PROOFS AND RUNTIME

PROOF REVIEW

Prove by contradiction: if n^2 is odd, then n is odd.

- Assume that n^2 is odd and n is even.
- $n = 2k$
- Therefore, $n^2 = 4k^2$, which is even. Contradiction!

Prove by contradiction: there are an infinite number of primes.

- Assume there are a finite number of primes.
- Therefore, there is a largest prime, p.
- p! is divisible by all primes $\leq p$
- Therefore $p!+1$ is divisible by no primes $\leq p$
- Every number has a prime factorization, so either $p!+1$ is prime (and larger than p) or its prime factorization contains primes only larger than p. Contradiction!

PROOF REVIEW

Prove or disprove: for any sets A, B, and C, if $A \times C = B \times C$, then $A = B$

Recall: $A \times B = \{ \langle a, b \rangle : a \in A \text{ and } b \in B \}$

Proof attempt:

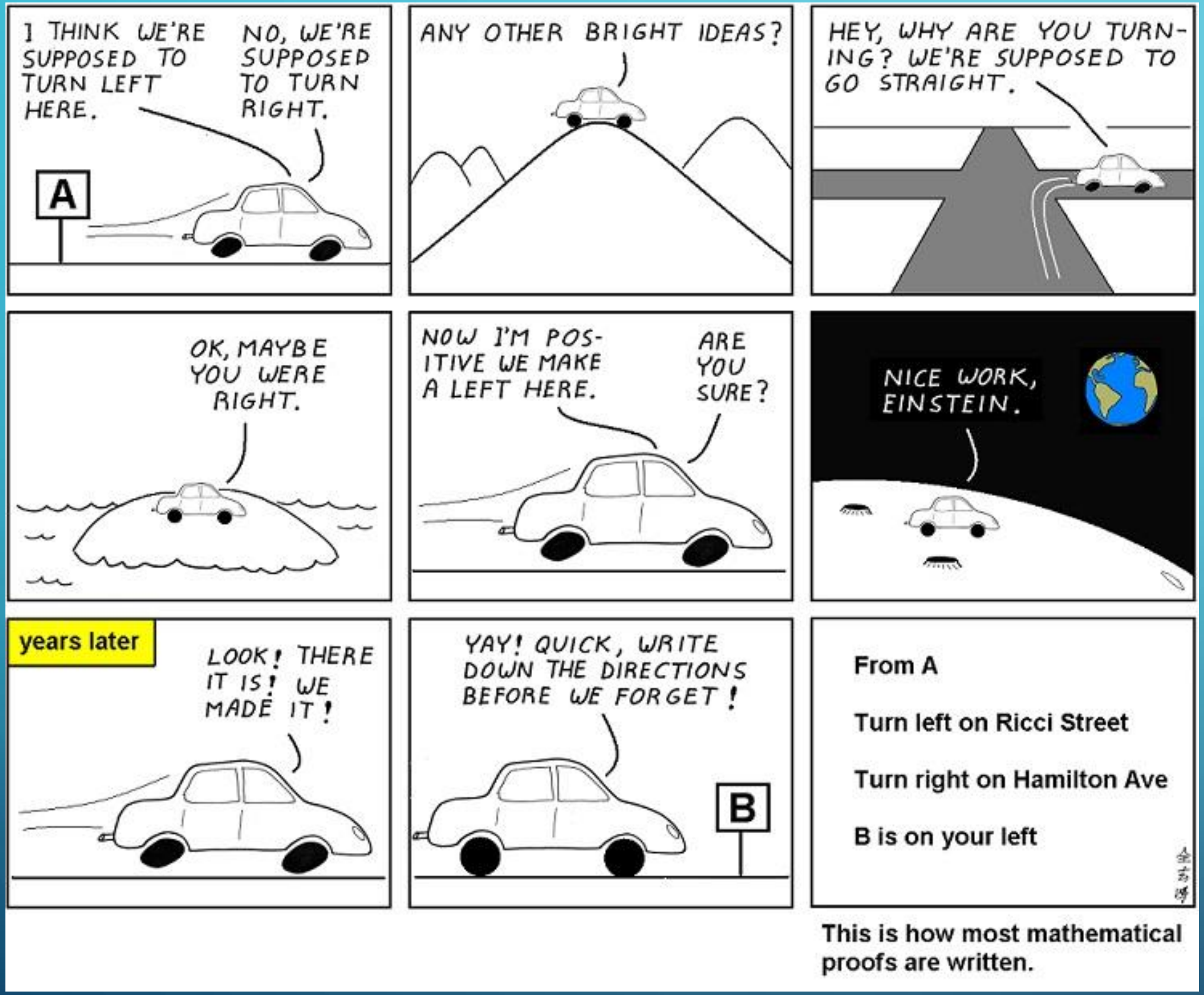
- Assume $A \times C = B \times C$, but $A \neq B$
- There must be an element in one of A or B which is not in the other set.
- Wlog, assume $a \in A$, but $a \notin B$
- Choose an arbitrary element $c \in C$.
- $\langle a, c \rangle \in A \times C$, but $\langle a, c \rangle \notin B \times C$, contradiction!

Are there any holes in the proof?

PROOF REVIEW

We're assuming there is an element from C to take!

- The proof works fine, except when $C = \phi$
- This identifies what our counter-example for the problem should be!
- Let $A = \{1, 2\}$, $B = \{2, 3\}$, and $C = \phi$. $A \neq B$, but $A \times C = B \times C = \phi$



ABSTRUCE GOOSE #230

After working on this proof for years, I have finally decided that it IS, in fact, obvious.

PROOF TIPS

- Run through some examples. This will help convince yourself the claim is true, as well as give an intuitive understanding for **why** it is true.
- Use the definition to translate a statement into mathematical form, when possible. This allows you to use the many rules of arithmetic to help prove it.
- When doing a proof by contradiction, make sure you are assuming the logical opposite. Make a truth table if you have to.
- Finding a proof is not a straight line from A to B. Even the most experienced research scientists take wrong turns. Just keep deriving stuff until you get what you need.
- If you don't know whether to prove or disprove a statement, follow your intuition. If you fail, you probably learned something about the problem: use this and try the other path.

PROVE: ANY 2^N X 2^N CHESSBOARD WITH ONE SQUARE REMOVED CAN BE TILED BY 3-SQUARE L-SHAPE PIECES, $\forall N \geq 1$

<u>Base Case</u>:

<u>Inductive Hypothesis</u>: The claim is true $\forall n : 1 \leq n \leq k$

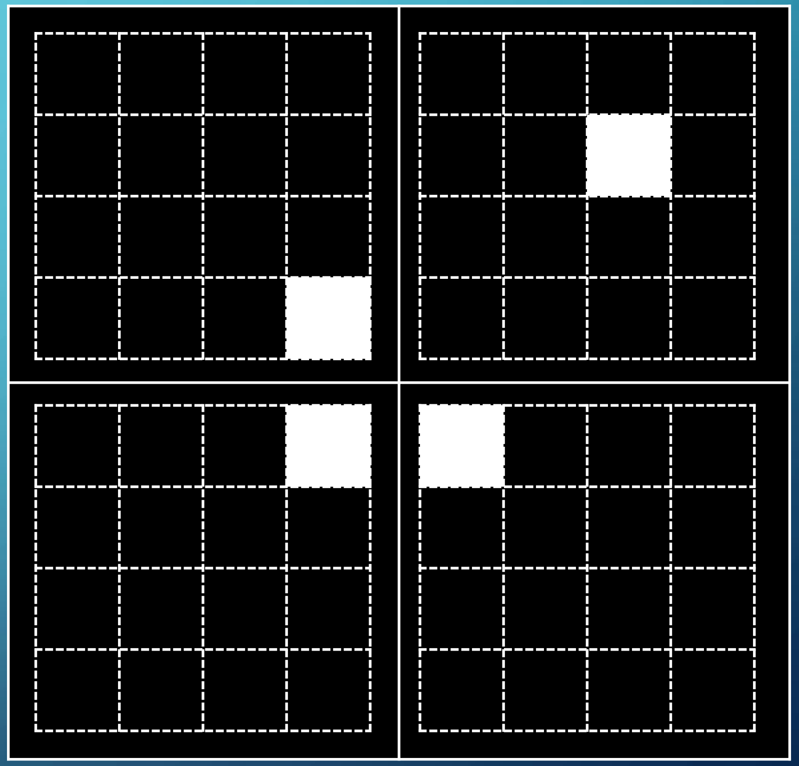
<u>Inductive Step</u>: Consider $n = k+1$.

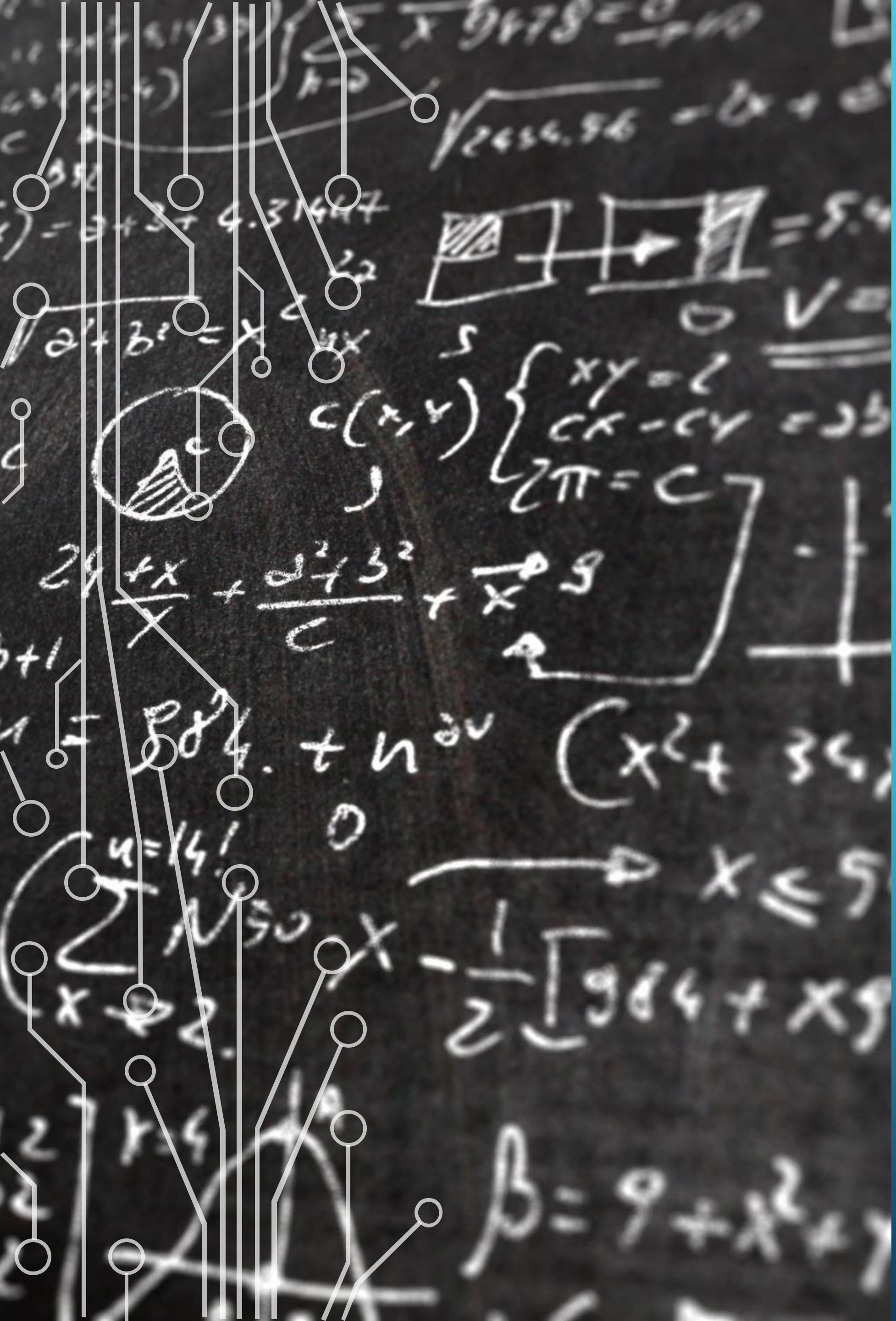
Split the board in four 2^k x 2^k quadrants.

Tile the quadrant with the missing square (by the I.H.)

Of the 4 center squares, remove 1 per remaining quadrant, tile the rest by the I.H.

Tile the 3 removed squares with one piece.





PROOF BY INDUCTION

Find the flaw in the proof that $a^n = 1$, for all non-negative integers n and all non-zero reals a.

Base Case: $a^0 = 1$

Inductive Hypothesis: $a^n = 1$, for all $n \leq k$

Inductive Step: $a^{k+1} = \frac{a^k \cdot ak}{a^{k-1}} = 1$, by the inductive hypothesis

To prove k+1, we are referring to k-1.

This falls apart when we let $k = 0$, because it refers to a^{-1}, which we haven't proven (and can't).

We would have to show a second base case to make this work.

RECURRENCE RELATIONS

```
Mergesort( A[1:n] )
    If (n == 1) then Return A
    B = Mergesort( A[1:n/2] )
    C = Mergesort( A[n/2+1:n] )
    Return Merge(B, C)
```

How do you analyze the runtime of a recursive function?

$f(n) = 2 \cdot f(n/2) + \Theta(n)$, $f(1) = \Theta(1)$

We need to solve the recurrence relation!

MERGESORT

$f(n) = 2 \cdot f(n/2) + x \cdot n$, $f(1) = y$

Hypothesize that $f(n) \leq c \cdot n \log n$, for all $n \geq 2$

<u>Base Case</u>: We need $f(2) = 2 \cdot y + 2 \cdot x \leq 2 \cdot c$. Choose c to be $\geq x + y$

<u>Inductive Hypothesis</u>: Assume $f(n) \leq c \cdot n \log n$, for all n: $2 \leq n \leq k$

<u>Inductive Step</u>: $f(k+1) = 2 \cdot f(\frac{k+1}{2}) + x \cdot (k+1)$

$\leq 2c \cdot \frac{k+1}{2} \log \frac{k+1}{2} + x \cdot (k+1)$, by the inductive hypothesis.

$= c \cdot (k+1) [(\log (k+1)) -1] + x \cdot (k+1)$

$= c \cdot (k+1) \cdot \log (k+1) + (x - c) \cdot (k+1)$

We want this to be $\leq c \cdot (k+1) \cdot \log (k+1)$, which is true if $c \geq x$.

It is, since we already chose c to be $\geq x + y$. Proven!

MERGESORT, CONT.

Does this prove that Mergesort takes $\Theta(n \log n)$?

- No, we only showed $O(n \log n)$. We'd need another proof to show $\Omega(n \log n)$!

Is it valid to use $n = 2$ as the base case?

- Yes, because O-notation asserts the claim is true for all $n \geq n_0$. We can choose $n_0 = 2$.

What would have happened if we tried $n = 1$ as our base case?

- It would have failed, since $1 \log 1 = 0$, and our algorithm doesn't take 0 time when $n = 1$. The claim is not true when $n = 1$.

We generally don't use induction to prove recurrences because it is difficult, and you need to already know the inductive hypothesis to even get started.

SOLVE-BY-TREE

$f(n) = 2 \cdot f(n/2) + cn,\ f(1) = d$

How much work at each level?

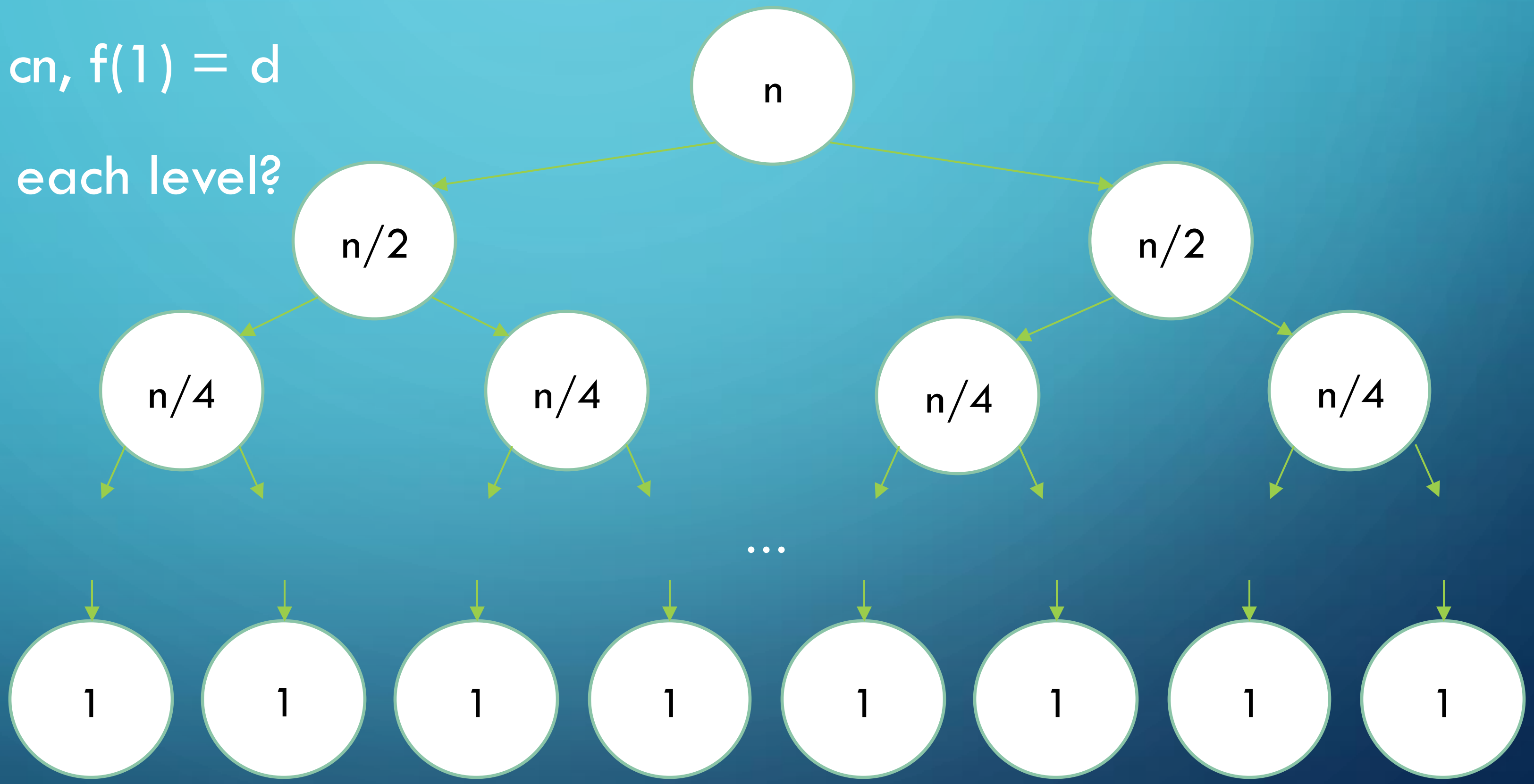
- cn

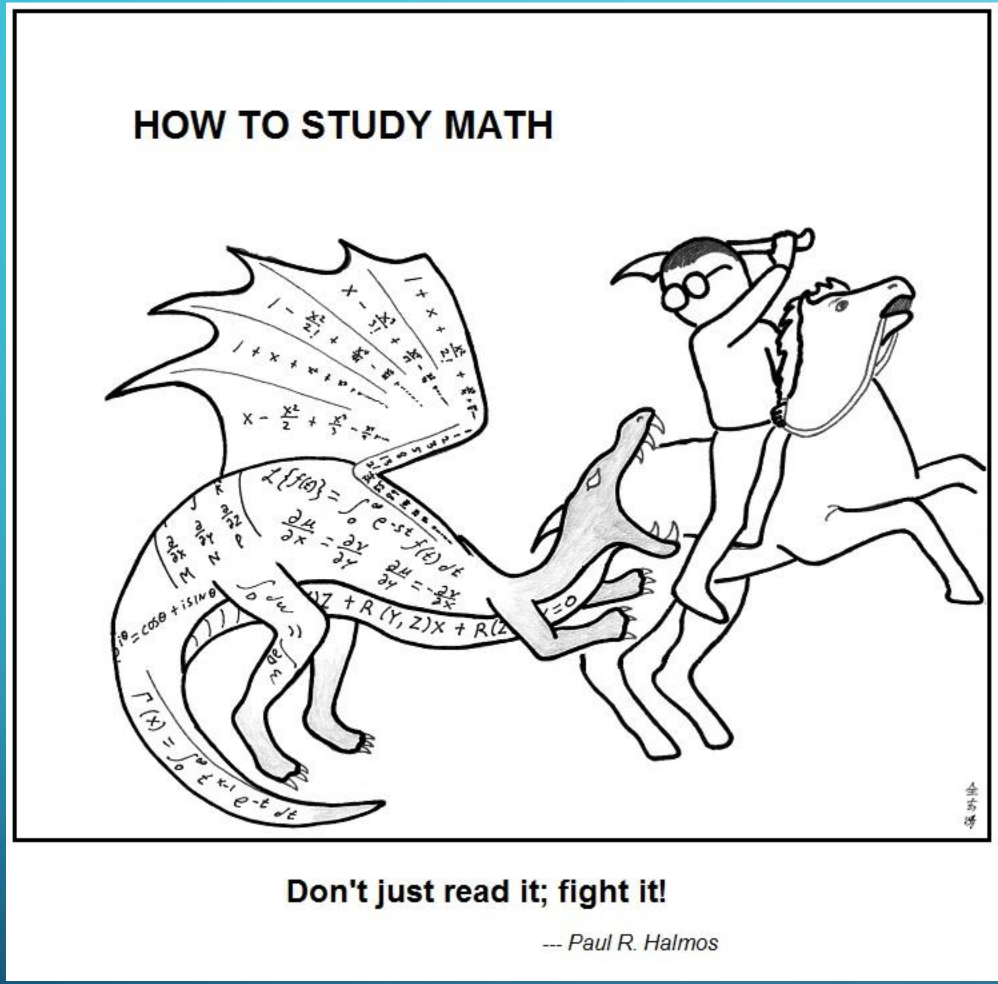
How many levels?

- log n

Total work =

$\Theta(n \log n)$





ABSTRUSE GOOSE #353

MASTER THEOREM

Master Theorem can solve (almost) any recurrence relation of the form

f(n) = a · f($^{n}/_{b}$) + g(n), for constants a ≥ 1 and b > 1.

Compare g(n) with $n^{\log a / \log b}$

- Case 1: If g(n) = $\Theta(n^{\log a / \log b})$, then f(n) = Θ(g(n) · log n)

f(n) = 2 · f($^{n}/_{2}$) + cn

- f(n) = Θ(n · log n)

MASTER THEOREM, CASES 2 AND 3

Case 2: If g(n) = $\Omega(n^{(\log a/\log b)\,+\,\varepsilon})$ for some $\varepsilon > 0$, then f(n) = Θ(g(n))

- $\Omega(n^{\log a/\log b})$ is true when case 1 is true, so we're (kind of) saying that g(n) must be strictly larger.

What should case 3 be?

Case 3: If g(n) = $O(n^{(\log a/\log b)\,-\,\varepsilon})$ for some $\varepsilon > 0$, then f(n) = $\Theta(n^{\log a/\log b})$

MASTER THEOREM ANALYSIS

$g(n)$

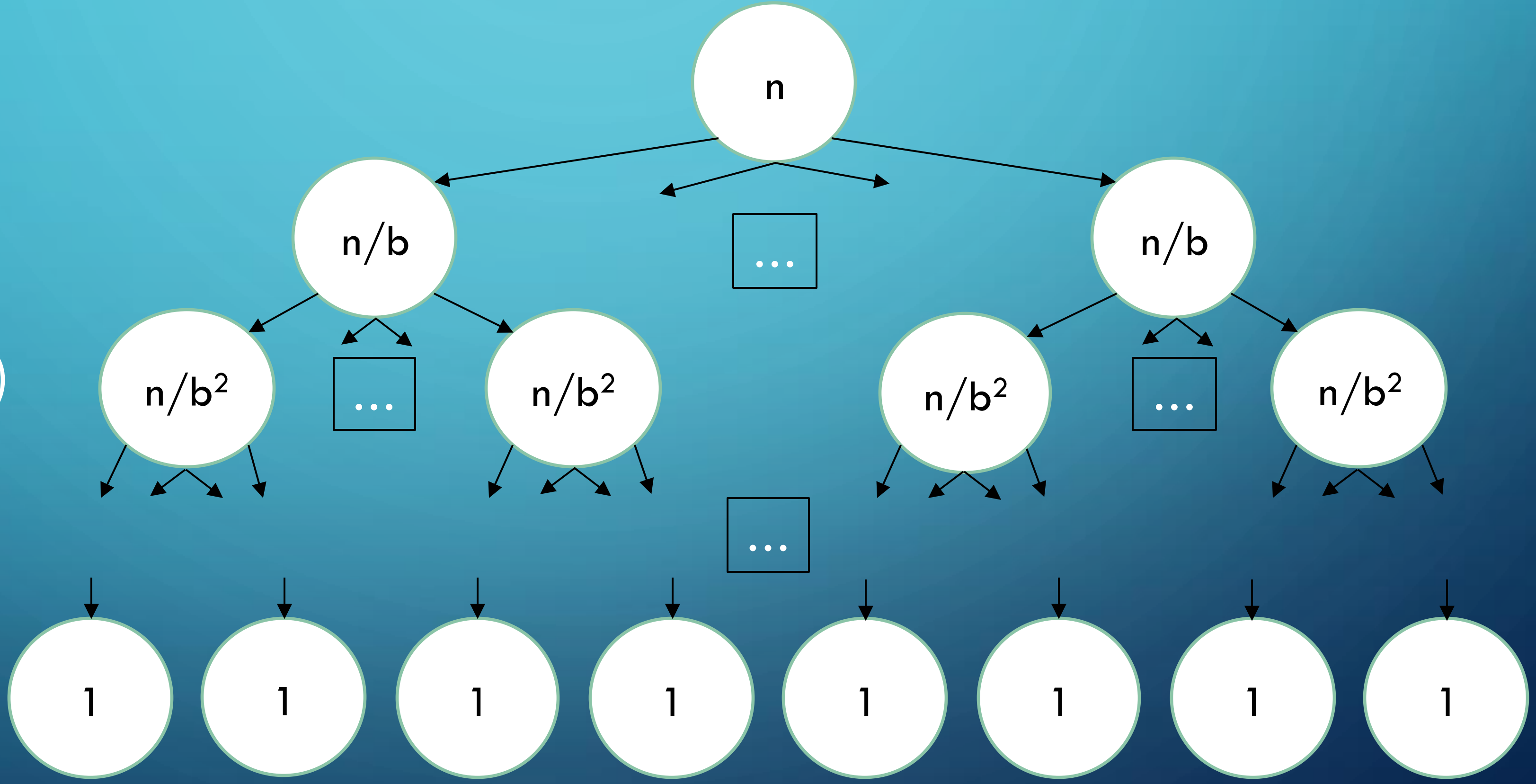
$a \cdot g(\, n/b \,)$

$a^2 \cdot g(\, n/b^2 \,)$

$a^{\log_b n}$

$= n^{\log_b a}$

$= n^{\log a / \log b}$



PRACTICE

$f(n) = f(n/2) + 1$

- $f(n) = \Theta(\log n)$

$f(n) = 8 \cdot f(n/2) + 1000n^2$

- $f(n) = \Theta(n^3)$

$f(n) = 2 \cdot f(n/2) + n^2$

- $f(n) = \Theta(n^2)$

LIMITS OF MASTER THEOREM

$f(n) = 2 \cdot f(\,^{n}/_{2}\,) + \,^{n}/_{\log n}$

- $n > \,^{n}/_{\log n}$, so does that mean the runtime is $\Theta(n)$?
- We need $g(n) = O(\, n^{(\log a/\log b) - \varepsilon}\,)$
- So we need $n^{1-\varepsilon} > \,^{n}/_{\log n}$
- That means $^{n}/_{n^{\varepsilon}} > \,^{n}/_{\log n}$
- Alternatively, $n^{\varepsilon} < \log n$
- …which isn't true.

INTERPRETING MASTER THEOREM

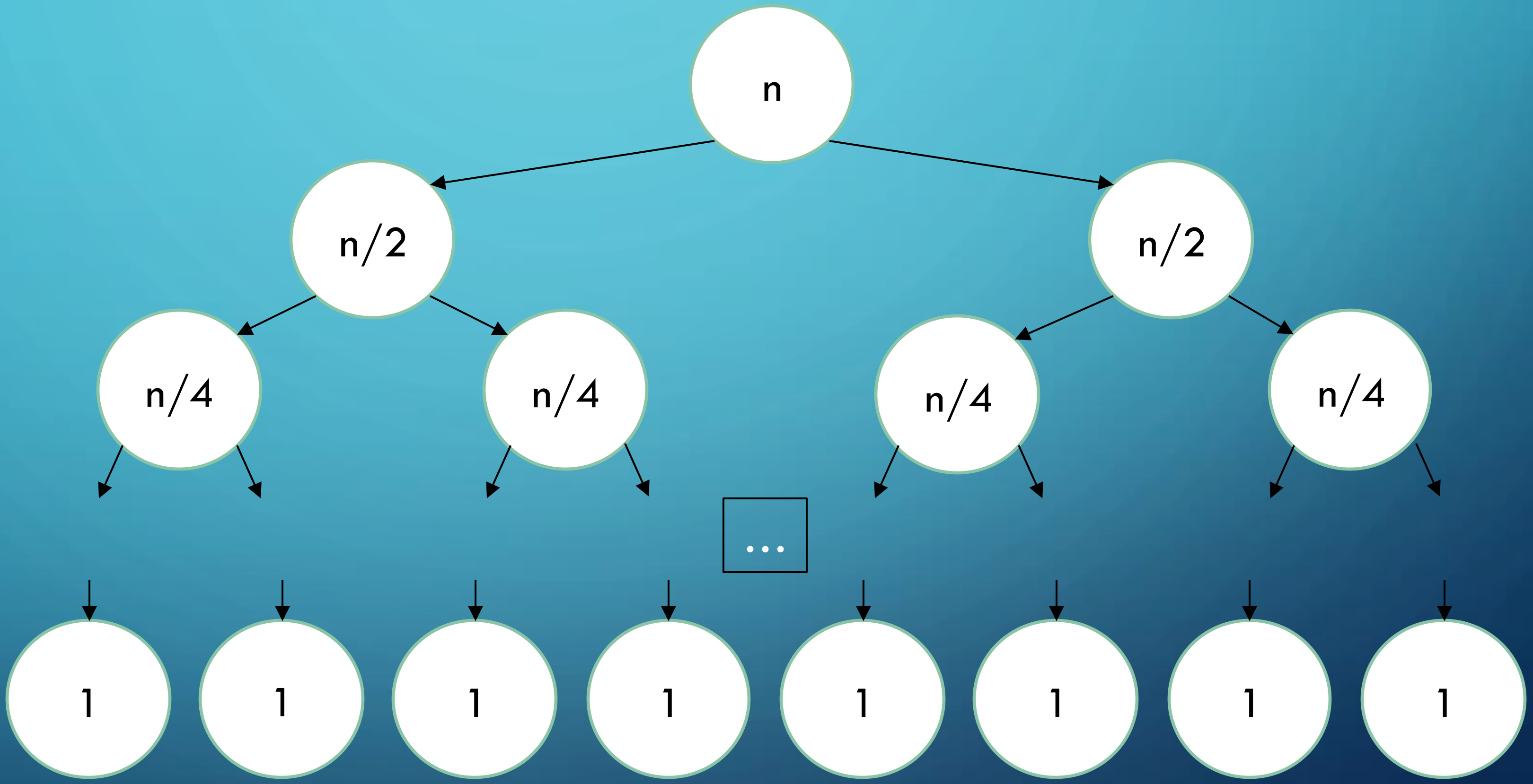
- If the work done on the bottom level is a **polynomial-factor** bigger than the work on the top level, then the total work is the work on the bottom level.
- If the work done on the top level is a **polynomial-factor** bigger than the work on the bottom level, then the total work is the work on the top level.
- If the difference between the top and bottom level is greater than a constant, but smaller than a polynomial, then master theorem **will not help us.**

SOLVING F(n) = 2 · F($^n/_2$) + $^n/_{\log n}$

$= {}^{n}/_{\log n}$

$= {}^{n}/_{(\log n)-1}$

$= {}^{n}/_{(\log n)-2}$



SOLVING F(n) = 2 · F($^n/_2$) + $^n/_{log\, n}$

$$\sum_{i=1}^{\log n} {}^{n}/_{i} = n \sum_{i=1}^{\log n} {}^{1}/_{i}$$
$$= n \log(\log n)$$

The runtime is Θ(n log log n)

We would have to also use solve-by-tree if we wanted to solve, for example,

f(n) = f($\sqrt{n}$) + 1

A DIFFERENT FORM OF RUNTIME ANALYSIS

Recall that a vector (from the STL) is implemented using an array.

What is the worst-case runtime for the pushback function?

- Is it $O(1)$?
- If the array is full, we'll need to double the size of the array, which takes $\Theta(n)$ time!
- It is correct to say that pushback takes worst-case $\Theta(n)$ runtime.
- This analysis seems rather unfair, given that the worst-case will happen rarely, and at predictable intervals.

AMORTIZED RUNTIME

We could accurately say that the average runtime for pushback is O(1).

- This still doesn't capture everything: that implies that if we get bad luck, the average will be worse than O(1).
- There is no luck involved: we know exactly how many inputs will be required to produce the worst-case scenario, and it will always be the same effect.
- **Amortized Runtime** is a blend between average-case and worst-case. It is kind of the "worst-case average-case".

AMORTIZED RUNTIME

If the first x operations take a total of $\Theta(y)$ time, then the average time per operation is $\Theta({}^{y}/_{x})$.

- The amortized runtime chooses the number and sequence of operations that produces the worst-possible average runtime.
- It is like the "worst-case average-case".
- Assume that the array starts at size 1, and you do n inserts. What is the amortized runtime for pushback?

PUSHBACK ANALYSIS, METHOD 1

There will be a few expensive pushbacks, when we have to resize the array.

How costly is an expensive pushback?

- Θ(i), where i is the current size of the array.

How many expensive pushbacks will there be?

- log n

The total runtime is $\sum_{i=1}^{\log n} 2^i + (n - \log n) = \Theta(n)$

So the average time per operation is O(1). Guaranteed!

PUSHBACK ANALYSIS, METHOD 2

Let a new "phase" start just after the array has resized.

Analyze the amortized runtime for an arbitrary phase:

- The array has just grown to size n, because we inserted $1 + {}^{n}/_{2}$ things.
- We insert ${}^{n}/_{2}$ things this phase, all but one of them take O(1) time.
- The last thing takes Θ(n) time.

Amortized runtime $= \frac{1 \cdot n + ({}^{n}/_{2} - 1) \cdot 1}{{}^{n}/_{2}} = \Theta(1)$

PUSHBACK ANALYSIS, METHOD 3

Every time we call pushback, we pay 5 dollars.

- Cheap operations only require 1 dollar, so we place the excess in a piggy bank.
- When we get to an expensive operation, the last $^{n}/_{2}$ things have each paid 4 extra dollars.
- We need to make an array of size 2n, so we have one dollar for each index we need to make: we always have enough money saved up!
- $5 = \Theta(1)$, so the amortized runtime is constant.

PRACTICE

We are using a Boolean array as a binary counter.

- Each index starts at 0 (false), and the counter counts up in binary.
- Some increments (from 1010 to 1011, for example) require only constant time.
- Other increments (from 01111111 to 10000000) take a long time.

What is the worst-case runtime of our increment function?

- $\Theta(\log n)$, since if we insert n times, we require log n bits.

AMORTIZED ANALYSIS OF THE BINARY COUNTER

Starting at the least significant bit, if the current bit is a 0, we flip it and stop. Otherwise we flip the 1 to a 0 and continue to the next bit.

- We will always flip a single 0 to a 1.
- We will flip a variable number of 1s to 0s.

We will use the piggy bank method (method 3) to solve this.

PRACTICE

When we call the increment function, we pay 2 dollars. Every bit takes a single dollar to flip, from either 0 to 1 or 1 to 0.

All of the bits start at 0.

- Whenever we flip a bit from 0 to 1, we spend both of our 2 dollars towards that bit. 1 dollar to cover the immediate costs, and the other dollar to be stored for when it eventually flips from 1 to 0.
- Since only a single bit flips from 0 to 1 every increment, we always have enough money saved up for the 1s that flip to 0s.
- Since $2 = \Theta(1)$, this takes amortized constant time!

TAKE-HOME PRACTICE

- Prove for all integers n: n is odd iff 3n+1 is even.
- Use induction to prove that $\sum_{i=0}^{n} \frac{1}{2^i} < 2$
- Chapter 2, exercises 3, 4, 5, 6
- Challenge problem: Chapter 2, exercise 8

Solve the following recurrence relations:

- $\mathrm{f(n)} = \mathrm{f}({}^{n}/_{4}) + \sqrt{n}$
- $\mathrm{f(n)} = \mathrm{f}({}^{n}/_{4}) + 1$
- $\mathrm{f(n)} = \mathrm{f}({}^{n}/_{4}) + \log n$