Midterm Review!!!

Announcements

- **Exam 1 is on Thursday!!**
  - 6-9pm!
  - Please let us know if you don't get an email with your exam location by EOD today (Monday).
  - See course website (under the HW/Exams tab) for a practice exam, etc.
- **Special OH! Tuesday May 2, Huang basement, 10am-6:30pm!**
  - No homework party or Section this week
  - No OH on Thursday/Friday, and we'll also be de-activating Ed temporarily.

Announcements

- Slight deviation from HW schedule:
  - HW4 will be released Wednesday 5/3 as usual.
  - You will have until FRIDAY 5/12 to hand it in! Two-day extension!
    - Up to two late days gives a late deadline of Sunday 5/14.

- A note about grading:
  - Some folks have told us they are concerned about either:
    - The midterm being hard and so everyone will get a bad grade in the class
    - The midterm being easy and then things will get curved down
  - We **will not** "curve down" in any situation!
  - If the midterm is hard, we **will** curve up!

It will be okay!!!!

- I don't think that the exam is easy.
  - This is on purpose: it raises the signal-to-noise ratio and makes grades a better reflection of your knowledge.
- That means it's okay if you don't get every question right!
  - Exam tip: if you get stuck, move on and come back to it later.
- Exam philosophy: we have done our best to have:
  - No "trick" questions
  - No tricky "aha" moments needed (except bonus pts) – just understand the concepts/facts/skills well!

Agenda

1. A **recap** about hash tables.
2. A **quick recap** of everything else we’ve seen so far.
3. If time, answering (more) questions!

Recap of Hash Tables

*Assuming h(x) takes O(1) time to evaluate.

Hash tables

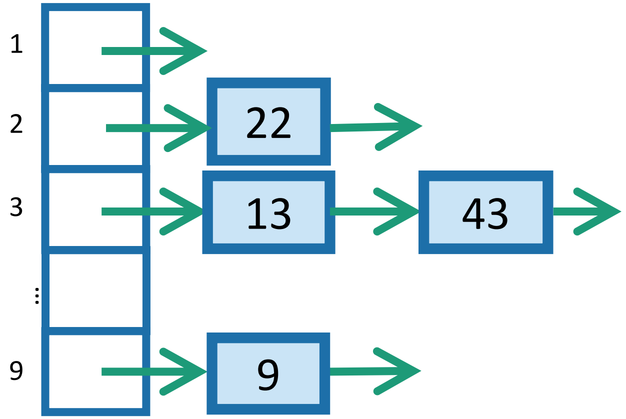
- A hash table:
  - Stores items from a universe U
  - Supports INSERT/DELETE/SEARCH

- A **hash table** consists of:
  - An array A of n “buckets,” each of which contains a linked list
  - A hash function $h: U \to \{1, \ldots, n\}$

Time O(1)
- INSERT(x):
  - Insert x into A[h(x)]

Time O(len(A[h(x)]))
- SEARCH(x):
  - Go through A[h(x)] to look for x

Time O(len(A[h(x)]))
- DELETE(x):
  - Go through A[h(x)] to look for x
  - If you find it, delete it



n buckets (say n=9)

Question: How do we pick the hash function?

- We saw last week that if an adversary knows the hash function h ahead of time, we will never do well in a worst-case model.
  - There will be some x so that A[h(x)] is very full.
- Instead we choose h **randomly**.
  - Uniformly randomly is a bad idea: how will we store the function h????
- In more detail, we choose randomly from a **Universal Hash Family**.

Universal Hash Families

- H is a ***universal hash family*** if, when h is chosen uniformly at random from H,

$$\text{for all } u_i, u_j \in U \quad \text{with } u_i \neq u_j,$$
$$P_{h \in H}\{\, h(u_i) = h(u_j)\} \leq \frac{1}{n}$$

In English: When we choose $h \in H$ uniformly at random (out of all functions in H), the probability of any two elements colliding is no larger than what it would be if we chose h uniformly at random (out of all possible functions).

Example

- Choose a prime p so that $|U| \leq p \leq 2|U|$.
- Define $h_{a,b}(x) = ax + b \quad mod\ p \quad mod\ n$
- H is the set of all such $h_{a,b}$ for $1 \leq a \leq p - 1; 0 \leq b \leq p - 1$.

- To choose h uniformly at random from H, just choose a random a, b.

- *We did not prove in class that this H is a universal hash family, and you are not responsible for that proof.*

So the whole setup is:

- Initialize(p): //p is prime, so that $|U| < p$
  - Store p.
  - Choose a random a, b and store them.
  - Def h(x):
    - Return ax + b mod p mod n
  - Initialize an array A with n buckets.

- INSERT(x):
  - Insert x into A[h(x)]
- SEARCH(x):
  - Go through A[h(x)] to look for x
- DELETE(x):
  - Go through A[h(x)] to look for x
  - If you find it, delete it

Space:

- $O(n)$ to store the buckets
- $O(\log |U|)$ per item in U that we INSERT
- $O(\log |U|)$ to store a, b, p
- $\Rightarrow O(\, n \log |U| \,)$, assuming we store O(n) items.

*Assuming h(x) takes O(1) time to evaluate.

So the whole setup is:

- Initialize(p): //p is prime, so that $|U| < p$
  - Store p.
  - Choose a random a, b and store them.
  - Def h(x):
    - Return ax + b mod p mod n
  - Initialize an array A with n buckets.

Time O(1)

- INSERT(x):
  - Insert x into A[h(x)]
- SEARCH(x):
  - Go through A[h(x)] to look for x
- DELETE(x):
  - Go through A[h(x)] to look for x
  - If you find it, delete it

Time O(len(A[h(x)]))

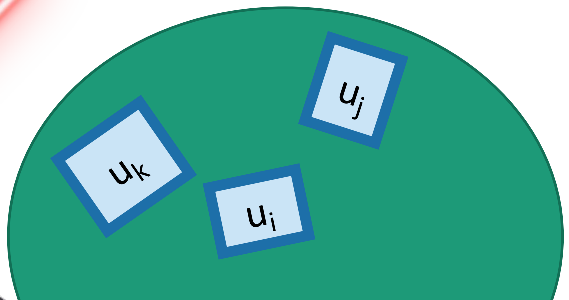
Time:

- What is len(A[h(x)]) after INSERTing n items?
- Worst-case: $O(n)$
- Expected: E[len(A[h(x)])] = E[number of items that collide with x] = $O(1)$.

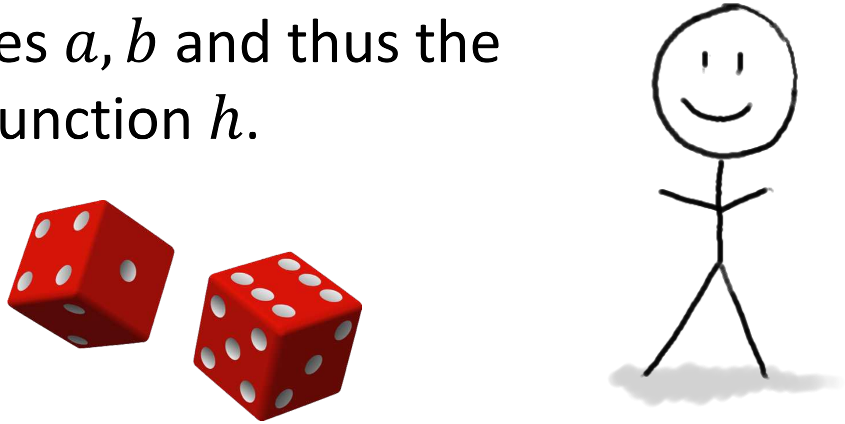
The game

1. An adversary chooses any n items $u_1, u_2, \ldots, u_n \in U$, and any sequence of L INSERT/DELETE/SEARCH operations on those items.

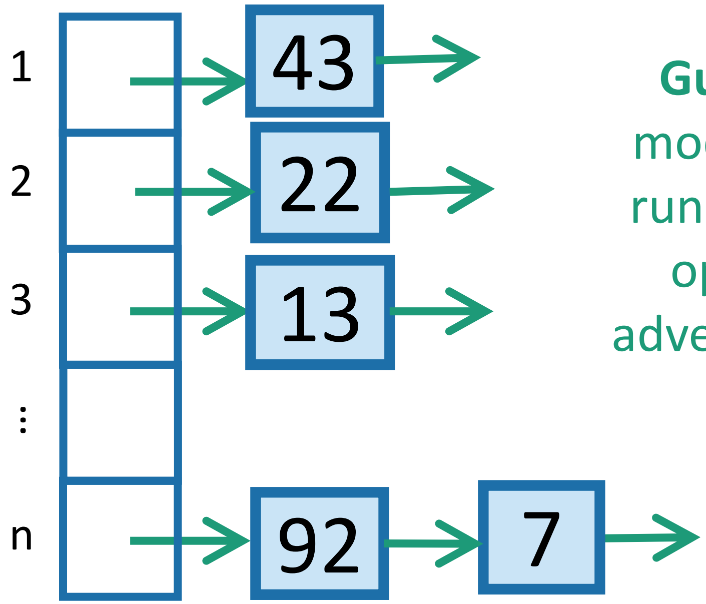
INSERT 13, INSERT 22, INSERT 43, INSERT 92, INSERT 7, SEARCH 43, DELETE 92, SEARCH 7, INSERT 92



2. You, the algorithm, chooses a, b and thus the hash function h.



3. HASH IT OUT



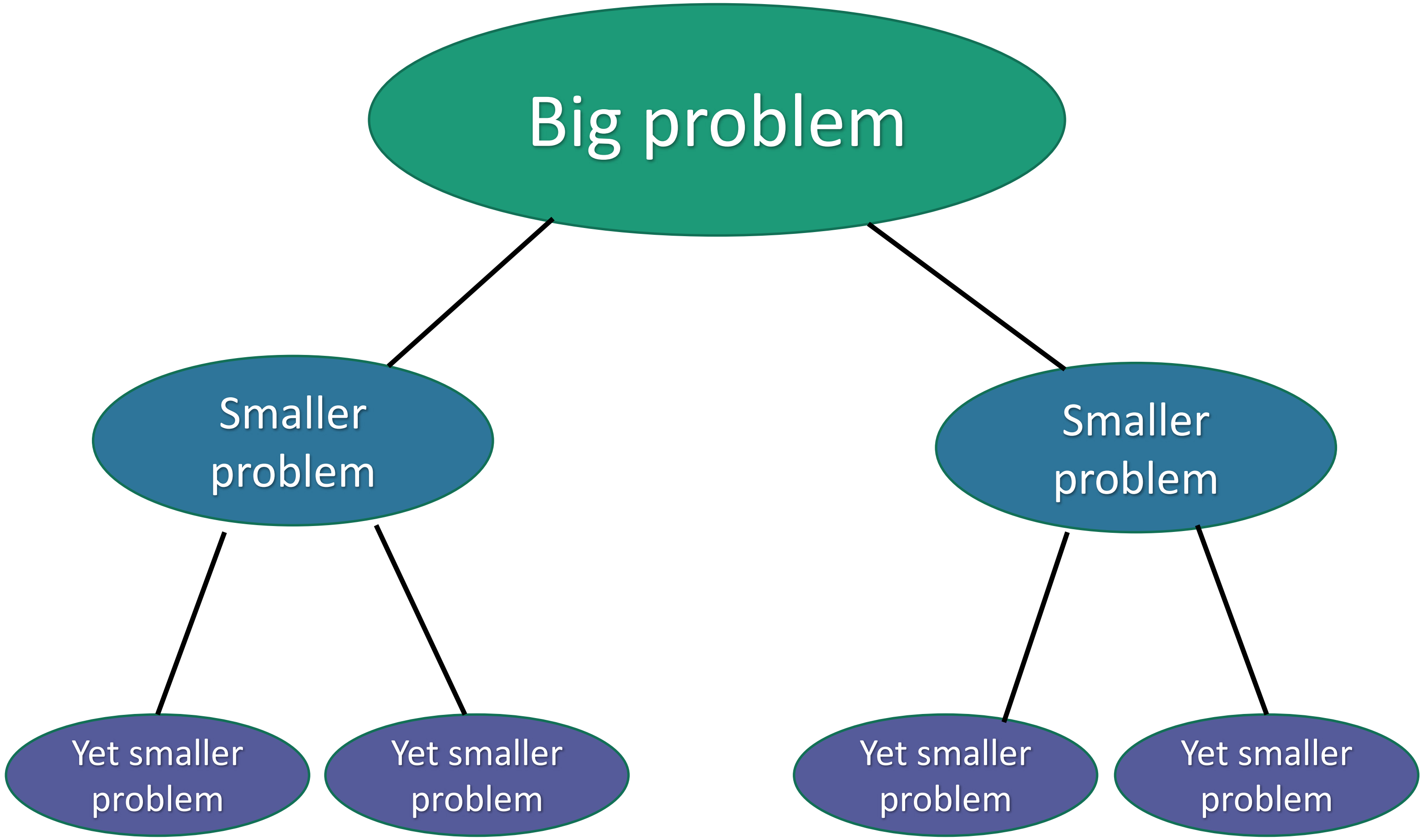
Guarantee: In this model, the expected running time of each operations in the adversary's list is $O(1)$

Questions about hash tables?

Real quick recap of everything else!

Lecture 1

- Divide and Conquer!

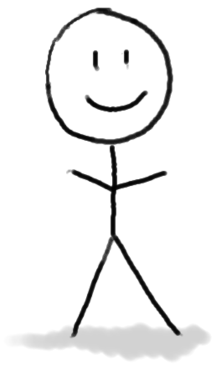


- **Karatsuba integer multiplication**: divide-and-conquer algorithm for multiplying n-digit numbers that runs in time $O\left(n^{\log_2 3}\right) = O(n^{1.6})$

Lecture 2: Worst-case analysis

Think of it like a game:

Worst-case analysis guarantee:
`Algorithm` should work (and be fast) on that worst-case input.



Here is my algorithm!

```
Algorithm:
  Do the thing
  Do the stuff
  Return the answer
```

Algorithm designer

HERE IS AN INPUT!
(WHICH I DESIGNED TO BE TERRIBLE FOR YOUR ALGORITHM!)

- Pros: very strong guarantee
- Cons: very strong guarantee

*Not the only way to do it, but this is typically how it goes in this class.

Lecture 2: Proving that an algorithm is correct

- Often* we use ***induction***!

- For a recursive algorithm:
  - Inductive hypothesis is often* "The algorithm is correct on inputs of size up to n"
  - Examples: MergeSort (Lecture 2); DuckTroupeSort (HW1); SELECT (Lecture 4 Handout)

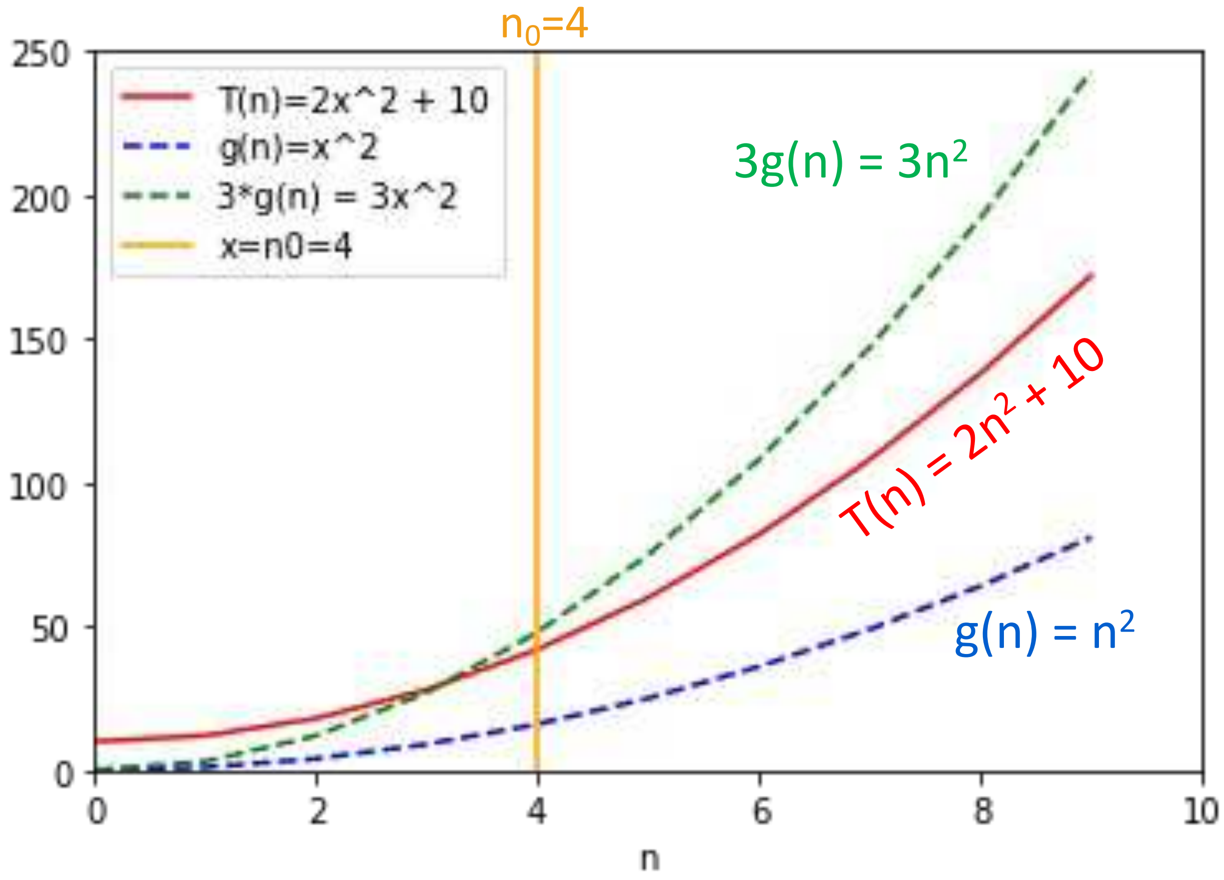
- For an iterative algorithm:
  - Inductive hypothesis is often* "After iteration i, [things are going the way they should be]."
  - Examples: InsertionSort (Lecture 2 Handout); Proof that RadixSort is correct (Lecture 6)

Lecture 2: Asymptotic Notation

- Informally, $T = O(g)$ means that T grows "about the same order of, or slower, than g."

- Formally, $T = O(g)$ means that there is some $c, n_0 > 0$ so that for all $n \geq n_0$, $T(n) \leq c \cdot g(n)$.

$O(\ )$ is an upper bound
$\Omega(\ )$ is a lower bound
$\Theta(\ )$ is both



Lecture 2: MergeSort!

- Divide-and-conquer
- Runs in (worst-case) time $O(n \log n)$

- Basic idea:
  - Recursively sort left and right halves
  - Merge them!

Lecture 3: The master theorem

We can also take n/b to mean either $\left\lfloor \frac{n}{b} \right\rfloor$ or $\left\lceil \frac{n}{b} \right\rceil$ and the theorem is still true.

- Suppose that $a \geq 1, b > 1,$ and d are constants (independent of n).
- Suppose $T(n) = a \cdot T\left(\frac{n}{b}\right) + O(n^d)$. Then

$$T(n) = \begin{cases} O\left(n^d \log(n)\right) & \text{if } a = b^d \\ O\left(n^d\right) & \text{if } a < b^d \\ O\left(n^{\log_b(a)}\right) & \text{if } a > b^d \end{cases}$$

Three parameters:

a : number of subproblems

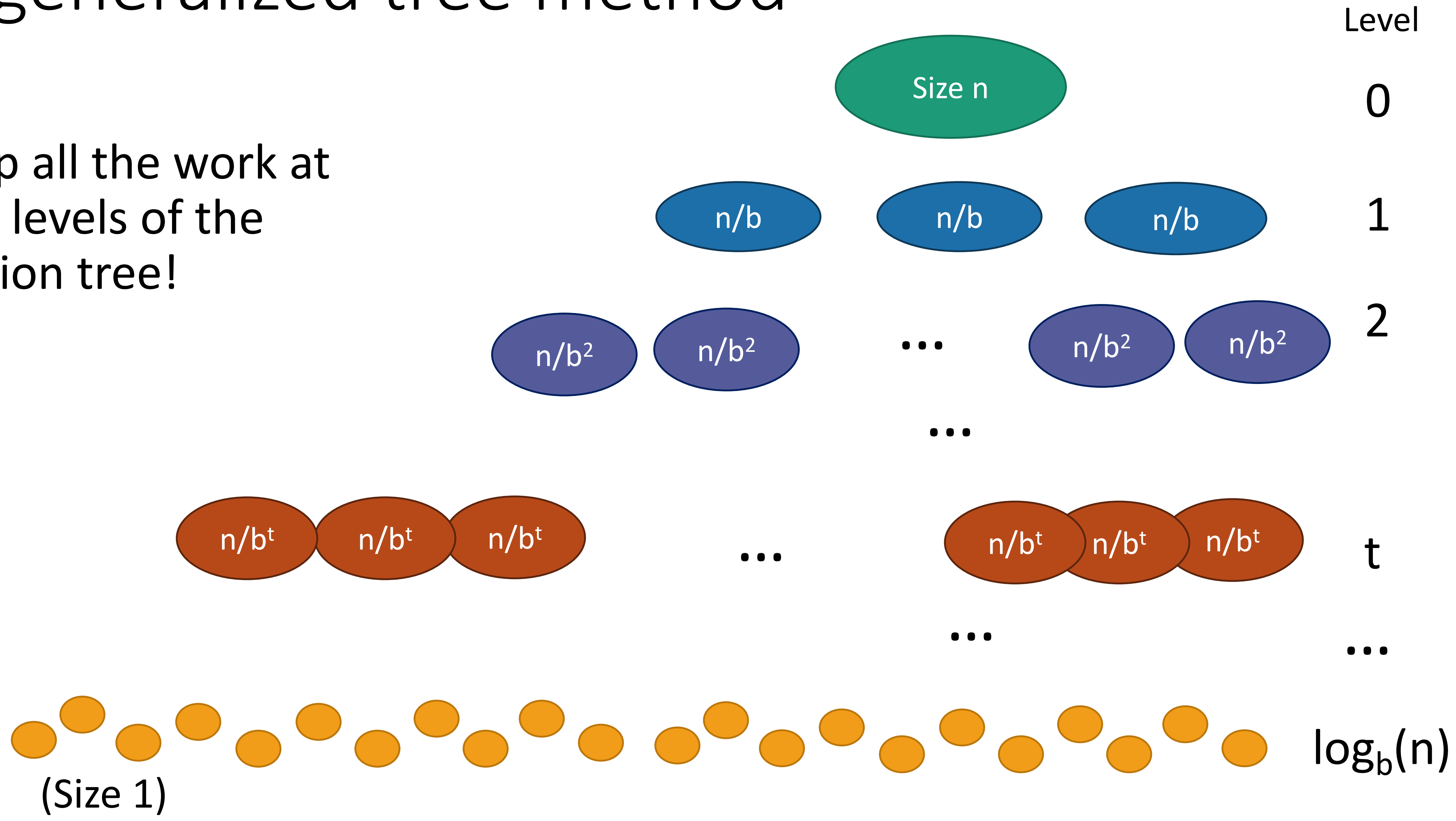
b : factor by which input size shrinks

d : need to do n^d work to create all the subproblems and combine their solutions.

Many symbols those are….

Proof: generalized tree method

- Add up all the work at all the levels of the recursion tree!



Lecture 3: Substitution Method aka "Guess and Check"

- Step 1: Generate a guess at the correct answer.
- Step 2: Try to prove that your guess is correct.
- (Step 3: Profit.)

Examples!

- One from Lecture 3
- Another from Lecture 3 at the end that we skipped (but check out the slides)
- Lecture 4 (SELECT) analysis. See also Section 6.4.5 in Algorithms Illuminated text.
- HW2, Exercise 3

How to guess?

- "Unrolling" the recurrence relation
- Doing a few examples
- Divine inspiration or meta-analysis
- Hope

How to prove guess is correct?

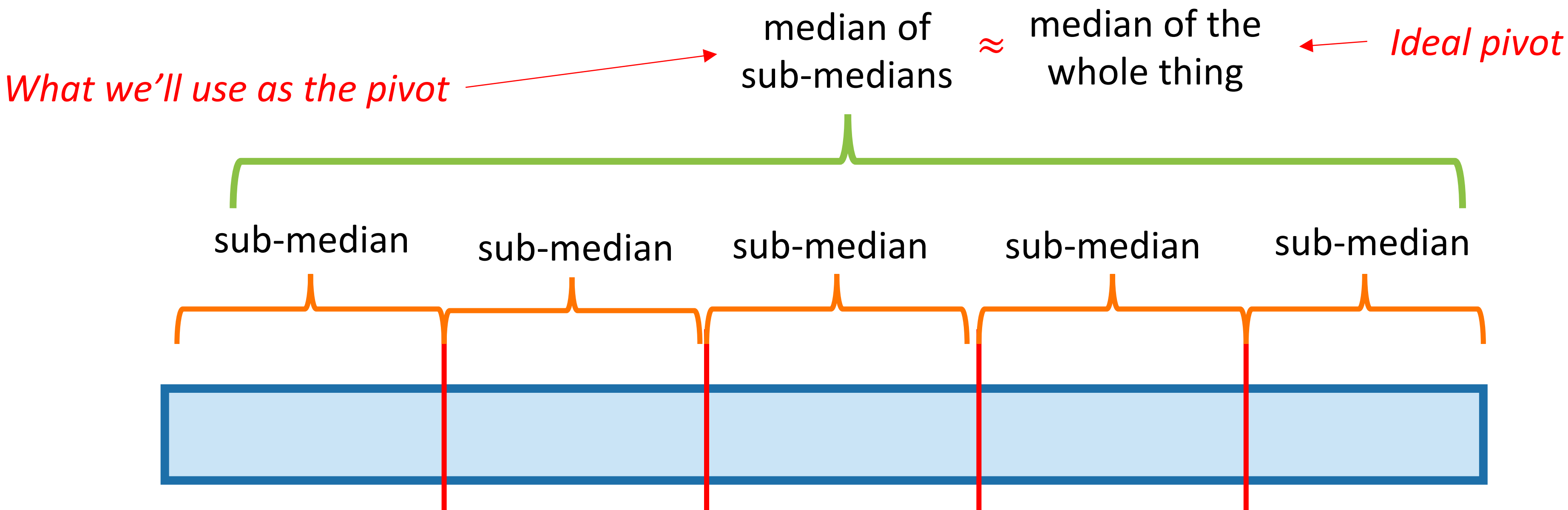
- Proof by induction!
- IH is (often*): "My guess holds for all values up to n."

How to profit?

- Write down a nice proof!
- (Take it over to Sand Hill Road?)

Lecture 4: k-Select

- Deterministic $O(n)$ time algorithm to find the k^{th} smallest element in an array.
- Algorithm idea:
  - Partition around a pivot, and recursively search on either right or left side.
  - To pick the pivot, recursively find a "median of medians"



Aside: Common question on Ed:

How did we come up with the recurrence relation for SELECT?

- Let T(n) = running time of SELECT
- What does SELECT do?
  - Find a pivot that gives us at worst a 3n/10 – 7n/10 split. Time...TBD
  - Partition around the pivot Time O(n)
  - Recursively call select on a list of size at most 7n/10. Time at most T(7n/10)
- How do we find the pivot?
  - Break up the list into chunks of size 5
  - Find the median of each of those small lists Time O(n)
  - Recursively call SELECT to find the median of the n/5 medians Time T(n/5)

So $T(n) = T(n/5) + T(7n/10) + O(n)$

$T(n) = T(n/5) + T(7n/10) + O(n)$

Aside: Common question on Ed:

How do we solve this recurrence relation?

- Substitution / Guess and check method!
- We guess O(n)
  - Wishful thinking, some experiments, the fact that "n" is sitting right there.
- More precisely, we guess $T(n) \leq 10\, n$
  - By ***trying*** to do a proof by induction and playing around to find out what IH works!

Note: it's mathematically legit to just start off by guessing $T(n) \leq 10000n$. Just not very elegant.

- To check our guess, we do proof by induction!
  - Inductive hypothesis: $T(n) \leq 10\, n$
  - Check out Lecture 4 for the actual proof by induction
  - It's not any trickier than any other proof by induction, the only tricky part was coming up with the right guess!

Lecture 5: Randomized Algorithms

- Scenario 1: Expected Running time

1. You publish your algorithm.
2. Bad guy picks the input.
3. You run your randomized algorithm.

- Scenario 2: Worst-case Running time

1. You publish your algorithm.
2. Bad guy picks the input.

1. Bad guy chooses the randomness (fixes the dice) and runs your algorithm.

- In **Scenario 1**, the running time is a **random variable.**
  - It makes sense to talk about **expected running time**.
  - This still counts as "worst-case analysis" since there's a bad guy, but it's not the same as "worst-case running time."
- In **Scenario 2**, the running time is **not random**.
  - We call this the **worst-case running time** of the randomized algorithm.

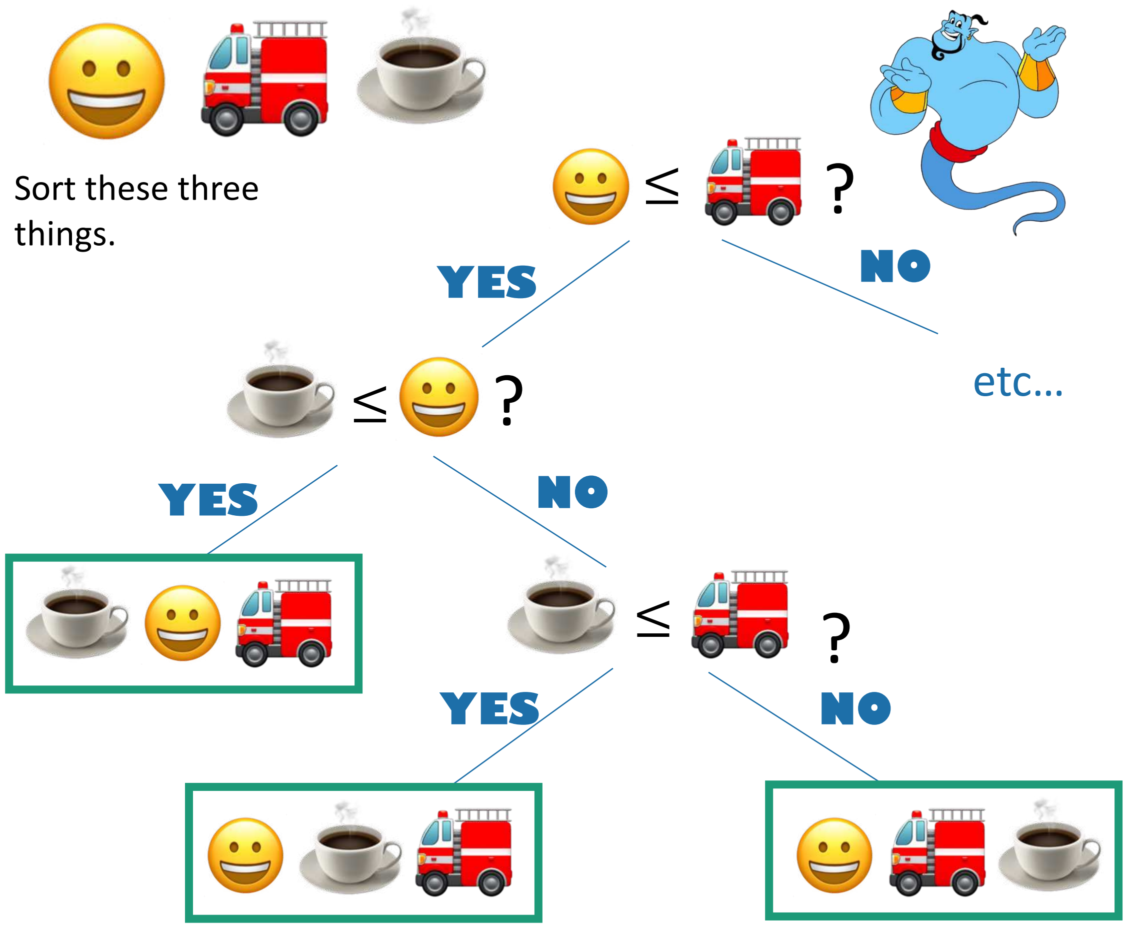
Lecture 5: QuickSort

- Runs in expected time $O(n \log n)$, worst-case time $O(n^2)$
- Divide and Conquer!
- Pick a **random** pivot.
- Partition around the pivot, recurse on left and right halves.

- Analysis was a bit tricky!
  - It's NOT okay to just say "$E[|L|], E[|R|] = \frac{n}{2}$, so in expectation the relevant recurrence relation is $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$"
  - Instead we counted comparisons, and looked at the expected number of those.
  - Key tool: linearity of expectation!

Lecture 6: Comparison-based lower bound

- Comparison-based model:
  - Can only interact with values by comparing items to each other.
  - QuickSort, MergeSort, InsertionSort all follow this model.
- **Theorem:** Any comparison-based sorting algorithm needs $\Omega(n \log n)$ comparisons.
- Proof idea:
  - Look at the decision tree corresponding to the algorithm.
  - It has at least $n!$ leaves
  - So it has depth at least $\log(n!) = \Omega(n \log n)$.



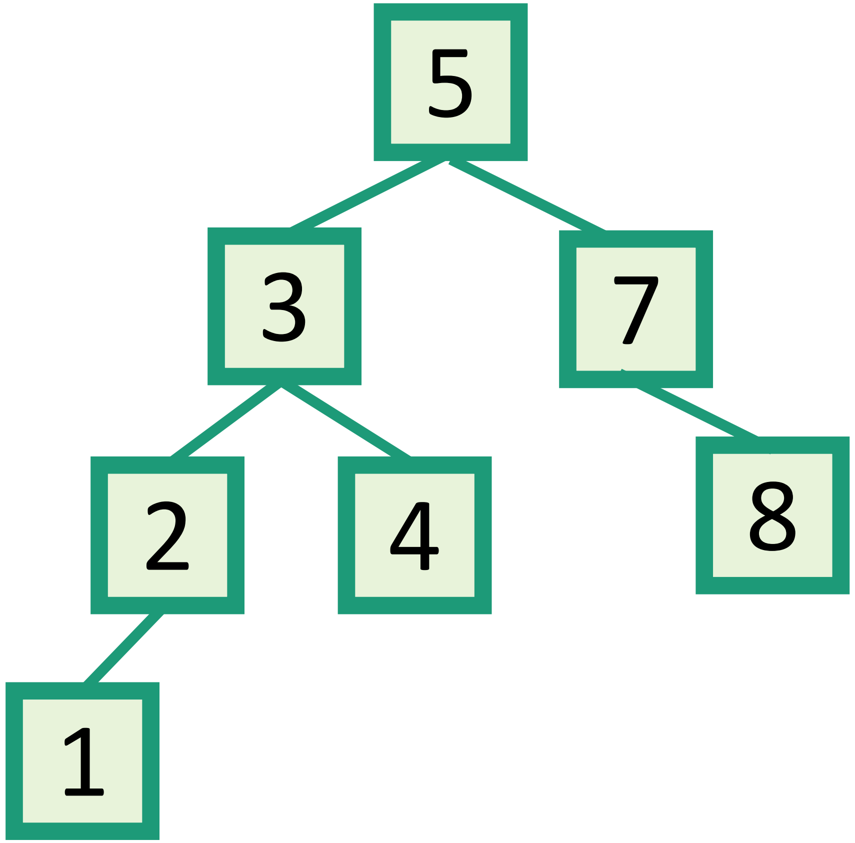
Lecture 6: CountingSort and RadixSort

- CountingSort idea:
  - If you are sorting integers that are all between, say, 1 and 10, put everything into buckets labeled 1,2,…,10.
  - Then take them out in order.
- RadixSort idea:
  - If you have d-digit numbers (base r), first CountingSort by least-sig-digit, then by next-least-sig-digit, etc.
- RadixSort running time: $O\left(\left(\lfloor \log_r(M) \rfloor + 1 \right) \cdot (n + r) \right)$
  - Choosing the base r to be equal to n is a good choice.
- RadixSort space: r buckets.

n: number of items
M: max size of item (if they are all integers)
r = base

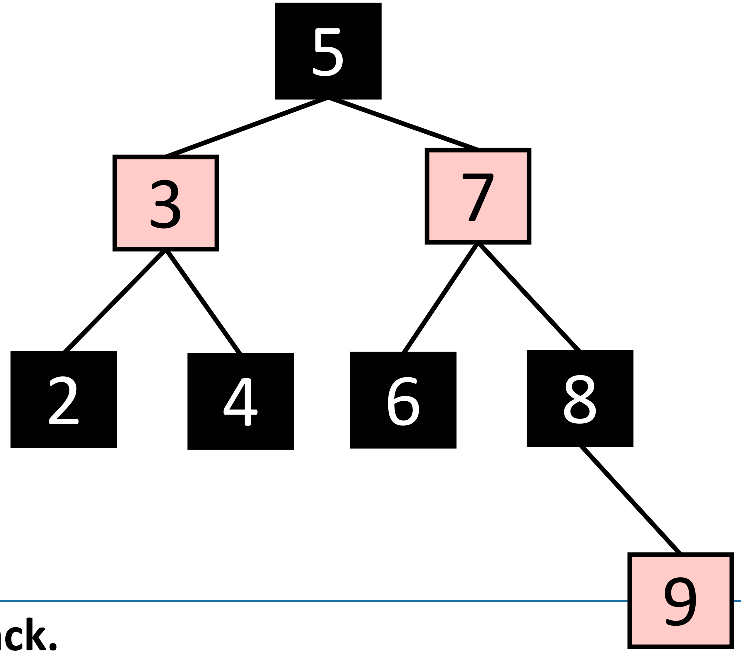
Lecture 7: Binary Search Trees

- A BST is a binary tree so that:
  - Every LEFT descendant of a node has key less than that node.
  - Every RIGHT descendant of a node has key larger than that node.
- Things you can do with a BST:
  - INSERT/SEARCH/DELETE in time **O([height of tree])**
    - In the worst-case this might be O(n) ☹
  - In-order traversal (print out in sorted order, time O(n))



Lecture 7: Red-Black Trees

- Self-balancing binary search trees.
- INSERT/SEARCH/DELETE in time O(log n) ☺
- Main idea: RBTree properties are a proxy for balance.
  - You should know what these properties are, but not the details of how to implement INSERT/SEARCH/DELETE.



- Every node is colored **red** or **black.**
- The root node is a **black node**.
- NIL children count as **black nodes**.
- Children of a **red node** are **black nodes**.
- For all nodes x:
  - all paths from x to NIL's have the same number of **black nodes** on them.

Lecture 8: Hashing

- We did that already!

So here we are!

- Who has questions?
  - Let’s prioritize conceptual questions now – for questions about specific HW problems, practice exam questions, etc, ask on Ed or in OH!

Next time!

- Graphs!

Before Next time!

- Pre-lecture exercise!