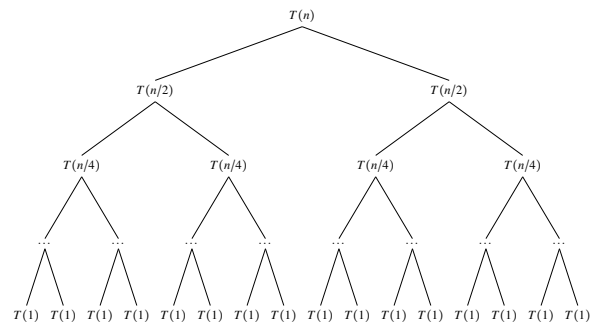


CSE 332

Data Abstractions

Algorithm Analysis 2



What is a recurrence?

In CSE 311, you saw a bunch of questions like:

Induction Problem

Let $f_0 = 0, f_1 = 1, f_n = f_{n-1} + f_{n-2}$ for all $n \geq 2$. Prove $f_n < 2^n$ for all $n \in \mathbb{N}$.

(Remember the Fibonacci Numbers? You'd better bet they're going to show up in this course!)

That's a recurrence. That's it.

Definition (Recurrence)

A recurrence is a recursive definition of a function in terms of smaller values.

Let's start with trying to analyze this code:

LinkedList Reversal

```

1 reverse(L) {
2   if (L == null) {
3     return null;
4   }
5   else {
6     Node front = L;
7     Node rest = L.next;
8     L.next = null;
9
10    Node restReversed = reverse(rest);
11    append(front, restReversed);
12  }
13 }
```

Notice that `append` is the same function from the beginning of lecture that had runtime $\mathcal{O}(n)$.

So, what is the time complexity of `reverse`?

We split the work into two pieces:

- Non-Recursive Work
- Recursive Work

LinkedList Reversal

```

1 reverse(L) {
2   if (L == null) { //O(1)
3     return null;
4   }
5   else {
6     Node front = L; //O(1)
7     Node rest = L.next; //O(1)
8     L.next = null; //O(1)
9
10    Node restReversed = reverse(rest);
11    append(front, restReversed); //O(n)
12  }
13 }
```

Non-Recursive Work: $\mathcal{O}(n)$, which means we can write it as $c_0 + c_1 n$ for some constants c_0 and c_1 .

LinkedList Reversal

```

1 reverse(L) {
2   if (L == null) {
3     return null;
4   }
5   else {
6     Node front = L;
7     Node rest = L.next;
8     L.next = null;
9
10    Node restReversed = reverse(rest);
11    append(front, restReversed);
12  }
13 }
```

Non-Recursive Work: $\mathcal{O}(n)$, which means we can write it as $c_0 + c_1 n$ for some constants c_0 and c_1 .

Recursive Work: The work it takes to do `reverse` on a list one smaller. Putting these together almost gives us the recurrence:

$$T(n) = c_0 + c_1 n + T(n-1)$$

We're missing the base case!

LinkedList Reversal

```

1 reverse(L) {
2   if (L == null) {
3     return null;
4   }
5   else {
6     Node front = L;
7     Node rest = L.next;
8     L.next = null;
9
10    Node restReversed = reverse(rest);
11    append(front, restReversed);
12  }
13 }
```

$$T(n) = \begin{cases} d_0 & \text{if } n = 0 \\ c_0 + c_1 n + T(n-1) & \text{otherwise} \end{cases}$$

Now, we need to **solve** the recurrence.

$$T(n) = \begin{cases} d_0 & \text{if } n = 0 \\ c_0 + c_1 n + T(n-1) & \text{otherwise} \end{cases}$$

$$\begin{aligned}
T(n) &= (c_0 + c_1 n) + T(n-1) \\
&= (c_0 + c_1 n) + (c_0 + c_1(n-1)) + T(n-2) \\
&= (c_0 + c_1 n) + (c_0 + c_1(n-1)) + (c_0 + c_1(n-2)) + \dots + (c_0 + c_1(1)) + d_0 \\
&= \sum_{i=0}^{n-1} (c_0 + c_1(n-i)) + d_0 \\
&= \sum_{i=0}^{n-1} c_0 + \sum_{i=0}^{n-1} c_1(n-i) + d_0 \\
&= nc_0 + c_1 \sum_{i=1}^n i + d_0 \\
&= nc_0 + c_1 \left(\frac{n(n+1)}{2} \right) + d_0 \\
&= \mathcal{O}(n^2)
\end{aligned}$$

Solving Linear Recurrences

10

A recurrence where we solve some constant piece of the problem (e.g. "-1", "-2", etc.) is called a **Linear Recurrence**.

We solve these like we did above by **Unrolling the Recurrence**.

This is a fancy way of saying "plug the definition into itself until a pattern emerges".

Now, back to mergesort.

Analyzing Merge Sort

11

Merge Sort

```
1 sort(L) {
2   if (L.size() < 2) {
3     return L;
4   }
5   else {
6     int mid = L.size() / 2;
7     return merge(
8       sort(L.subList(0, mid)),
9       sort(L.subList(mid, L.size()))
10    );
11  }
12 }
```

First, we need to find the recurrence:

$$T(n) = \begin{cases} d_0 & \text{if } n = 0 \\ d_1 & \text{if } n = 1 \\ c_0 + c_1 n + 2T(n/2) & \text{otherwise} \end{cases}$$

This recurrence isn't linear! This is a "divide and conquer" recurrence.

Analyzing Merge Sort

12

$$T(n) = \begin{cases} d_0 & \text{if } n = 0 \\ d_1 & \text{if } n = 1 \\ c_0 + c_1 n + 2T(n/2) & \text{otherwise} \end{cases}$$

This time, there are multiple possible approaches:

Unrolling the Recurrence

$$\begin{aligned} T(n) &= (c_2 + c_1 n) + 2(c_2 + c_1 n + 2T(n/4)) \\ &= (c_2 + c_1 n) + 2(c_2 + c_1 n + 2(c_2 + c_1 n + 2T(n/8))) \\ &= c_2 + 2c_2 + 4c_2 + \dots + \text{argh} + \dots \end{aligned}$$

This works, but I'd rarely recommend it.

Insight: We're **branching** in this recurrence. So, represent it as a tree!


Merge Sort: Solving the Recurrence

13

$$T(n) = \begin{cases} d_0 & \text{if } n = 0 \\ d_1 & \text{if } n = 1 \\ c_0 + c_1 n + 2T(n/2) & \text{otherwise} \end{cases}$$

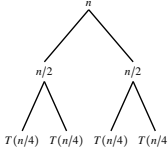
Merge Sort: Solving the Recurrence

14

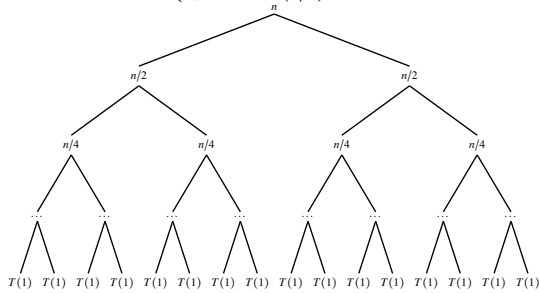
$$T(n) = \begin{cases} d_0 & \text{if } n = 0 \\ d_1 & \text{if } n = 1 \\ c_0 + c_1 n + 2T(n/2) & \text{otherwise} \end{cases}$$


Merge Sort: Solving the Recurrence

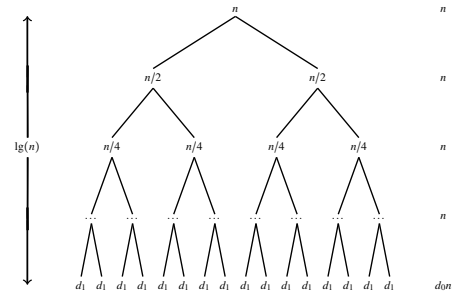
15

$$T(n) = \begin{cases} d_0 & \text{if } n = 0 \\ d_1 & \text{if } n = 1 \\ c_0 + c_1 n + 2T(n/2) & \text{otherwise} \end{cases}$$


$$T(n) = \begin{cases} d_0 & \text{if } n = 0 \\ d_1 & \text{if } n = 1 \\ c_0 + c_1 n + 2T(n/2) & \text{otherwise} \end{cases}$$



$$T(n) = \begin{cases} d_0 & \text{if } n = 0 \\ d_1 & \text{if } n = 1 \\ c_0 + c_1 n + 2T(n/2) & \text{otherwise} \end{cases}$$



Since the recursion tree has height $\lg(n)$ and each row does n work, it follows that $T(n) \in \mathcal{O}(n \lg(n))$.

Find A Big-Oh Bound For The Worst Case Runtime

```

1 sum(n) {
2   if (n < 2) {
3     return n;
4   }
5   return 2 + sum(n - 2);
6 }

```

$$T(n) = \begin{cases} d_0 & \text{if } n = 0 \\ d_0 & \text{if } n = 1 \\ c_0 + T(n-2) & \text{otherwise} \end{cases}$$

$$\begin{aligned}
 T(n) &= c_0 + c_0 + \dots + c_0 + d_0 \\
 &= c_0 \left(\frac{n}{2} \right) + d_0 \\
 &= \mathcal{O}(n)
 \end{aligned}$$

Find A Big-Oh Bound For The Worst Case Runtime

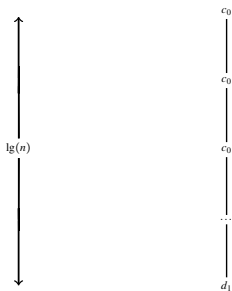
```

1 binarysearch(L, value) {
2   if (L.size() == 0) {
3     return false;
4   }
5   else if (L.size() == 1) {
6     return L[0] == value;
7   }
8   else {
9     int mid = L.size() / 2;
10    if (L[mid] < value) {
11      return binarysearch(L.subList(mid + 1, L.size()), value);
12    }
13    else {
14      return binarysearch(L.subList(0, mid), value);
15    }
16  }
17 }

```

$$T(n) = \begin{cases} d_0 & \text{if } n = 0 \\ d_1 & \text{if } n = 1 \\ c_0 + T(n/2) & \text{otherwise} \end{cases}$$

$$T(n) = \begin{cases} d_0 & \text{if } n = 0 \\ d_1 & \text{if } n = 1 \\ c_0 + T(n/2) & \text{otherwise} \end{cases}$$



$$\text{So, } T(n) = c_0(\lg(n) - 1) + d_1 = \mathcal{O}(\lg n).$$

■ Gauss' Sum: $\sum_{i=0}^n i = \frac{n(n+1)}{2}$

■ Infinite Geometric Series: $\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$, when $|x| < 1$.

■ Finite Geometric Series: $\sum_{i=0}^n x^i = \frac{1-x^{n+1}}{1-x}$, when $x \neq 1$.

Consider a recurrence of the form:

$$T(n) = \begin{cases} d & \text{if } n = 1 \\ aT\left(\frac{n}{b}\right) + n^c & \text{otherwise} \end{cases}$$

Then,

- If $\log_b(a) < c$, then $T(n) = \Theta(n^c)$.
- If $\log_b(a) = c$, then $T(n) = \Theta(n^c \lg(n))$.
- If $\log_b(a) > c$, then $T(n) = \Theta(n^{\log_b(a)})$.

Sanity Check: For Merge Sort, we have $a = 2, b = 2, c = 1$. Then, $\log_2(2) = 1 = 1$. So, $T(n) = n \lg n$.

$$T(n) = \begin{cases} d & \text{if } n = 1 \\ aT\left(\frac{n}{b}\right) + n^c & \text{otherwise} \end{cases}$$

We assume that $\log_b(a) < c$. Then, unrolling the recurrence, we get:

$$\begin{aligned} T(n) &= n^c + aT(n/b) \\ &= n^c + a((n/b)^c + aT(n/b^2)) \\ &= n^c + a(n/b)^c + a^2(n/b^2)^c + \dots + a^{\log_b(n)}(n/b^{\log_b(n)})^c \\ &= \sum_{i=0}^{\log_b(n)} a^i \left(\frac{n^c}{b^{ic}}\right) \\ &= n^c \sum_{i=0}^{\log_b(n)} \left(\frac{a}{b^c}\right)^i \\ &= n^c \left(\frac{\left(\frac{a}{b^c}\right)^{\log_b(n)+1} - 1}{\left(\frac{a}{b^c}\right) - 1} \right) \approx n^c \left(\left(\frac{a}{b^c}\right)^{\log_b(n)} \right) \approx n^c \end{aligned}$$

Today's Takeaways!



- Know how to make a recurrence from a recursive program
- Know the different types of recurrences
- Be able to find a closed form for each type of recurrence
- Know the common summations
- Understand why Master Theorem can be helpful