CMSC 27200 - Guerilla Section 2.

Recursion, divide and conquer, and induction.

Topics

1. Recursion and recurrence relation.
2. Divide and conquer: binary search.
   - Pseudocode.
   - Correctness.
   - Running time.

Recursion

A function f is recursive if its defined w.r.t. itself.

The definition of a recursive fn is a recurrence relation

→ Anatomy

1) Base case: fn evaluated for fixed input
2) Inductive case: fn evaluated on itself.

Example: factorial.

From CMSC 27100: $n! := 1 \cdot 2 \cdot \ldots \cdot n$

We can define this recursively

$$T(n) = \begin{cases} 1 & \text{if } n = 0, 1, \ldots \\ n \cdot T(n-1) & \text{o.w.} \end{cases}$$

recurrence relation: recursive fn on natural #s.

→ Induction and recursion go hand in hand

For example. How would you prove that in fact. $T(n) = n!$? One could use ~~recursion~~ induction.

formal vs. ~~informal~~ sketch.

Claim (formal): $\forall n \in \mathbb{Z}_{\geq 0}$, $T(n) = n!$

Pf: Let $P(n)$ denote the predicate $T(n) = n!$.

As a base case, take $n = 0$. Note that

- $0! = 1$.
- By defn $T(0) = 1$.

→ $T(0) = 0!$ thus $P(0)$ holds.

Now suppose for some $n > 0$, $P(n)$ holds, we consider the inductive step. $P(n+1)$. Note that by defn.

Inductive hypothesis.

$$P(n+1) = (n+1) \cdot P(n) = (n+1) \cdot n! = (n+1)!$$

thus $P(n+1)$ also holds. Since $P(n) \Rightarrow P(n+1)$ by the ~~mathematical~~ principle of weak induction the claim holds. □.

Remark: This is a formal proof.! Note that

1) I clearly state that we use induction.
2) I prove a base case
3) I prove the inductive step.
   - In proving the inductive step. I clearly state what the inductive hypothesis is and where I use it.

What would a proof sketch look like?

" Suppose the claim holds for n.

$T(n+1) = (n+1) \cdot T(n) = (n+1) \cdot n! = (n+1)!$ □ "

Two lines.

what's complicated about applying IH, how do you ultimately apply it?

Divide and Conquer.

Remember CMSC 27200 is about learning different paradigms. ~~for~~ designing algorithms.

↳ Divide and conquer is our first named paradigm.

This is what a divide and conquer algorithm is.

"A divide and conquer algorithm is an algorithm that solves a problem by

1) Breaking the problem into subproblems that are smaller ~~probs~~ instances of the same problem.
2) Recursively solving those problems.
3) Appropriately recombining those solutions.

Example: binary search.

Binary search is an algorithm for finding an element in a sorted list

INPUT: A sorted list A of ~~numbers~~ $n \in \mathbb{Z}$.

and $k \in \mathbb{Z}$.

OUTPUT: If $k \in A$, output ~~the index of k~~ YES if $k \in A$.

Otherwise output ~~DONE~~ NO.

Now here's the pseudocode.

~~PROCEDURE~~: Binary search

INPUT: ~~Sorted list~~ A, integer k.

Do: the following.

~~1. If len(A) = 1, output~~ return ~~if~~ $A[0] = k$.

recombine: ~~2. let mid = $\left\lfloor \frac{len(A)}{2} \right\rfloor$.~~

⟹ Pseudocode

PROCEDURE: BinSearch.

INPUT: Sorted list A, integer k, integer "low"
integer "high".

Do: the following...

1. Let mid = $\lfloor \frac{low + high}{2} \rfloor$
2. If A[mid] = k → recombine solution
   - return Yes
3. If A[mid] < k
   - return BinSearch(A, k, low, mid−1)
4. If A[mid] > k.
   - return BinSearch(A, k, mid+1, high)

break into smaller subproblem.

recursively solve the subproblems.

Question: does a divide and conquer algorithm have to be recursive?

→ No... we could have written the pseudocode iteratively

Strat: break proof down into
⟹ Correctness. main "technical" lemma and a thm that "alg is correct"

Now let's try to formally prove correctness.

⟶ We'll proceed by induction.

Claim: (technical claim.) Given any $n \geq 0$, if high − low = n then BinSearch(A, k, low, high) works correctly for any A, k. ***

... actually let's be even more precise

Claim: Given any $n \geq 0$. if high − low = n then for any sort A and int k, BinSearch(A, k, low, high) satisfies the following.

1) If $k \in A[low \ldots high]$ then BinSearch ... returns yes

2) Else it returns NO.

Pf: Let us proceed by induction.

→ base case: $n = 0$. If $n = 0$ then

low = high. = 0 and so low = high.

Let's verify each subpart of the claim.

1) Suppose $k \in A[low \ldots high]$.

Since low = high. that means

$k = A[low]$.

Now what does algo do? It computes

$$mid = \left\lfloor \frac{low + high}{2} \right\rfloor = \left\lfloor \frac{2 \cdot low}{2} \right\rfloor = low.$$

Then checks if $k \stackrel{?}{=} A[mid] = A[low]$

Since we know $k = A[low]$, it returns

Yes correctly.

2) Suppose $k \notin A[low \ldots high]$.

Since low = high, we have $k \neq A[low]$

Algo will check mid = low and see that $k \neq A[mid]$ and thus correctly return NO

→ Inductive Step: We'll now use Strong induction. Fix $n \in \mathbb{Z}_{\geq 0}$ and suppose $\forall n' \leq n$ the claim holds for n'.

WTS: claim holds for n.

~~For this we look at the algorithm directly~~

~~There are three cases.~~

1. ~~$k = A[mid]$: then algorithm does the right thing. i.e. claim case 1 holds and case 2 holds vacuously.~~

2. ~~$k < A[mid]$~~ :

For this we go through each pair of the claim and look at what the algorithm does.

(1) Suppose $k \in A[low \ldots high]$. There are three cases.

Case 1 → $k = A[mid]$ → algorithm returns yes as it should.

Important: sorted

Case 2 → $k < A[mid]$ → because A is sorted and $k \in A[low \ldots high]$ we now know

↳ $k \in A[low \ldots \cancel{high}\ mid-1]$.

This is where we use the inductive hypothesis

precondition to the IH

a) $(mid-1) - low < high - low = n$.

b) $k \in A[low \ldots mid-1]$

by the inductive hypothesis.

BinSearch $(A, k, low, mid-1)$

returns yes. ← go through slowly.

Case 3: → $k > A[mid]$. → because A is sorted

and $k \in A[low \ldots high]$ we know

$k \in A[mid+1, \ldots, high]$

Now. apply the inductive hypothesis

a) $high - (mid+1) < high - low = n$.

b) $k \in A[mid+1, \ldots, high]$.

Thus BinSearch(A, k, ~~high, hi~~ mid+1, high)

returns yes.

⟶ In all cases we show

$A \in k \in A[low \ldots high] \Rightarrow$ BinSearch(A, k, low, high)

returns yes.

Now to show item (2) in the claim.

Suppose $k \notin A[low \ldots high]$. ~~Two cases~~. Then

$k \neq A[mid]$ and two cases must occur.

either $k < A[mid]$ or $k > A[mid]$.

⟶ In both cases

$mid-1 - low < high - low \leq n$.

high − (mid+1) < high − low = n.

Additionally...

$k \notin A[low \ldots high]$

$\longrightarrow$ $k \notin A[low \ldots mid-1]$ and

$k \notin A[mid+1 \ldots high]$.

Thus by <u>inductive hypothesis</u> ...

BinSearch(A, k, low, mid−1)

BinSearch(A, k, mid+1, hi)

must both return <u>no</u>. ✡

What would a sketch look like?

$\longrightarrow$ really just the inductive case for proving claim (1)

$\longrightarrow$ Alg is correct?

Note since claim holds $\forall n$. BinSearch(A, k, 0, n−1) for n = length(A) returns ~~true~~ yes. $\Leftrightarrow$ $k \in A$.

~~⟹ Running Time.~~

~~When analyzing running times of divide and conquer algorithms, one typically uses recurrence relations.~~

Big picture view: a correct proof is sufficiently "entropic".

↳ you use all the "weird" things you made your algorithm do.

↳ you use all the assumptions you felt you need to make.

~~⟹ Running Time.~~

~~→ What does a proof sketch look like~~

⟹ Running time.

When analyzing running times of divide and conquer algorithms, you typically encounters

recurrence relations.

The point is this: ~~divide-and-conquer algs~~

→ many divide and conquer algorithms take the following shape.

1) Take a problem of size n.

2) ~~Recursive~~ Breaks the problem into a pieces of size $\frac{n}{b}$ input.

3) Recursively call the alg on each of a branches into n/b sized pieces

4) Do ~~f(n)~~ ~~work~~ polynomial time work to put the solution back together. → n^d

→ How does this look for binary search?

↳ problem of size n.

↳ break into $a = 1$ branch of size $\frac{n}{2}$ input.

- Check the middle elem. If $k <$ middle elem you recurse on left half.
- If $k >$ middle → recurse on right half.
- All cases ∴ recurse only on 1 half.

↪ $O(1)$ work to put the string back together

- you just return the result.

Now how do you model the part of code done mathematically?

Let $T(n) :=$ # of unit ops done on an input of size n.

We have

$$T(n) = \underset{a}{1} \cdot T\left(\left\lceil \frac{n}{\underset{b}{2}} \right\rceil\right) + O(n^d).$$

Okay but now we want an explicit running time.

⟹ Master theorem.

Theorem: Suppose you have a recurrence relation given a, b, d.

$$T(n) = \overset{a}{b} \cdot T\left(\left\lceil \frac{n}{a} \right\rceil\right) + O(n^d)$$

Then three cases.

1. If $d > \log_b a \longrightarrow T(n) = O(n^d)$
2. If $d = \log_b a \longrightarrow T(n) = O(n^d \cdot \log n)$
3. If $d < \log_b a \longrightarrow T(n) = O(n^{\log_b a})$.

→ before going through Master thm ... primer on log rules.

- $\log_b(x)$ is defined as the α s.t. $\forall x \in \mathbb{R}$

$$\log_b(b^x) = x.$$

- Some rules.

1) Product rule: $\log_b(xy) = \log_b|x| + \log_b(y)$

2) quotient rule: $\log_b\left(\frac{x}{y}\right) = \log_b(x) - \log_b(y)$

3.) Power rule: $\log_b(x^k) = k \cdot \log_b(x)$

4) Change of basis.

$\log_a x = \log_b x \cdot \log_a b$.

Question for $f(n) = \log_{10} n$ and $g(n) = \ln(n)$

Is $f(n) = O(g(n))$, $\Omega(g(n))$ or $\Theta(g(n))$?

↳ Now go over master theorem

Okay so ~~let's build some~~ how do we use this theorem?

→ binary search

$T(n) = T(n/2) + \Theta(1)$

↳ $b = 2$, $a = 1$, $d = 0$.

→ $\log_b a = \log_2 1 = 0$.

→ $d = 0$

→ $d = \log_b a$.

Thus master theorem = $O(n^d \cdot \log n) = O(\log n)$

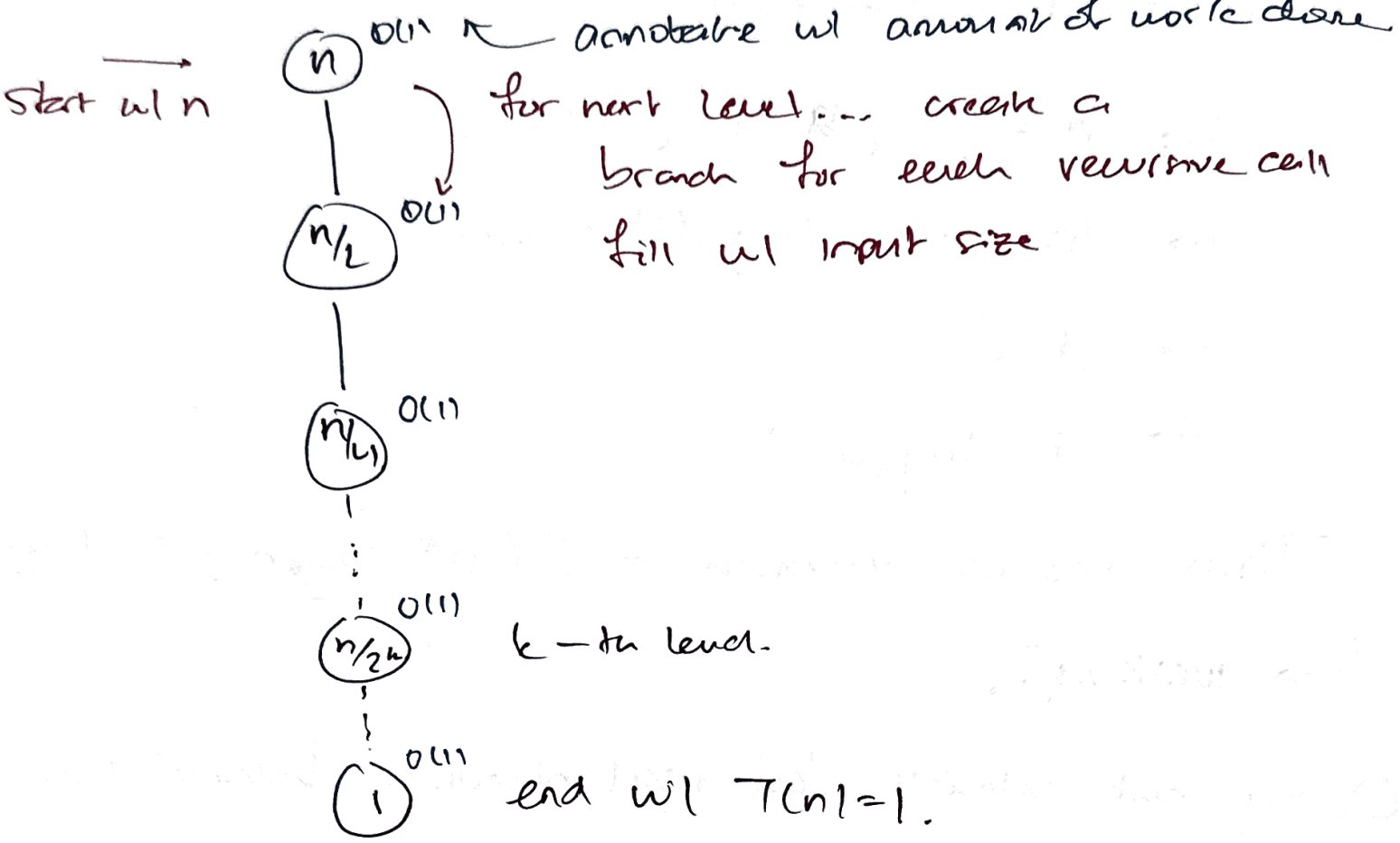
↳ questions?

Okay but what's the real intuition behind this theorem?

→ How do you hand analyze a recurrence relation?

→ draw a tree

$$T(n) = T\left(\left\lceil \frac{n}{2} \right\rceil\right) + O(1)$$



Now... here's the heuristic... Stochiometry

$$T(n) = \underbrace{\#\text{ layers in tree}}_{\text{how many divides until input size} = 1} \cdot \underbrace{\left(\frac{\#\text{ nodes}}{1\text{ layer}}\right)}_{\text{how many factors of the branching factor have accumulated}} \cdot \underbrace{\left(\frac{\text{amt work}}{1\text{ node}}\right)}_{\text{sum the cost.}}$$

# of layers ⇒ $1 = \frac{n}{2^k} \rightarrow 2^k = n \rightarrow k = \log_2 n$

# of nodes per layer ⇒ 1 node / layer.

amt of work per node ⇒ $O(1)$ work per node

$$T(n) = \underbrace{\log_2 n}_{\text{\# of layers}} \cdot \underbrace{1}_{\text{\# nodes/layer}} \cdot \underbrace{O(1)}_{\text{amt of work/node.}} = \underbrace{O(\log_2 n)}_{\text{running time!}}$$

→ questions?

⇒ Is it always just multiply a bunch of numbers to get our? Nope

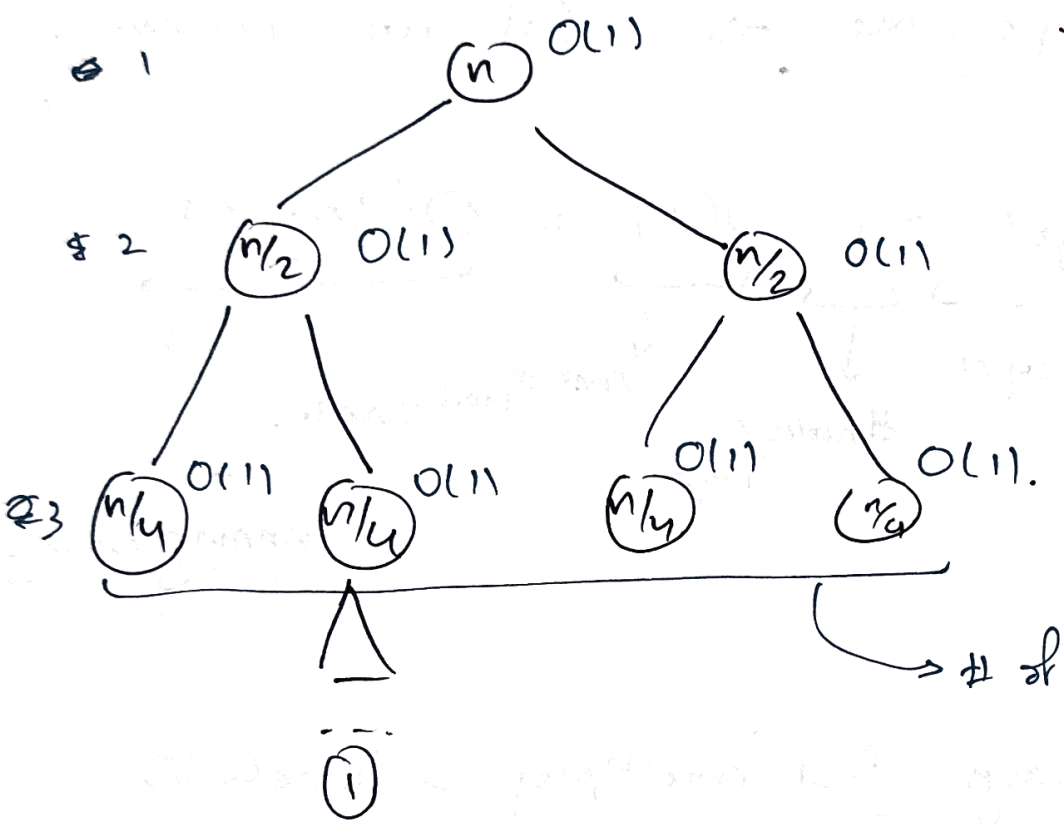
↳ We can multiply because each term is independent of one another.

→ This is not always the case.

(44)

Another example. —

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n).$$



# of layers?

k s.t.

$\frac{n}{2^k} = 1$

$\longrightarrow k = \log_2 n.$

→ # of nodes at layer $i = 2^{i-1}$ (0-index)

Now do you just multiply.

$(\log_2 n) \cdot (\#\text{ nodes / layer})$ (# O(1) ?

this depends on layer!

No! you gotta sum it out!

$$T(n) = \sum_{i=1}^{\log_2 n} 2^{i-1} \cdot O(1)$$

↳ ind of which node layer.

↳ # nodes for node i.

$$= \left(\sum_{i=1}^{\log_2 n} 2^{i-1} \cdot \right) O(1)$$ fact: $\sum_{i=1}^{k} 2^{i-1} = 2^k - 1$

$$= 2^{\log_2 n} - 1 \cdot O(1)$$

$$= (n-1) \cdot O(1)$$

$$= O(n) \ldots$$

Compare to master theorem!

$$T(n) = a \cdot T\left(\left\lceil \frac{n}{b} \right\rceil\right) + O(n^d)$$

$a = 2, \; b = 2, \; d = 0.$

$\log_b a = \log_2 2 = 1 \qquad d = 0. \longrightarrow \log_b a > d$

↳ $T(n) = O(n^{\log_b a}) = O(n)$!

→ Questions?