# Recitation 2 — Recurrences and Scan

Parallel and Sequential Data Structures and Algorithms, 15-210 (Spring 2012)

*January 25, 2012*

## 1 Announcements

- HW1 is due tonight at 23:59EST. Hopefully you have all started by now; if not, now would be a good time.
- HW2 will be out tomorrow, and due next Thursday. There will be two main programming problems, which we will introduce to you towards the end of today's recitation.
- Office Hours have been posted on the course website. Locations are currently listed as TBA, but will in most cases be held in the 4th floor TA offices (either 4122 or 4126).
- There is now a staff mailing list. If you have questions which you feel should not be posted to the bboard (either due to course policy or for personal reasons), email

  `15210-staff@lists.andrew.cmu.edu.`
- Questions from lecture, homework, or life?

## 2 Recurrences

Let's start by solving a recurrence which should be familiar to all of you as a warmup:

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

Suppose $W(1) \in O(1)$. We claim $W(n) \in O(n)$. Is this true? Let's try to prove it, by induction.

**Base case:** Given.

**Inductive hypothesis:** $W(n) = O(n)$

**Inductive case:**

$$\begin{aligned} W(n) &= 2W(n/2) + O(n) \\ &= 2\left[O(n/2)\right] + O(n) \\ &= 2O(n) + O(n) \\ &= O(n) \end{aligned}$$

So, we proved that $W(n) \in O(n)$. Or did we?

## 2.1 A Closer Look

What went wrong? Let's take a closer look at the definition of Big-$O$. When we say $W(n) \in O(n)$, we mean there exists some $n_0, c$ such that for all $n > n_0$, $W(n) \le c \cdot n$. That is to say, $n_0$ and $c$ must both be fixed. Specifically, we want to show $W(n) \le c_1 \cdot n + c_2$. This isn't the case in our proof of the inductive case:

$$\begin{aligned} W(n) &\le 2W(n/2) + k_1 \cdot n + k_0 \\ &\le 2\left[c_1 \cdot n/2 + c_2\right] + k_1 \cdot n + k_0 \\ &\le (c_1 + k_1) \cdot n + 2 \cdot c_2 + k_0 \end{aligned}$$

Suddenly, we have a constant factor $k$ more than $c_1$ of $n$, so we cannot conclude that $W(n) \in O(n)$.

## 2.2 Brick Method

Yesterday in lecture we went over the brick method for determining if a recurrence is root-dominated, leaf-dominated, or balanced. It's a good way to get started when solving a recurrence.

Recall from lecture:

**Lemma 2.1.** *If $f = O(n)$, there exist constants $k_1, k_2$ so that $f(n) \le k_1 n + k_2$*

*Proof.* Since $f = O(n)$, there exist $c$ and $n_0$ so that $f(n) \le cn$ for $n > n_0$. So $k_1 = c, k_2 = \max(f(i) : 0 \le i < n_0)$ works. □

- For $W(n) = 4W(n/2) + O(n)$, we have at level $i$:

| **Problem Size** | $n/2^i$ |
|---|---|
| **Node Cost** | $\le k_1(n/2^i) + k_2$ |
| **Number of Nodes** | $4^i$ |

  So the cost at each level is bounded by

$$4^i \cdot \left(k_1(n/2^i) + k_2\right) = k_1 \cdot 2^i \cdot n + 4^i \cdot k_2$$

  This gives us a stack of bricks which is dominated at the leaves because the cost at level $i$ geometrically *increases* by more than a constant factor of 2. So $W(n) = O(\text{number of leaves}) = O(n^2)$.

- For $W(n) = W(3n/4) + O(n)$, we have at level $i$:

| **Problem Size** | $(3/4)^i n$ |
|---|---|
| **Node Cost** | $\le k_1(3/4)^i n + k_2$ |
| **Number of Nodes** | 1 |

The cost at each level is bounded by

$$1 \cdot \left(k_1(3/4)^i + k_2\right) = k_1 \cdot (3/4)^i \cdot n + k_2$$

This gives us a stack of bricks which is dominated at the node because the cost at level $i$ geometrically *decreases* by a constant factor of $3/4$. So $W(n) = O(\text{cost at root}) = O(n)$.

- For $W(n) = 2W(n/2) + O(n)$, we have at level $i$:

| **Problem Size** | $n/2^i$ |
|---|---|
| **Node Cost** | $\leq k_1(n/2^i) + k_2$ |
| **Number of Nodes** | $2^i$ |
| **Level Cost** | $2^i \cdot \left(k_1(n/2^i) + k_2\right)$ |

The cost at each level is bounded by

$$2^i \cdot \left(k_1(n/2^i) + k_2\right) = k_1 \cdot n + 2^i \cdot k_2$$

This gives us a stack of bricks which is balanced throughout because the cost at every level is the same, within a constant factor. So $W(n) = O(\text{height of tree} * \text{work at each level}) = O(n \log(n))$.

## 3 Scan

Yesterday, we gave a little preview at the end of lecture of a function which we call `scan`. We'll go over the definition of `scan` briefly today, and show you how to solve parentheses matching with it.

`scan` takes a function as one of its arguments. All of the text below makes the assumption that this function is *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 *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 `scan` later that doesn't have them, but it's a cleaner way to start thinking about `scan` with these properties.

### 3.1 Definition

`scan` has type

$$\texttt{scan} : (\alpha * \alpha \to \alpha) \to \alpha \to \alpha\ \texttt{seq} \to (\alpha\ \texttt{seq} * \alpha)$$

Informally, scan computes both the reduction of the sequence using the supplied operator and the sequence of all the partial results that were computed along the way. We'll go with this for now, and provide a more detailed definition tomorrow in lecture.

With the assumption that `f` is associative, `(scan f b)` is logically equivalent to `(iterh f b)` in the same way that `(reduce f b)` is logically equivalent to `(iter f b)`.

Specifically, if `f` is a function that takes no more than a constant number of steps on all input, `(iterh f)` and `(iter f)` have both work and span in $O(n)$, whereas `reduce` and `scan` both have work in $O(n)$ and span in $O(\lg n)$.

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

### 3.2 Note on Terminology

If $f$ is a function and $I$ is a relevant identity for $f$, we'll often say "$f$-scan" to mean

```
fn s => scan f I
```

For example, a "+-scan" is

```
fn s => scan (op+) 0
```

### 3.3 Example Uses of Scan

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

#### 3.3.1 +-`scan`

Yesterday, when solving the Maximal Contiguous Subsequence Sum problem, we saw a simple use of `scan` to compute all the prefix sums of a sequence:

```
fun prefixsum s = scan (op+) 0 s
```

For example, for the sequence $\langle 5, 2, 3, 2, -1 \rangle$, `prefixsum` computes the pair

$$(\langle 0, 5, 7, 10, 12 \rangle, 11)$$

### 3.3.2 Matching Parentheses

We can use `scan` to solve the parenthesis matching problem that we went over last week. The idea is that we first map each open parenthesis to 1 and each close parenthesis to $-1$. We then do a $+$-`scan` on this integer sequence. The elements in the sequence returned by `scan` exactly correspond how many unmatched parenthesis there are in that prefix of the string.

For example:

$$\langle (,),(,(,),),) \rangle$$

becomes

$$\langle 1,-1,1,1,-1,-1,-1 \rangle$$

and then

$$\langle 0,1,0,1,2,1,0,-1 \rangle$$

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

```
functor ParensScan (S : SEQUENCE) : PARENS =
struct
  structure Seq = S
  open Seq

  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  andalso total = 0
      end
end
```

## 4 Homework 2

This week's homework asks you to produce algorithms solving two different problems. As a way to introduce the problems to you, we'll now go over the specifications and some examples to both.

### 4.1 Closest Pair

The closest pair problem is to find the closest pair of points when given an unordered list of points in some two dimensional Euclidian space. Specifically, if $d$ is the distance function for the space and $s$ is a sequence set of points, you want to compute

$$\min\left\{d(p_i, p_j) \mid 0 \le i < |s|, 0 \le j < |s|, i \ne j\right\}$$

For example, for an input sequence $s$ of

$$\langle (0,0),(1,3),(2,2),(3,4),(4,1)\rangle$$

the closest pair would be $(1,3),(2,2)$.

### 4.2 Insertion Sort

Insertion sort is a simple algorithm which you should all be familiar with. In this problem, we want you to approximate the cost by counting the number of ordered pairs in a sequence of integers which are out of order (that is, the first element is greater than the second). Specifically, if $s$ is the sequence of integers, you want to compute

$$\left|\left\{(s_i, s_j) \mid 0 \le i < j < |s|, s_i > s_j\right\}\right|$$

For example, for an input sequence $s$ of

$$\langle 3,6,2,5,0\rangle$$

We have the pairs (3,2), (3,0), (6,2), (6,5), (6,0), (2,0) and (5,0) which are out of order, so the output should be 7.