These are *rough notes* based on my own notes I prepared for lecture. They go with the annotated lecture slides you can see on the "More" page of the course website. Although they are rough, I'm providing them here in case they help you remember what we did in lecture.

1 Legend

Text that was already on the slides is in a box, like this.
Descriptions of things that were already on the slides are formatted like this.

Text I planned to write during lecture is formatted like this, with a line to the left.

Descriptions of other things I planned to draw or write or do are formatted like this.

Anything else is text I planned to say, or things I planned to ask.

2 Solving recurrences (continued)

Define $M : \mathbb{Z}^+ \to \mathbb{N}$ by
- $M(1) = c$
- For $n > 1$, $M(n) = M(\lceil n/2 \rceil) + M(\lfloor n/2 \rfloor) + dn$

where $c, d \in \mathbb{N}$ are constants.

Last week:
- Closed form when n is a power of 2: $M(n) = cn + dn \log_2 n$.
- We found this using *repeated substitution*.
- For a recap, see Section 3.2.1 of *Course notes for CSC B36/236/240*, linked from the *Textbooks* section the course homepage.

(Repeat definition of M.)
Theorem 1. $M(n) \in O(n \log n)$

To prove this, we'll use the fact that M is a nondecreasing function.

Lemma. $\forall a \in \mathbb{Z}^+. \forall b \in \mathbb{Z}^+. (a \leq b$ IMPLIES $M(a) \leq M(b))$
(Proof: see Lemma 3.6 on page 84 of *Course notes for B36/236/240* (Textbooks section of course website).)

(Arrow to Lemma.) Summary: use induction to prove M is nondecreasing on $\{1, 2, \ldots, k\}$ for all $k \in \mathbb{Z}^+$.

Proof of Theorem using Lemma:

Given any $n \in \mathbb{N}$, let 2^k be smallest power of 2 that is $\geq n$. So $n \leq 2^k < 2n$.

$$\begin{aligned} M(n) \leq & M(2^k) \\ = & c2^k + dk2^k \\ < & 2cn + 2dn \log_2(2n) \\ \in & O(n \log n) \end{aligned}$$

□

Master Theorem (for "divide and conquer" recurrences)
Suppose $T : \mathbb{Z}^+ \to \mathbb{R}^+$ is defined by:

- For $n < B$, $T(n) = c$
- For $n \geq B$, $T(n) = a_1 T(\lceil n/b \rceil) + a_2 T(\lfloor n/b \rfloor) + dn^\ell$

where $a_1, a_2, B \in \mathbb{N}$, $a = a_1 + a_2 \geq 1$, $b > 1$, $c, d, \ell \in \mathbb{R}^+$
Then:

$$T(n) \in \begin{cases} & \text{if } a = b^\ell \\ & \text{if } a < b^\ell \\ & \text{if } a > b^\ell \end{cases}$$

(Copy of the recurrence we just solved.)

Which case does the recurrence we just solved fit into?

(Beside the recurrence we just solved:)

$a = 2$, $b = 2$, $c = c$, $d = d$, $\ell = 1$

$a = b^\ell$ *(Arrow to $a = b^\ell$ case.)*

Fill in the missing parts to get:

$$T(n) \in \begin{cases} O(n^\ell \log n), & \text{if } a = b^\ell \\ O(n^\ell), & \text{if } a < b^\ell \\ O(n^{\log_b a}), & \text{if } a > b^\ell \end{cases}$$

2.1 Transformations

Note: some students pointed out a mistake in the following slide: the definition $G(n/2) + \cdots$ doesn't work when n is not even. So we should replace $n/2$ with $\lfloor n/2 \rfloor$ (or state that the recurrence is only defined when n is a power of 2), and we should also say that our solution to the recurrence assumes n is a power of 2.

Using transformations to solve recurrences
Example:

- $G(1) = 0$
- $G(n) = 2G(n/2) + \log_2 n$ when $n \in \mathbb{Z}^+$

Let $n = 2^k$.

Then $G(2^k) = 2G(2^{k-1}) + k$; $G(2^0) = 0$.

Let $H(k) = G(2^k)$.

$H(0) = 0$

$H(k) = 2H(k-1) + k$

Solve using repeated substitution: $H(k) = 2^{k+1} - k - 2$

So $G(n) = 2n - \log_2 n - 2$.

Domain transformation *(arrow to work we just did)*

This is called a domain transformation, because we transformed the input to the function.

We can also do range transformations, where we transform the output.

Example:

- $A(0) = 1$
- $A(n) = 3A(n-1)^2$ for $n \in \mathbb{Z}^+$

Let $B(n) = \log_2 A(n)$.

$B(n) = \log_2 3 + 2\log_2 A(n-1) = \log_2 3 + 2B(n-1)$

$B(0) = \log_2 1 = 0$

Repeated substitution:

$$\begin{aligned} B(n) &= \log_2 3 + 2 \cdot \log_2 3 + 4 \cdot \log_2 3 + \cdots + 2^n B(0) \\ &= (2^n - 1)\log_2 3 \end{aligned}$$

So $A(n) = 2^{B(n)} = 2^{(2^n-1)\log_2 3} = 3^{2^n-1}$

Range transformation *(arrow to work we just did)*

2.2 Linear recurrences using characteristic polynomials

Methods for solving recurrences:
1. Guess and verify
2. Repeated substitution and verify
3. Special techniques for "divide and conquer" recurrences / Master Theorem
4. Transformations
5. Solving linear recurrences using characteristic polynomials

Here's our last method for solving recurrences.

Linear recurrences using characteristic polynomials
Fibonacci recurrence:
- $F(0) = 0$
- $F(1) = 1$
- $F(n) = F(n-1) + F(n-2)$ for $n \in \mathbb{N}$, $n > 1$

▌ 0, 1, 1, 2, 3, 5, 8, 13, ...

This is an example of a *homogeneous linear recurrence.*

(Repeat Fibonacci recurrence.)
In general, the recursive case for a *homogeneous linear recurrence* looks like this:

$$\begin{aligned} f(n) =& a_1 f(n-1) + a_2 f(n-2) + \cdots + a_d f(n-d) \\ =& \sum_{i=1}^{d} a_i f(n-i) \end{aligned}$$

$d, a_1, \ldots, a_d$ are constants.

In a homogeneous linear recurrence, the recursive case is a fixed linear combination of previous terms.

(Repeat Fibonacci recurrence.)
Find a closed form for $F(n)$.

Let's try something easier first.

▌ *Cross out* $F(n-1) + F(n-2)$. *Replace it with* $2F(n-1)$. *Cross out* $F(0)$.

▌ $1, 2, 4, 8, \ldots$.

▌ $F(n) = 2^{n-1}$

(Repeat previous slide.)

To solve: guess $f(n) = cx^n$. Need to find c, x.

Plug in: $cx^n = cx^{n-1} + cx^{n-1}$

$x^2 = x + 1$

$x = (1 \pm \sqrt{5})/2$

$F(n) = c[(1 + \sqrt{5})/2)]^n$ or $F(n) = c[(1 - \sqrt{5})/2)]^n$

Okay, that's strange. And it doesn't even work.

(Arrow to $F(n) = c[(1 + \sqrt{5})/2)]^n$.) This would mean $\frac{F(2)}{F(1)} = \frac{1+\sqrt{5}}{2}$. But $\frac{F(2)}{F(1)} = \frac{1}{1} = 1$.

What if we combined them?

Try this: $F(n) = c_1[(1 + \sqrt{5})/2)]^n + c_2[(1 - \sqrt{5})/2)]^n$

Theorem 2. Consider any homogeneous linear recurrence, ignoring the base case(s):

$$f(n) = \sum_{i=1}^{d} a_i f(n-i)$$

If x_1^n and x_2^n are both solutions, then $c_1 x_1^n + c_2 x_2^n$ is a solution too, for any $c_1, c_2 \in \mathbb{R}$.

Proof:

x_1^n is a solution means

$$\begin{aligned} \forall n \in \mathbb{N}.\, x_1^n &= \sum_{i=1}^{d} x_1^{n-i} \\ c_1 x_1^n &= c_1 \sum_{i=1}^{d} x_1^{n-i} \\ &= \sum_{i=1}^{d} c_1 x_1^{n-i} \end{aligned}$$

Similarly $c_2 x_2^n = \sum_{i=1}^{d} c_2 x_1^{n-i}$.

Add:

$$c_1 x_1^n + c_2 x_2^n = \sum_{i=1}^{d} (c_1 x_1^{n-i} + c_2 x_2^{n-i}$$

So $f(n) = c_1 x_1^n + c_2 x_2^n$ is a solution.

Back to the Fibonacci recurrence

(Repeat Fibonacci recurrence.)

Try $F(n) = c_1[(1+\sqrt{5})/2)]^n + c_2[(1-\sqrt{5})/2)]^n$.

$F(0) = c_1 \cdot 1 + c_2 \cdot 1 = 0$ so $c_2 = -c_1$.

$$\begin{aligned} F(1) &= c_1(1+\sqrt{5})/2 - c_1(1-\sqrt{5})/2 = 1 \\ 1 &= c_1\sqrt{5} \\ c_1 &= 1/\sqrt{5} \end{aligned}$$

Solution:

$$F(n) = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n$$

Looks wrong ($\sqrt{5}$??) but can be proved by induction.

3 Analysis of algorithms

Analysis of algorithms
How long does this algorithm take to run?

(Still on slide...)

```
def linear_search(l : List[int], x : int):
  # If x occurs in l, returns an index of l at which x occurs.
  # Otherwise, returns None.
  i = 0
  while i < len(l):
    if l[i] == x:
      return x
    i += 1
  return None
```

Why does it matter?

- Predict how long it will take.
- Decide what inputs are reasonable.
- Compare different algorithms.

The actual runtime depends on lots of things.

Depends on:

- Programming language / compiler / interpreter
- The computer running it
- What other processes are running
- ...

So: instead of exact time, estimate to within a constant factor, and write $O(\cdot)$.

(Repeat code.)

Running time also depends on input. For algorithm A and input I, let $t_A(I)$ denote # steps algorithm performs on input I.

What is a step?

Since we're only estimating within a constant factor, we can choose! Make sure # steps is within a constant factor of total # operations.

(Circle `i < len(l)`, draw arrow to that.) 1 step.

- $t_{LS}((\texttt{[2, 4, 6, 8]}, 2)) = 1$
- $t_{LS}((\texttt{[2, 4, 6, 8]}, 8)) = 4$
- $t_{LS}((\texttt{[2, 4, 6, 8]}, 1)) = 4$

Runtime usually longer for bigger inputs.

$T_A(n)$ = running time on input of size n? Not well-defined.

Worst-case running time $T_A(n) = \max\{t_A(I) \mid \text{size}(I) = n\}$.

E.g. $T_{LS}(n) \in O(n)$.

Remember, we write big O because we're only estimating to within a constant factor.