Solving Recurrences II

Data Structures and Parallelism

Logistics

Project 1 Final turn-in is Thursday night!

- Must have all code finished as well as the write-up done
- Final code turn-in is your last push to the master repo
- Write-ups submitted via GradeScope

Exercises 2 & 3 are out now

- Due this Friday
- Handouts to help you are on the website

Warm Up

Write a recurrence to describe the running time of Mystery.

If you have extra time, find the big-Θ running time.

```
Mystery(int[] arr){
    if(arr.length == 1)
        return arr[0];
    else if(arr.length == 2)
        return arr[0] + arr[1];
    //copies all but first two elements of arr.
    int[] smaller = Arrays.copyOfRange(arr, 2, arr.length);
    return a[0] + a[1] + Mystery(smaller);
}
```

Warm Up

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2 & \text{if } n = 2 \\ T(n-2) + 4 & \text{otherwise} \end{cases}$$

Is there an easier way?

We do all that effort to get an exact formula for the number of operations,

But we usually only care about the Θ bound.

There must be an easier way

Sometimes, there is!

Master Theorem

Given a recurrence of the following form:

$$T(n) = \begin{cases} d & when\ n \leq \text{some constant} \\ aT\left(\frac{n}{b}\right) + n^c & \text{otherwise} \end{cases}$$

Where a, b, c, and d are all constants.
The big-theta solution always follows this pattern:

If $\log_b a < c$ then $T(n)$ is $\Theta(n^c)$

If $\log_b a = c$ then $T(n)$ is $\Theta(n^c \log n)$

If $\log_b a > c$ then $T(n)$ is $\Theta\left(n^{\log_b a}\right)$

Apply Master Theorem

Given a recurrence of the form:

$$T(n) = \begin{cases} d \text{ when } n \leq \text{some constant} \\ aT\left(\frac{n}{b}\right) + n^c \text{ otherwise} \end{cases}$$

If $\log_b a < c$ then $T(n)$ is $\Theta(n^c)$

If $\log_b a = c$ then $T(n)$ is $\Theta(n^c \log n)$

If $\log_b a > c$ then $T(n)$ is $\Theta\left(n^{\log_b a}\right)$

$$T(n) = \begin{cases} 1 \text{ when } n \leq 1 \\ 2T\left(\frac{n}{2}\right) + n \text{ otherwise} \end{cases}$$

a = 2
b = 2
c = 1
d = 1

$$\log_b a = c \Rightarrow \log_2 2 = 1$$

$$T(n) \text{ is } \Theta(n^c \log_2 n) \Rightarrow \Theta(n^1 \log_2 n)$$

Reflecting on Master Theorem

Given a recurrence of the form:

$$T(n) = \begin{cases} d \text{ when } n \leq \text{some constant} \\ aT\left(\frac{n}{b}\right) + n^c \text{ otherwise} \end{cases}$$

If $\log_b a < c$ then $T(n)$ is $\Theta(n^c)$

If $\log_b a = c$ then $T(n)$ is $\Theta(n^c \log n)$

If $\log_b a > c$ then $T(n)$ is $\Theta\left(n^{\log_b a}\right)$

$height \approx \log_b a$

$branchWork \approx n^c \log_b a$

$leafWork \approx d\left(n^{\log_b a}\right)$

The $\log_b a < c$ case

- Recursive case conquers work more quickly than it divides work
- Most work happens near "top" of tree
- Non recursive work in recursive case dominates growth, n^c term

The $\log_b a = c$ case

- Work is equally distributed across levels of the tree
- Overall work is approximately work at any level x height

The $\log_b a > c$ case

- Recursive case divides work faster than it conquers work
- Most work happens near "bottom" of tree
- Work at base case dominates.

Benefits of Solving By Hand

If we had the Master Theorem why did we do all that math???

Not all recurrences fit the Master Theorem.

- Recurrences show up everywhere in computer science.
- And they're not always nice and neat.

It helps to understand exactly where you're spending time.

- Master Theorem gives you a very rough estimate. The Tree Method can give you a much more precise understanding.

Unrolling

There's an alternative to the tree method called "unrolling"

Instead of going through the process of making the tree and finding the work at each level, sometimes we want to just plug the formula into itself until we see a pattern.

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2 & \text{if } n = 2 \\ T(n-2) + 4 & \text{otherwise} \end{cases}$$

Unrolling

$T(n) = 4 + T(n - 2)$

$T(n) = 4 + 4 + T(n - 4)$

$T(n) = 4 + 4 + 4 + T(n - 6)$

$T(n) = 4k + T(n - 2k)$ want $n - 2k = 1 \; or \; 2$ $k = \frac{n-1}{2}$

$T(n) = 4\left(\frac{n-1}{2}\right) + 1 = 2\mathrm{n} - 1$ $\theta(n)$!

Benefits: No drawing, no big table.

Drawbacks: No drawing, no big table – harder to find patterns.

This week's section handout will have more examples.

Unrolling Example

$$T(n) = \begin{cases} 15 & if\ n \leq 1 \\ 10 + T\left(\frac{n}{2}\right) & otherwise \end{cases}$$

Really common recurrences

Should know how to solve recurrences with previous methods, but also recognize some really common ones:

$T(n) = O(1) + T(n/2)$	logarithmic
$T(n) = O(1) + 2T(n/2)$	linear
$T(n) = O(1) + T(n-1)$	linear
$T(n) = O(n) + T(n-1)$	quadratic
$T(n) = O(1) + 2T(n-1)$	exponential
$T(n) = O(n) + T(n/2)$	linear
$T(n) = O(n) + 2T(n/2)$	$O(n \log n)$

Note big-Oh can also use more than one variable

Example: can sum all elements of an n-by-m matrix in $O(nm)$

Amortization

What's the worst case for inserting into an ArrayList?

- O(n). If the array is full.

Is O(n) a good description of the worst case behavior?

- If you're worried about a single insertion, maybe.
- If you're worried about doing, say, n insertions in a row. NO!

Amortized bounds let us study the behavior of a bunch of consecutive calls.

Amortization

The most common application of amortized bounds is for insertions/deletions and data structure resizing.

Let's see why we always do that doubling strategy.

How long in total does it take to do n insertions?

We might need to double, but the total resizing work is at most O(n)

And the regular insertions are at most $n \cdot O(1) = O(n)$

So n insertions take $O(n)$ work total

Or amortized $O(1)$ time.

Amortization

Why do we double? Why not increase the size by 10,000 each time we fill up?

How much work is done on resizing to get the size up to n?

Will need to do work on order of current size every 10,000 inserts

$$\sum_{i=0}^{\frac{n}{10000}} 10000i \approx 10{,}000 \cdot \frac{n^2}{10{,}000^2} = O(n^2)$$

The other inserts do $O(n)$ work total.

The amortized cost to insert is $O\left(\frac{n^2}{n}\right) = O(n)$.

Much worse than the $O(1)$ from doubling!