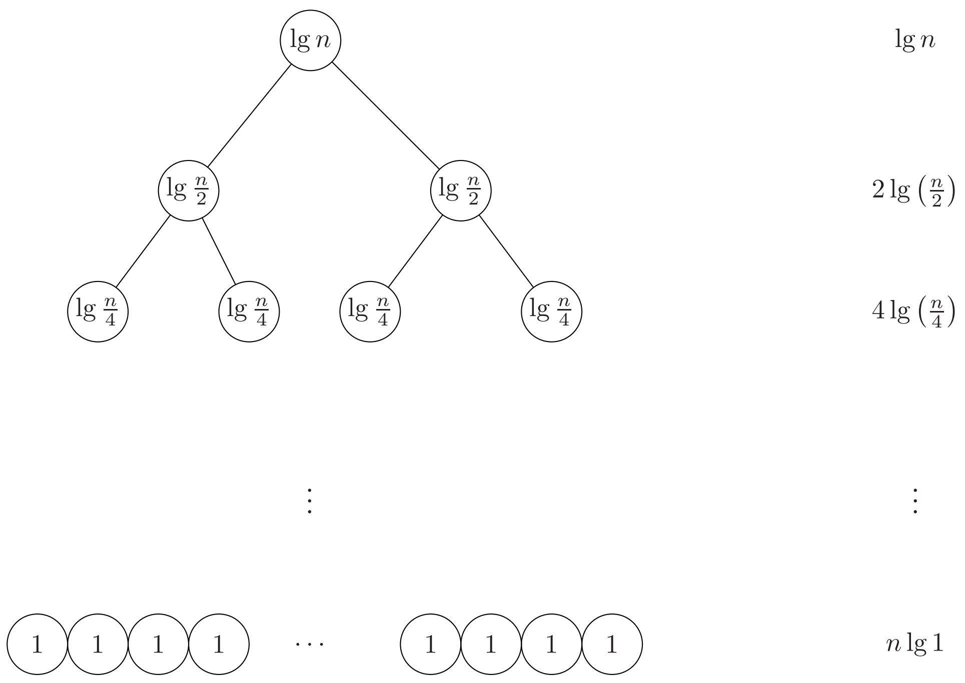
$\lg n$
$\lg \frac{n}{2}$
$\lg \frac{n}{4}$
1
$\lg n$
$2 \lg \left(\frac{n}{2}\right)$
$4 \lg \left(\frac{n}{4}\right)$
$n \lg 1$

Figure 1: tree method example

Although it may look like we showed $W(n) \in O(log^2 n)$, we have not. $W(n) \leq 2c \lg^2 n +$ ***(lower degree terms)*** does not imply $W(n) \leq c \lg^2 n$ for any choice of $c$. It is crucial that the proof is of the form $W(n) \leq 2W(n/2) + k_1 \lg n + k_2 \leq ... \leq c \lg^2 n$ to prove the inductive hypothesis. In this case, we showed that the work grows twice as fast as we expected, and our proof failed.

It turns out the correct answer is $O(n)$. But using $W(n) \leq cn$ as induction hypothesis will not work either - we are left with annoying terms that depend on $n$. But just because that form of the function failed to prove the hypothesis, does not in itself mean the hypothesis is wrong. For the induction method to work, one must then use a stronger hypothesis. However, it requires some trial and error to come with a good guess.

In this case, it is therefore good to use the **tree method** at least to get a grip on what is the correct complexity.

## 2.2 Using tree method

$$W(n) = 2W(n/2) + O(\lg n) + O(1)$$

The tree method will involve a nasty sum. To make things easier, and without loss of generality, let's assume $n$ is a power of 2, i.e $n = 2^k$ for some integer $k$.

Tree itself is easy to draw: at each level we branch each node to two and halve the size of the problem. For each level, we write what work is done on that level. The recursive part is taken into account when we sum all levels together.

- How many levels does the tree have? ($\lg n$)
- How many leaves are in the tree? ($n$, which is a good hint that $W(n) \notin O(\lg^2 n)$)
- Do you see the general form of the work on each level? ($2^i(a \lg \frac{n}{2^i} + b)$)

When we sum all the levels together, we come up with following sum:

$$\begin{aligned} S =& \sum_{i=0}^{\lg n} 2^i a \lg \frac{n}{2^i} + \sum_{i=0}^{\lg n} 2^i b \\ =& a \sum_{i=0}^{\lg n} 2^i \lg \frac{n}{2^i} + b \sum_{i=0}^{\lg n} 2^i \\ =& aS_1 + bS_2 \end{aligned}$$

- What does the $S_2$ evaluate to? This is familiar from binary arithmetic. $(2n - 1)$

So let's look at the first sum. With some simple rules of logs ($\log a^x = x \log a, \log \frac{a}{b} = \log a - \log b$)

$$\begin{aligned} S_1 =& \sum_{i=0}^{\lg n} 2^i \lg \frac{n}{2^i} \\ =& \sum_{i=0}^{\lg n} 2^i (\lg n - \lg 2^i) \\ =& \sum_{i=0}^{\lg n} 2^i (\lg n - i) \\ =& \left( \sum_{i=0}^{\lg n} 2^i \lg n - \sum_{i=0}^{\lg n} 2^i i \right) \end{aligned}$$

For the first sum, we have the same case as previously, and we get

$$S_1 = (2n - 1) \lg n - \sum_{i=0}^{\lg n} 2^i i$$

The second sum is the tricky one. But using **telescopic sums**, we can solve it. Telescopic sums are, by the way, quite common in recurrences and one needs some experience to detect them. For example, look how the formula for geometric sums is derived—it is telescoping.

So here is the trick:

$$\begin{aligned} s =& \sum_{i=0}^{k} 2^i i \\ 2s =& \sum_{i=0}^{k} 2^{i+1} i \\ s = 2s - s =& \sum_{i=0}^{k} 2^{i+1} i - \sum_{i=0}^{k} 2^i i \\ =& k2^{k+1} + (k-1)2^k - k2^k + (k-2)2^{k-1} - (k-1)2^{k-1} \ldots - 2 \end{aligned}$$

Here we ordered the sum elements by their exponent. Now you can see, if you look at the pairs with same exponent of form $(k-j)2^{k-j+1} - (k-j)2^{k-j}$, that this is:

$$(k-j-1)2^{k-j} - (k-j)2^{k-j} = -2^{k-j}$$

So we can simplify the sum to:

$$s = k2^{k+1} - \sum_{i=1}^{k} 2^i$$

And as $k = \lg n$:

$$s = 2n \lg n - (2n-2)$$

Back to our original sum, using the result above:

$$\begin{aligned} S &= aS_1 + bS_2 \\ &= a[(2n-1)\lg n - (2n \lg n - (2n-2)] + b(2n-1) \\ &= -a \lg n + (a+b)(2n-1) - a \end{aligned}$$

And this is clearly $\in O(n)$!

#### 2.2.1 Sidenote

So is $O(\lg^2 n)$ more or less work than $O(n)$? It is easy to test with some numbers:

| n | $\lg n$ | $\lg^2 n$ |
|---|---|---|
| 2 | 1 | 1 |
| 4 | 2 | 4 |
| 8 | 3 | 9 |
| 16 | 4 | 16 |
| 32 | 5 | 25 |
| 64 | 6 | 36 |

With very small numbers, the order changes, but when n¿=32, linear is more work than $\lg^2 n$.

It is useful also to compare the derivatives

$$\frac{d}{dn} \log^2 n = \frac{2 \log n}{n} < \frac{d}{dn} n = 1$$

when $n$ is large enough.

## 3 Proof About Code

### 3.1 General Thoughts

Proving the correctness of code and writing code are dependent processes. The structure of each informs the other, and you can reveal bugs in one through the other.

As you prove more and more about code, you should be able to look at a piece of code and very quickly envision the structure of proofs that you would want to prove it correct. If you can't, there's a good chance that your code isn't structured well. The structural induction principle for abstract sequences provided in HW2 hopefully helped you to do this.

To help you get practice with this, we'll be requiring that you annotate all the functions you write with a specification. You should think of this specification as a statement of the theorem of correctness that you would prove if you had to prove that function correct.

The documentation PDF should give you a feel for this level of specification. As you either noticed or should notice, the statements of the specs in that PDF are formal enough that you can use them in your proofs as well as in understanding the behaviour of the function.

## 3.2 Example

We'll now go through an example of a proof of one of the reference implementations of the stock market problem, emphasizing the identical structure.

### 3.2.1 Code

```
functor StocksDivAndConq (Seq : SEQUENCE) : STOCKS =
struct
  structure Seq = Seq
  open Seq
  open OC

  val max = Int.max
  val min = Int.min

  fun stock s =
      let
        fun stock' s =
            case showt s
             of EMPTY => raise Fail "invariant violated"
              | ELT(x) => (x, x, 0)
              | NODE(l,r) =>
                let
                  val ((minl, maxl, jl),(minr, maxr, jr)) = (stock' l, stock' r)
                in
                  (min(minl, minr),
                   max(maxl, maxr),
                   max(jl,  max (jr, maxr-minl)))
                end
      in
        case length s
         of 0 => NONE
          | _ =>
            let
              val (_,_, ans) = stock' s
            in
              SOME ans
            end
      end
end
```

### 3.2.2 Structure of Proof

We will prove the correctness of the implementation in `StocksDivAndConq` in `stocks.sml`. The other implementations are correct, and may have some stylistic advantages, but their proofs of correctness are choked with a lot of lemmas about option types.

Since this implementation has an inner helper function, we will prove two theorems. Since the inner function is recursive and the outer function isn't, one proof will be by induction and the other won't. Since the inner helper function raises an exception on argument sequences with length zero, our theorem about the inner function will be stated to exclude such sequences.

In the proof about the inner helper function, we'll never consider the extremal values of the empty set, so we'll represent them with integers. In the proof about the outer function, we will use `NONE` to represent either extremal value of an empty set and either `SOME(x)` to represent either extremal value of a non-empty set extreme is $x$.

### 3.2.3 Proof

Theorem: For all integer sequences $s$, if $|s| > 1$ and $\texttt{stock}'\texttt{s} \Longrightarrow \texttt{v}$ for some value $v$ of type $\texttt{int} * \texttt{int} * \texttt{int}$ then

$$v = (\min\{s_i | 0 \leq i < |s|\}, \max\{s_i | 0 \leq i < |s|\}, \max\{s_j - s_i | 0 \leq i \leq j < |s|\})$$

Theorem: For all integer sequences $s$, if $\texttt{stocks} \Longrightarrow \texttt{v}$ for some value $v$ of type `int option` then

$$v = \max\{s_j - s_i | 0 \leq i \leq j < |s|\}$$

*See HW01 solution for the actual proofs.*

### 3.2.4 What Isn't There

You'll notice that there are a few things that we didn't include in the above proof.

- When we called to one of the library functions, we applied its spec as we needed to and kept moving. In general, we only refer to the relevant parts of the spec, even if it's substantially more complicated.
- We didn't prove termination. We were interested in correctness, and mostly implicitly assumed termination.