Midterm Review!

Nikil Selvam

Logistics

- When?
  - Wednesday, February 11th. From 6pm to 9pm
- Where?
  - 420-040: Last names A-M (Inclusive)
  - 320-105: Last names N-Z (Inclusive)
- What's on the test?
  - Lectures 1-7. Lecture 8 and beyond is not tested.
  - EthiCS content is fair-game.

Go read Ziyi's detailed Ed Post about the midterm!

Asymptotics

Big-Oh Notation

- Let $T(n), g(n)$ be functions of positive integers.
  - Think of $T(n)$ as a runtime: positive and increasing in n.

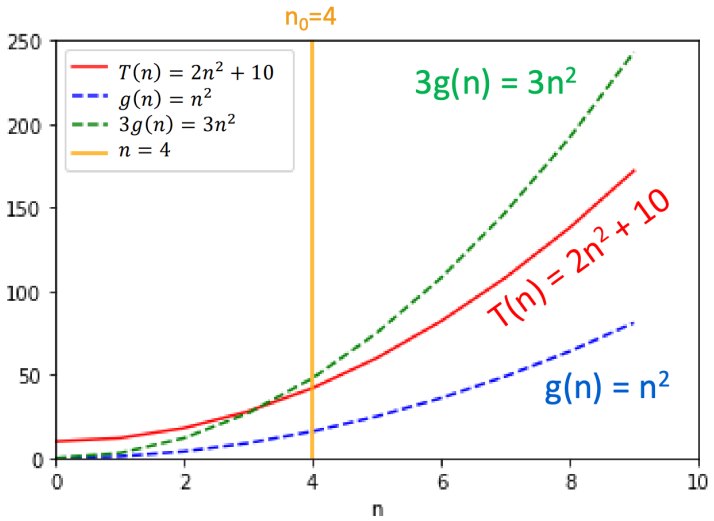
- Formally,

$$T(n) = O\big(g(n)\big)$$
$$\Leftrightarrow$$
$$\exists c > 0, n_0 \;\; s.t. \;\; \forall n \geq n_0,$$
$$T(n) \leq c \cdot g(n)$$

- Also know definitions of $\Omega(\cdots), \Theta(\cdots)$.

Example

- Show that $2n^2 + 10 = O(n^2)$



c=3 and n_0=4 works!

Question

- Show that $n^3 + n^2 + n + 2026 = O(n^3)$

c=4 and n_0=2026 works!

Takeaways | Asymptotics

- O(…): “Upper Bound”
- Ω(⋯): “Lower Bound”
- Θ(⋯): “Both”

- To formally show T(n) is O(g(n)), you need to explicitly find constants c and n_0 that satisfy the definition.

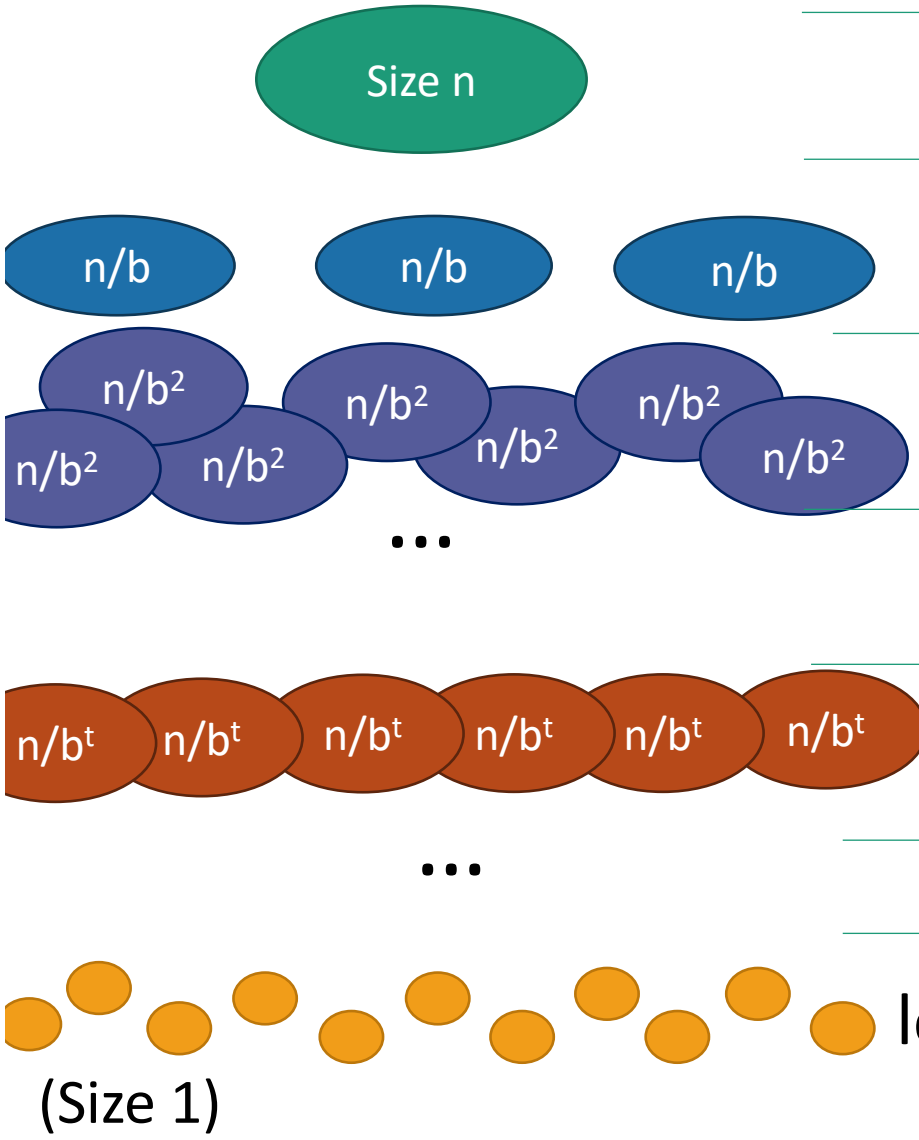
Recurrences

Question

- What do these summations evaluate to?
  - n + n + n+ … (log n terms)
  - n + n/2 + n/4 + n/8 + … (log n terms)
  - n + n + n+ … (n terms)
  - n + n/2 + n/3 + n/4 + … (n terms)

Recursion tree

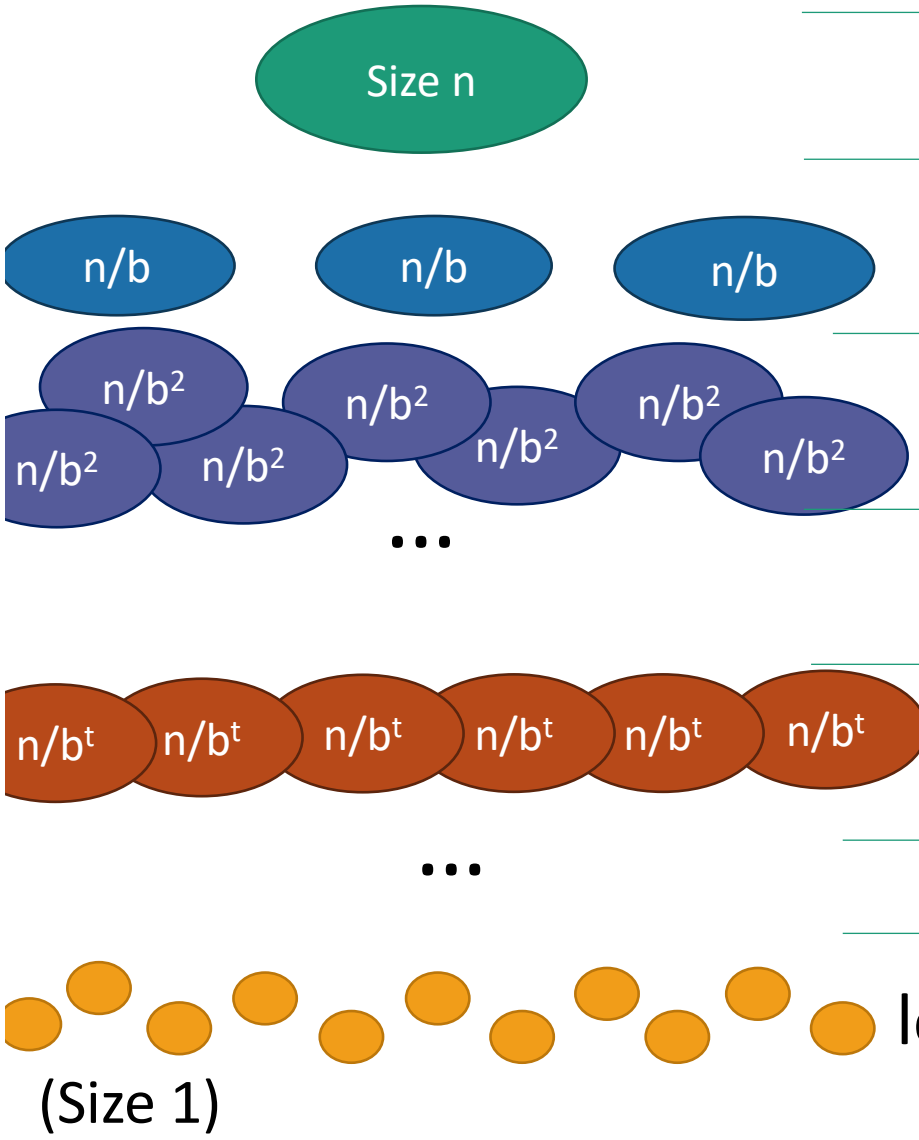
$$T(n) = a \cdot T\left(\frac{n}{b}\right) + c \cdot n^d$$



Level	# problems	Size of each problem	Amount of work at this level
0	1	n	
1	a	n/b	
2	a^2	n/b^2	
t	a^t	n/b^t	
...			
$\log_b(n)$	$a^{\log_b(n)}$	1	

Recursion tree

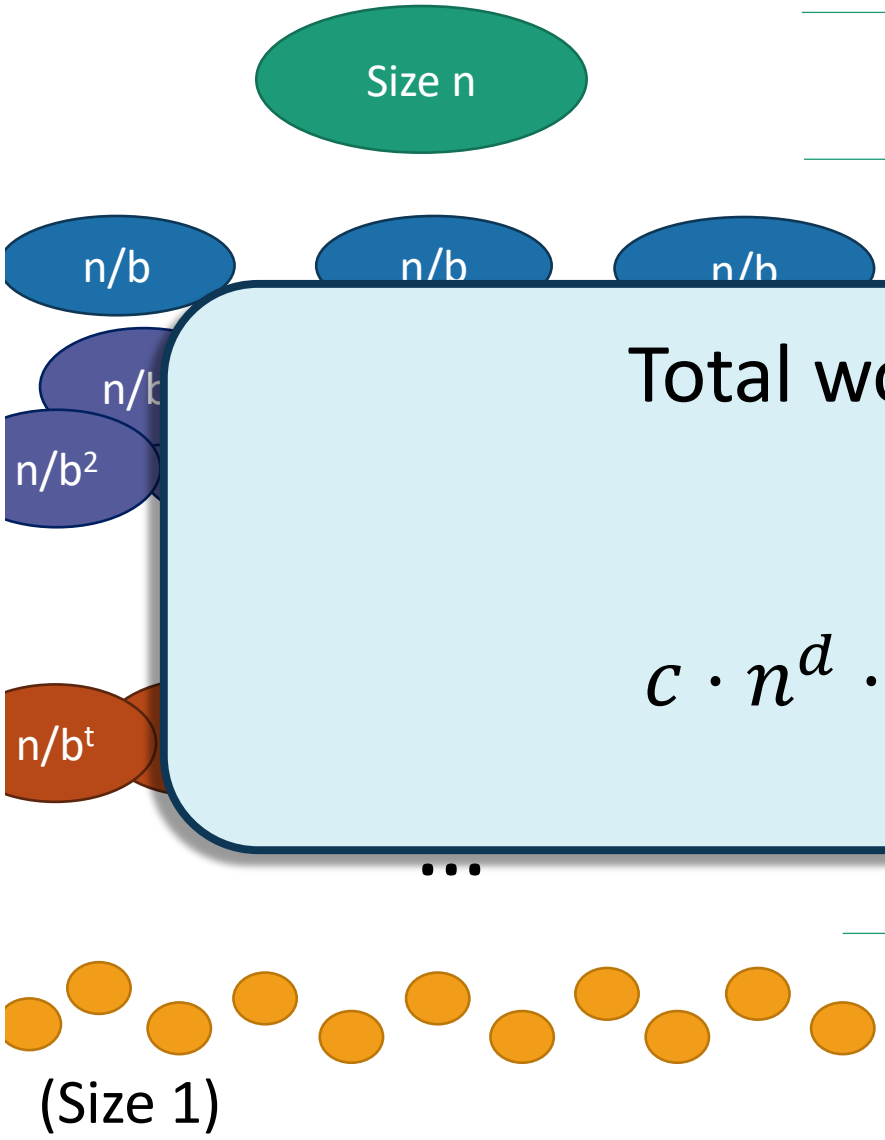
$$T(n) = a \cdot T\left(\frac{n}{b}\right) + c \cdot n^d$$



Level	# problems	Size of each problem	Amount of work at this level
0	1	n	$c \cdot n^d$
1	a	n/b	$ac\left(\frac{n}{b}\right)^d$
2	a^2	n/b^2	$a^2c\left(\frac{n}{b^2}\right)^d$
t	a^t	n/b^t	$a^tc\left(\frac{n}{b^t}\right)^d$
...			
$\log_b(n)$	$a^{\log_b(n)}$	1	$a^{\log_b(n)}c$

Recursion tree

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



Level	# problems	Size of each problem	Amount of work at this level
0	1	n	$c \cdot n^d$
1	a	n/b	$ac\left(\frac{n}{b}\right)^d$
			$\left(\frac{n}{b}\right)^d$
			$\left(\frac{n}{b}\right)^d$
...	...		
$\log_b(n)$	$a^{\log_b(n)}$	1	$a^{\log_b(n)}c$

Total work is at most:

$$c \cdot n^d \cdot \sum_{t=0}^{\log_b(n)} \left(\frac{a}{b^d}\right)^t$$

The master theorem

We can also take n/b to mean either $\lfloor \frac{n}{b} \rfloor$ or $\lceil \frac{n}{b} \rceil$ 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

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

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.

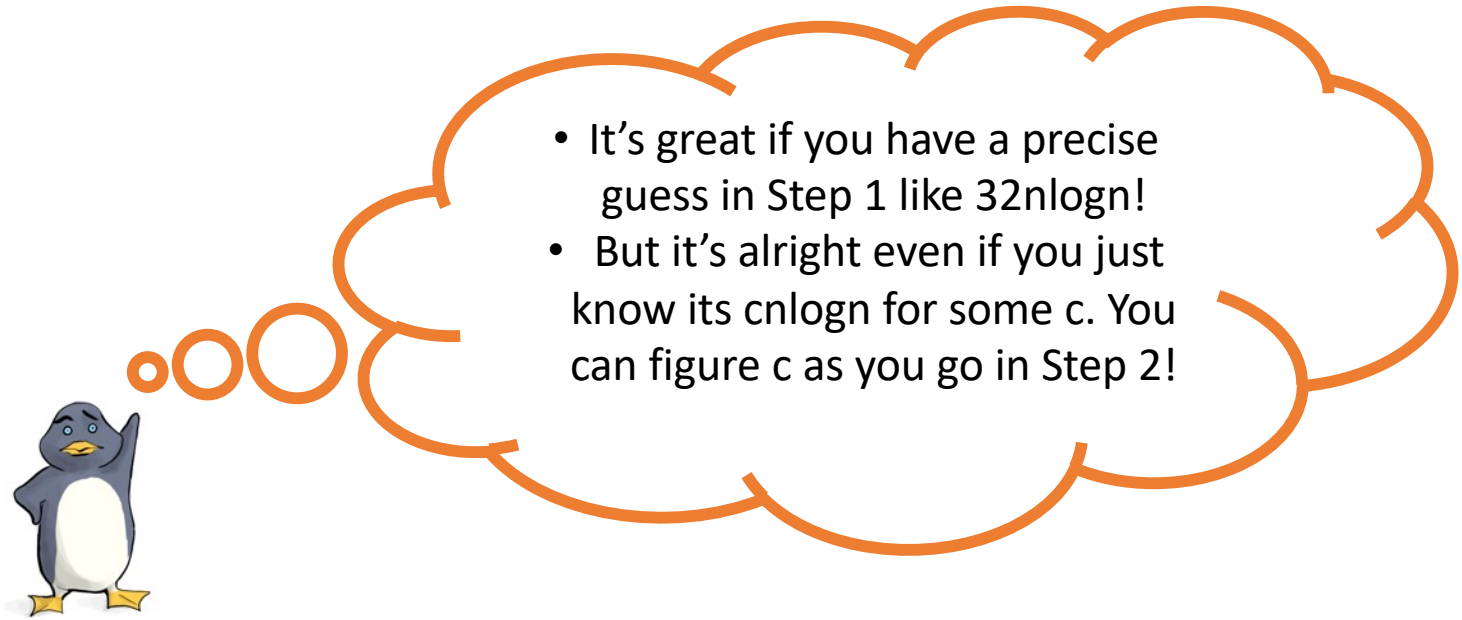
Question

- For the purposes of this class, when does Master Theorem not apply?

  - Subproblems don't have equal size
  - Work done "to combine" is not of the form n^d

The Substitution Method

- Step 1: Guess what the answer is.
- Step 2: Prove by induction that your guess is correct.
- Step 3: Pretend you never did step 1!



Example

- $T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + n$ for $n > 10$.
- Base case: $T(n) = 1$ when $1 \leq n \leq 10$

Step 1: guess the answer

$$T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + n \text{ for } n > 10.$$

Base case: $T(n) = 1$ when $1 \leq n \leq 10$

- Let's guess $O(n)$ and try to prove it.

Step 2: prove our guess is right

$T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + n$ for $n > 10$.
Base case: $T(n) = 1$ when $1 \leq n \leq 10$

- Inductive Hypothesis: $T(n) \leq \boldsymbol{C}n$
- Base case: $1 = T(n) \leq \boldsymbol{C}n$ for all $1 \leq n \leq 10$
- Inductive step:
  - Let k > 10. Assume that the IH holds for all n so that $1 \leq n < k$.
  - $T(k) \leq k + T\left(\frac{k}{5}\right) + T\left(\frac{7k}{10}\right)$
    $\leq k + \boldsymbol{C} \cdot \left(\frac{k}{5}\right) + \boldsymbol{C} \cdot \left(\frac{7k}{10}\right)$
    $= k + \frac{\boldsymbol{C}}{5}k + \frac{7\boldsymbol{C}}{10}k$
    $\leq \boldsymbol{C}k$??
  - (aka, want to show that IH holds for n=k).
- Conclusion:
  - There is some $\boldsymbol{C}$ so that for all $n \geq 1$, $T(n) \leq \boldsymbol{C}n$
  - By the definition of big-Oh, T(n) = O(n).

We don't know what C should be yet! Let's go through the proof leaving it as "C" and then figure out what works…

Whatever we choose C to be, it should have C≥1

Let's solve for C and make this true! C = 10 works.

Step 3: Pretend you never did Step 1

Theorem: $T(n) = O(n)$

$T(n) \leq n + T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right)$ for $n > 10$.

Base case: $T(n) = 1$ when $1 \leq n \leq 10$

Proof:

- Inductive Hypothesis: $T(n) \leq \mathbf{10}n$.
- Base case: $1 = T(n) \leq \mathbf{10}n$ for all $1 \leq \mathrm{n} \leq 10$
- Inductive step:
  - Let k > 10. Assume that the IH holds for all n so that $1 \leq n < k$.
  - $T(k) \leq k + T\left(\frac{k}{5}\right) + T\left(\frac{7k}{10}\right)$
    $\leq k + \mathbf{10} \cdot \left(\frac{k}{5}\right) + \mathbf{10} \cdot \left(\frac{7k}{10}\right)$
    $= k + 2k + 7k = \mathbf{10}k$
  - Thus, IH holds for n=k.
- Conclusion:
  - For all $n \geq 1, T(n) \leq \mathbf{10}n$
  - Then, T(n) = O(n), using the definition of big-Oh with $n_0 = 1, c = 10$.

Takeaways | Recurrences

- Suppose $T(n) = a \cdot T\left(\frac{n}{b}\right) + O(n^d)$ for your algorithm. Then
  - Option1 : Find the total work done by the algorithm by (optionally drawing out the recursion tree, and) summing up the work done by your algorithm at each level.
  - Option 2: Directly use Master Theorem.
- If your algorithm obeys a different recurrence
  - Option 1: You can still explicitly write a summation for total work done and evaluate it.
  - Option 2: Guess and prove by induction.

Select Algorithm

Pseudocode

- **`Select`**(A,k):
  - **If** len(A) <= 50:
    - A = **`MergeSort`**(A)
    - **Return** A[k-1]
  - p = **`choosePivot`**(A)
  - L, pivotVal, R = **`Partition`**(A,p)
  - **if** len(L) == k-1:
    - return pivotVal
  - **Else if** len(L) > k-1:
    - return **`Select`**(L, k)
  - **Else if** len(L) < k-1:
    - return **`Select`**(R, k – len(L) – 1)

Base Case: If len(A) = O(1), then any sorting algorithm runs in time O(1).

Case 1: We got lucky and found exactly the k'th smallest value!

Case 2: The k'th smallest value is in the first part of the list

Case 3: The k'th smallest value is in the second part of the list

Pseudocode

- **`Select`**(A,k):
  - **If** len(A) <= 50:
    - A = **`MergeSort`**(A)
    - **Return** A[k-1]
  - p = **`choosePivot`**(A)
  - L, pivotVal, R = **`Partition`**(A,p)
  - **if** len(L) == k-1:
    - return pivotVal
  - **Else if** len(L) > k-1:
    - return **`Select`**(L, k)
  - **Else if** len(L) < k-1:
    - return **`Select`**(R, k – len(L) – 1)

- **choosePivot**(A):
  - Split A into m = $\left\lceil \frac{n}{5} \right\rceil$ groups, of size <=5 each.
  - **For** i=1, .., m:
    - Find the median within the i'th group, call it p_i
  - p = SELECT([$p_1, p_2, p_3, \ldots, p_m$] , m/2)
  - **return** the index of p in A

Running time

- Turns out the choice of pivot guarantees that
  - $|L| \leq \frac{7n}{10} + 5$ and $|R| \leq \frac{7n}{10} + 5$
  - So, you are guaranteed to recurse into a subproblem that is at most ~70% of the original size!

- Recurrence relation:

$$T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + O(n)$$

The call to choosePivot makes one further recursive call to SELECT on an array of size n/5.

Outside of choosePivot, there's at most one recursive call to SELECT on array of size 7n/10

Takeaways | Select Algorithm

- SELECT(A,k) can be solved in linear time!
  - Pick a pivot
  - Rearrange elements around the pivot
  - Recurse left or right of pivot based on the value of k
- Choice of pivot matters!
  - Bad pivot can make this algorithm $O(n^2)$
  - Picking median as pivot is great, but we don't know how to do this.
    - Turns out we can get find a pivot using median-of-medians that is guaranteed to be "close" to the true median, and this is good enough to get a linear time solution!

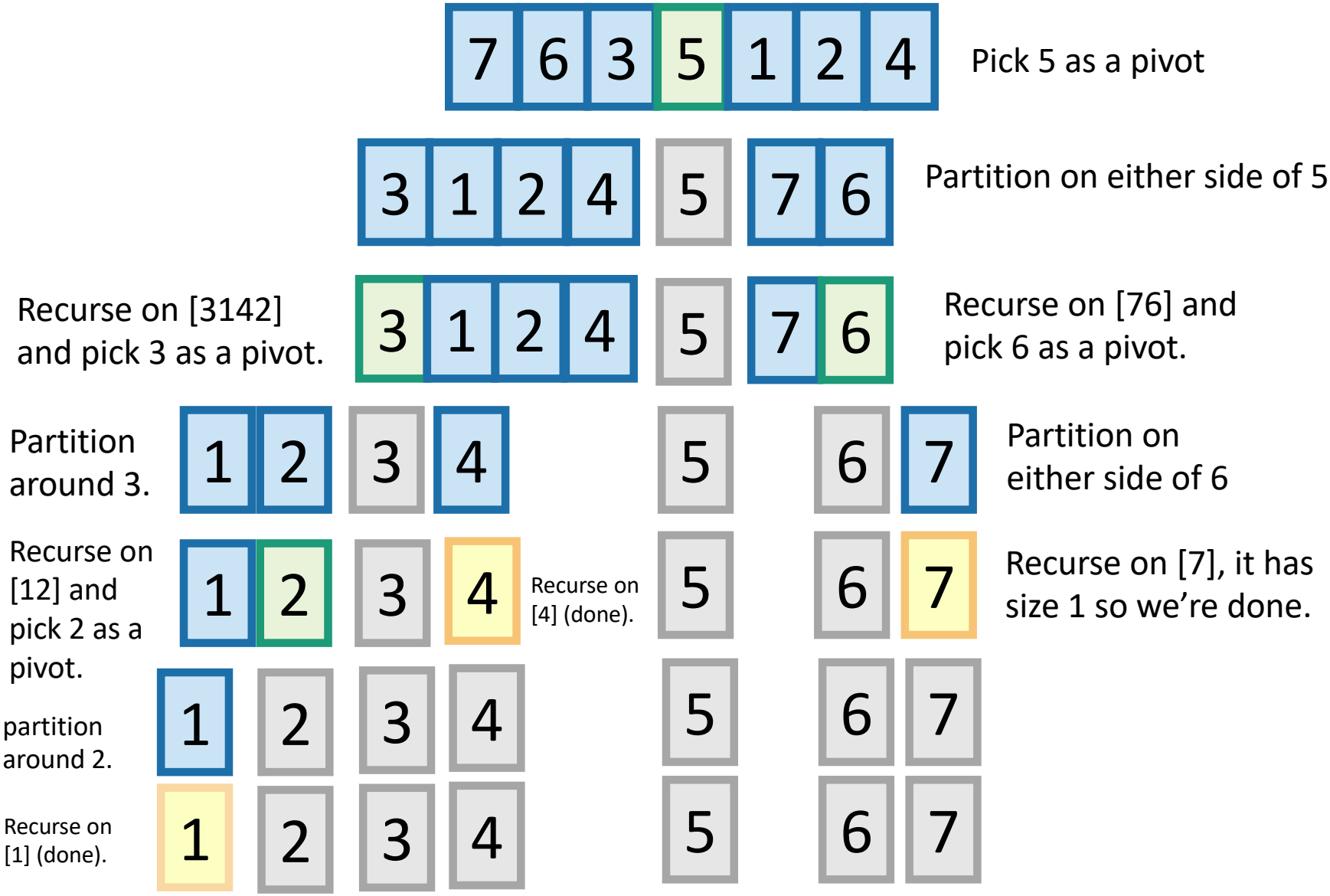
5 min Break!

QuickSort

QuickSort

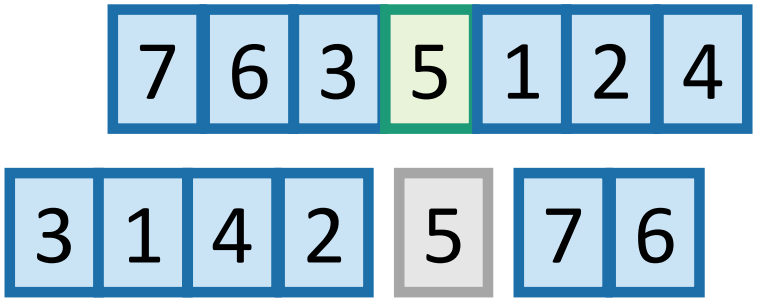
- QuickSort(A):
  - **If** len(A) <= 1:
    - **return**
  - Pick some x = A[i] at random. Call this the **pivot**.
  - PARTITION the rest of A into:
    - L (less than x) and
    - R (greater than x)
  - Replace A with [L, x, R] (that is, rearrange A in this order)
  - QuickSort(L)
  - QuickSort(R)

Example of recursive calls

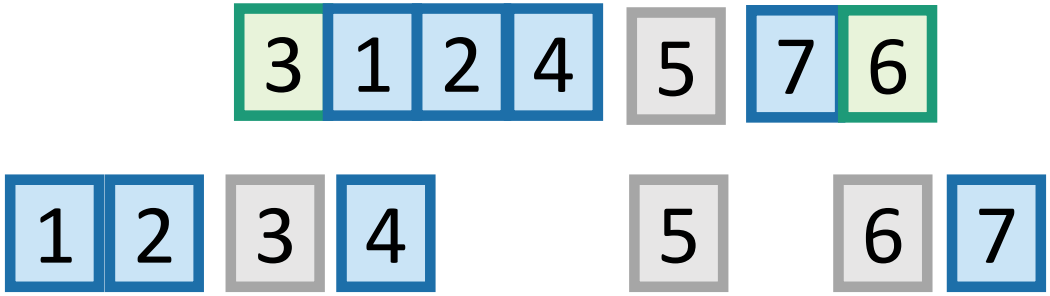


How long does this take to run?

- We will count the number of comparisons that the algorithm does.
- How many times are any two items compared?

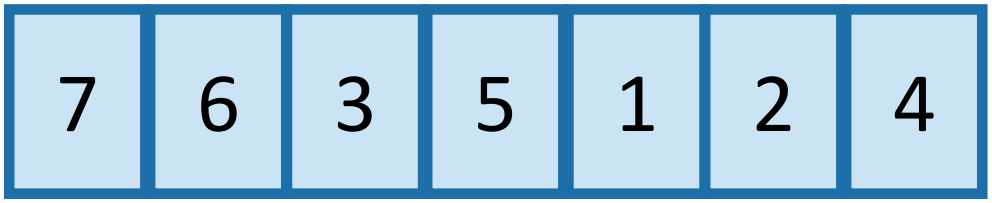


In the example before, everything was compared to 5 once in the first step….and never again.



But not everything was compared to 3.
5 was, and so were 1,2 and 4.
But not 6 or 7.

Each pair of items is compared either 0 or 1 times. Which is it?



Let's assume that the numbers in the array are actually the numbers 1,…,n

- **Whether or not a, b are compared** is a random variable, that depends on the choice of pivots. Let's say

$$X_{a,b} = \begin{cases} 1 & \text{if } a \text{ and } b \text{ are ever compared} \\ 0 & \text{if } a \text{ and } b \text{ are never compared} \end{cases}$$

- In the previous example $X_{1,5} = 1$, because item 1 and item 5 were compared.
- But $X_{3,6} = 0$, because item 3 and item 6 were NOT compared.

Counting comparisons

- The number of comparisons total during the algorithm is

$$\sum_{a=1}^{n-1} \sum_{b=a+1}^{n} X_{a,b}$$

- The expected number of comparisons is

$$E\left[\sum_{a=1}^{n-1} \sum_{b=a+1}^{n} X_{a,b}\right] = \sum_{a=1}^{n-1} \sum_{b=a+1}^{n} E[X_{a,b}]$$

by using linearity of expectations.

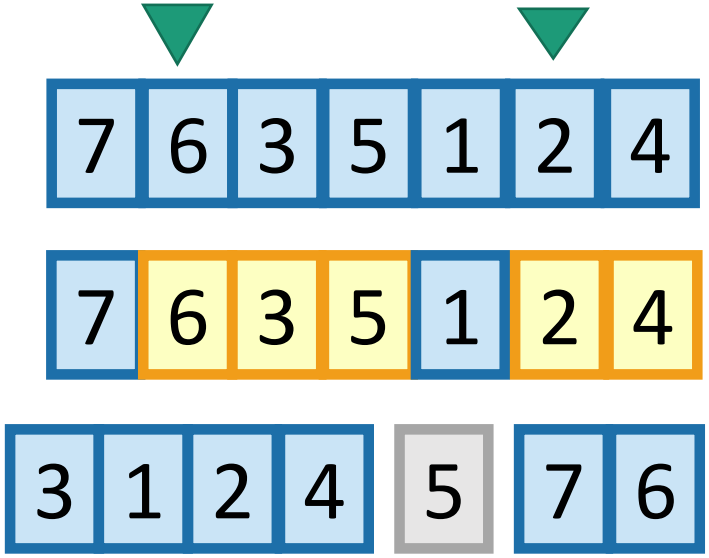
Counting comparisons

expected number of comparisons:

$$\sum_{a=1}^{n-1} \sum_{b=a+1}^{n} E[X_{a,b}]$$

- So we just need to figure out $E[X_{a,b}]$
- $E[X_{a,b}] = P(X_{a,b} = 1) \cdot 1 + P(X_{a,b} = 0) \cdot 0 = P(X_{a,b} = 1)$
  (by the definition of expectation)
- So we need to figure out:

$P(X_{a,b} = 1)$ = the probability that a and b are ever compared.



Say that a = 2 and b = 6. What is the probability that 2 and 6 are ever compared?

This is exactly the probability that either 2 or 6 is first picked to be a pivot out of the highlighted entries.

If, say, 5 were picked first, then 2 and 6 would be separated and never see each other again.

Counting comparisons

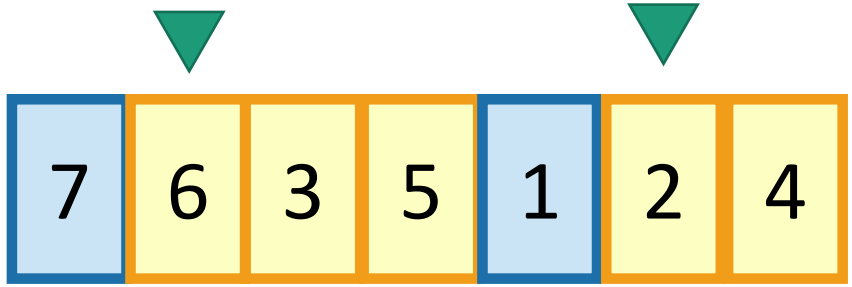
$P(X_{a,b} = 1)$

= probability a,b are ever compared

= probability that one of a,b are picked first out of all of the b – a +1 numbers between them.

2 choices out of b-a+1…

$= \frac{2}{b - a + 1}$



Expected number of comparisons

- $E\left[\sum_{a=1}^{n-1}\sum_{b=a+1}^{n} X_{a,b}\right]$ This is the expected number of comparisons throughout the algorithm
- $= \sum_{a=1}^{n-1}\sum_{b=a+1}^{n} E[\,X_{a,b}]$ linearity of expectation
- $= \sum_{a=1}^{n-1}\sum_{b=a+1}^{n} P(\,X_{a,b} = 1)$ definition of expectation
- $= \sum_{a=1}^{n-1}\sum_{b=a+1}^{n} \frac{2}{b-a+1}$ the reasoning we just did

- We get that this is less than 2n ln(n.)

Question

- Why not just pick the median as the pivot?
  - We know how to find median in linear time.
  - Are we then not guaranteed that QuickSort is nlogn?

- You could, but it's no longer a randomized algorithm!
- Also, finding the median is slow in practice due to big constant factors.

Takeaways | QuickSort

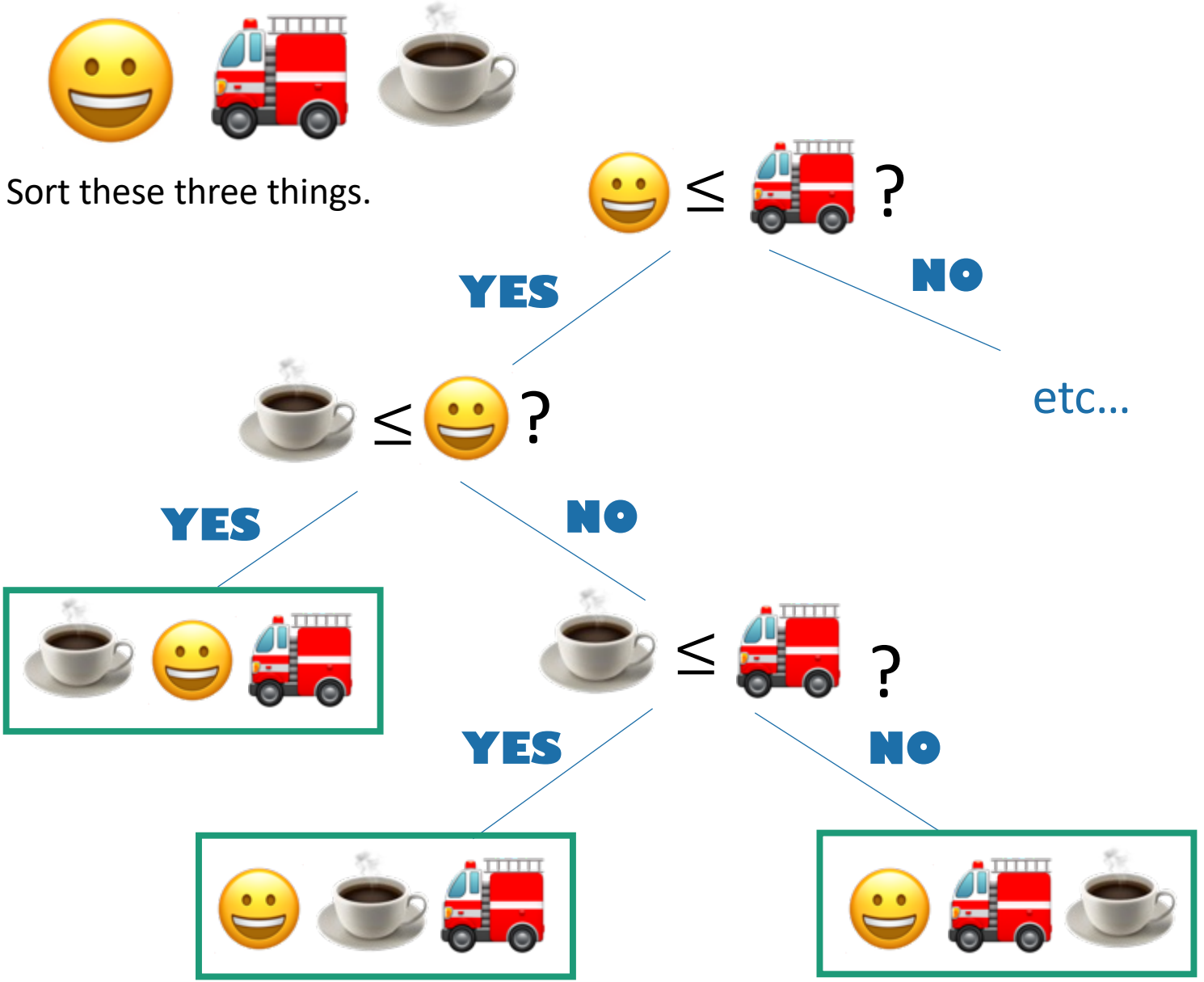
- Randomized algorithm to sort fast!
- Expected runtime is O(nlogn)
- Worst case runtime is $O(n^2)$

More Sorting

Comparison-based sorting algorithms

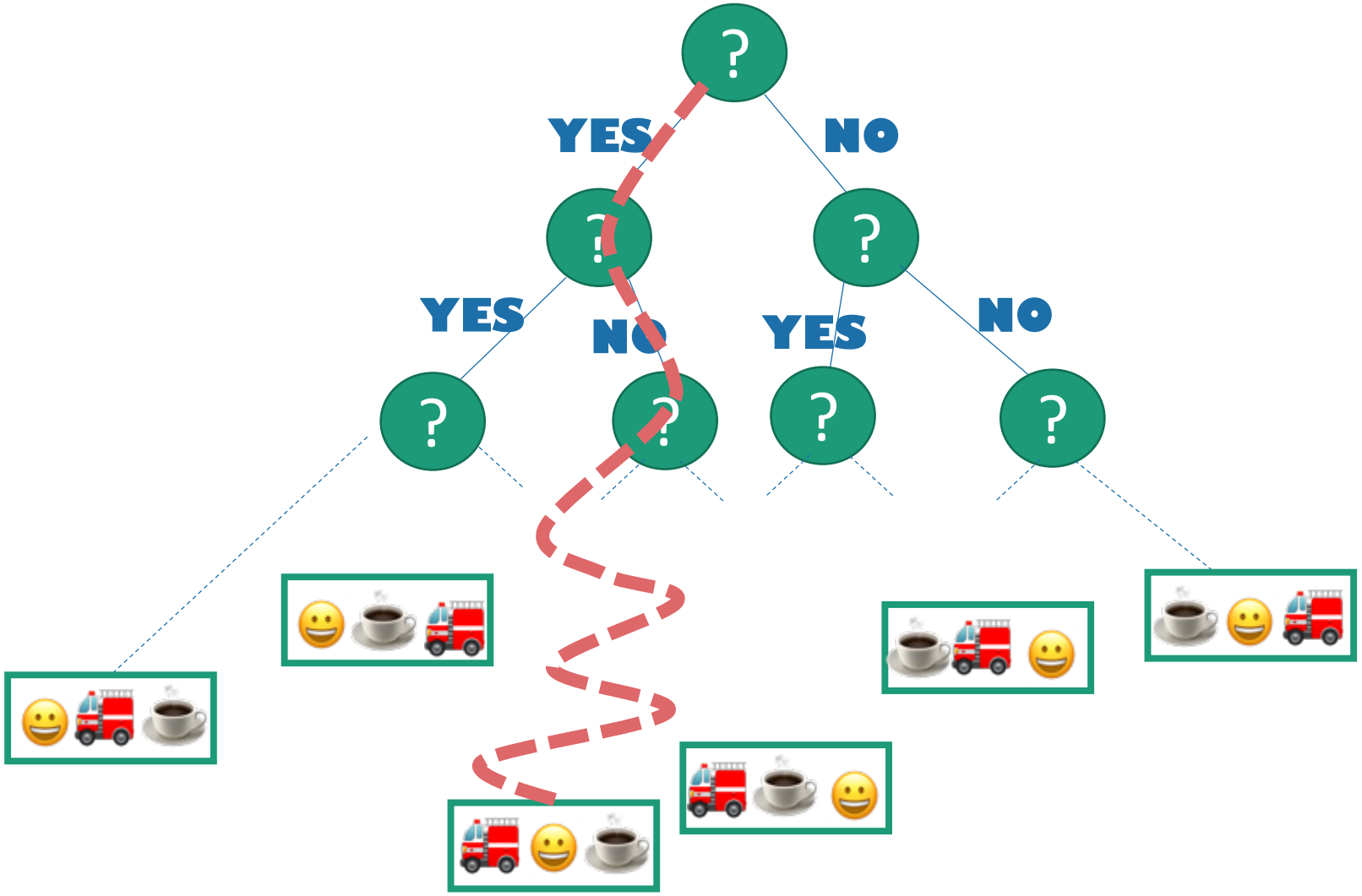
- You want to sort an array of items.
- You can’t access the items’ values directly: you can only compare two items and find out which is bigger or smaller.

They look like decision trees!



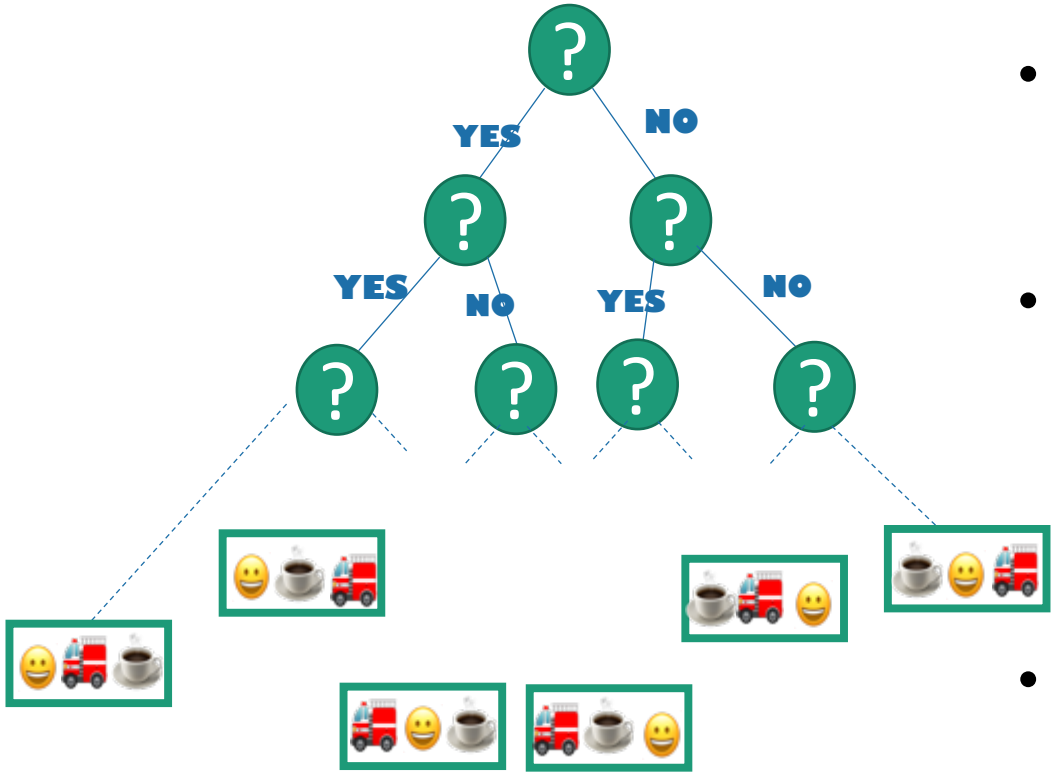
What's the worst-case runtime?

- At least Ω(length of the longest path)!



How long is the longest path?

We want a statement: in all such trees, the longest path is at least _____



- This is a binary tree with at least _____ leaves.
- The shallowest tree with n! leaves is the completely balanced one, which has depth _____.
- So in all such trees, the longest path is at least log(n!).

n!

log(n!)

Conclusion: the longest path has length at least Ω(n log(n)).

Question

- Show that $\log(n!) = \Theta(n \log n)$

$\log(n!) = \log(n \times n\text{-}1 \times n\text{-}2 \ldots 1)$

$\leq \log(n \times n \times n \ldots n)$

$= \log(n^n)$

$= n \log n$

Can do something similar for the lower bound!

Another model of computation

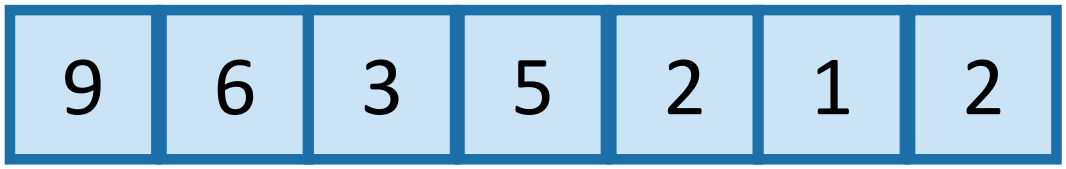
- The items you are sorting have meaningful values.

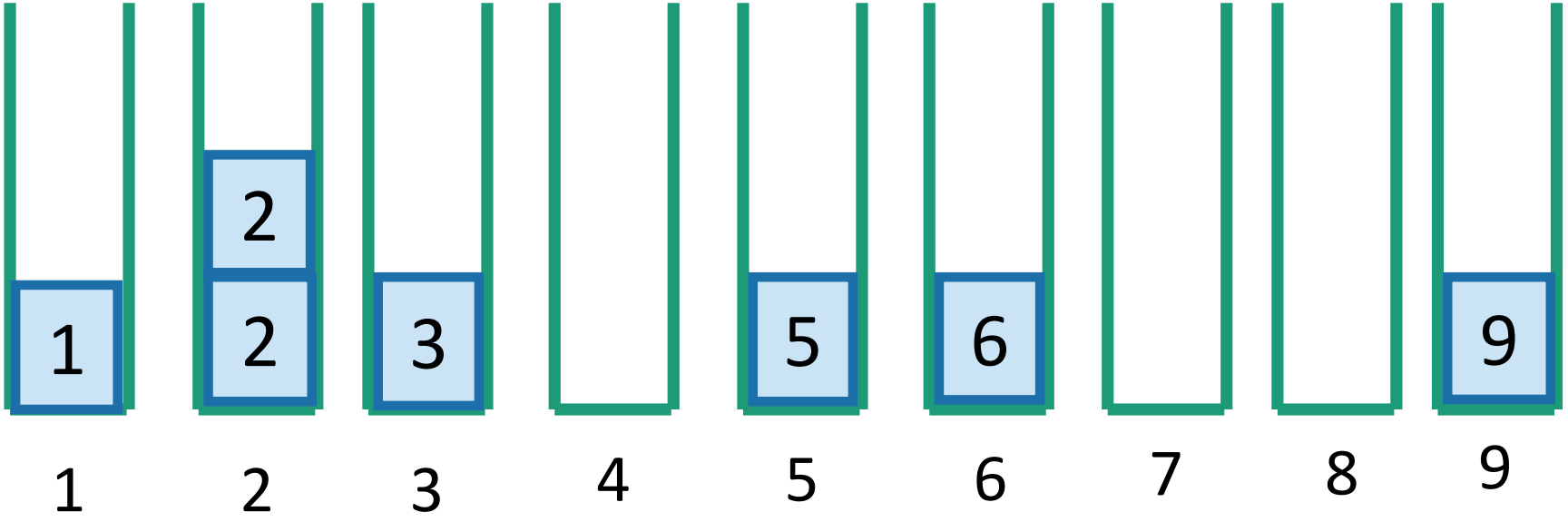
9	6	3	5	2	1	2

instead of

Why might this help?

CountingSort:





Concatenate the buckets!

SORTED!

In time O(n).

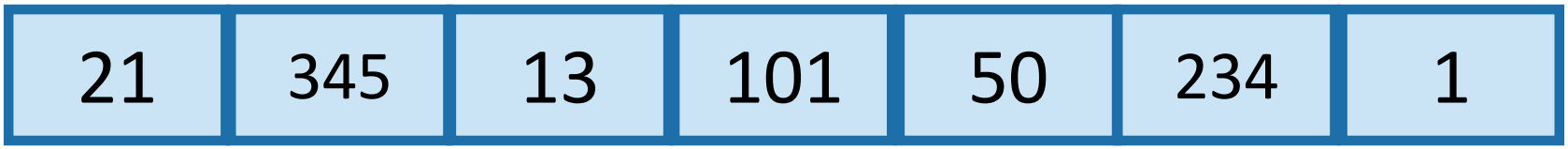
RadixSort

- For sorting integers up to size M
  - Can use less space than CountingSort

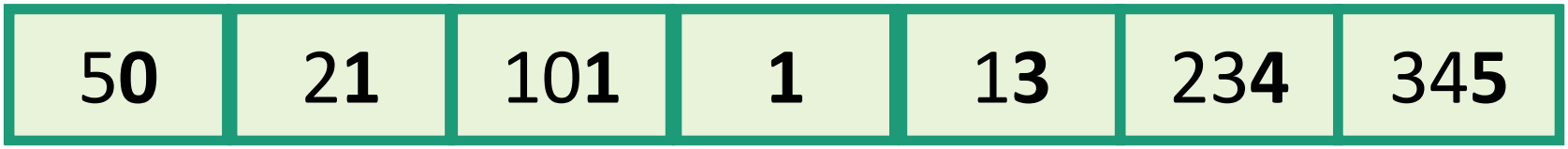
- Algorithm: CountingSort on the least-significant digit first, then the next least-significant, and so on.

Why does this work?

Original array:



Next array is sorted by the first digit.



Next array is sorted by the first two digits.

Next array is sorted by all three digits.

Sorted array

General running time of RadixSort

- Say we want to sort:
  - n integers,
  - maximum size M,
  - in base r.
- Number of iterations of RadixSort:
  - Same as number of digits, base r, of an integer x of max size M.
  - That is $\mathrm{d} = \lfloor \log_r(M) \rfloor + 1$
- Time per iteration:
  - Initialize r buckets, put n items into them
  - $O(n + r)$ total time.
- Total time:
  - $O\big(d \cdot (n + r)\big) = O\big((\lfloor \log_r(M) \rfloor + 1) \cdot (n + r)\big)$

Takeaways | More Sorting

- For comparison-based sorting algorithms, no algorithm can do better than nlogn.
- If we are sorting small integers, we could do better using Counting Sort or Radix Sort!
  - The runtime of Radix Sort is $O\big((\,\lfloor \log_r(M) \rfloor + 1\,) \cdot (n + r)\big)$

Binary Search Trees and Red Black Trees

Why do we care about these trees?

	Sorted Arrays	Linked Lists	(Balanced) **Binary Search Trees**
Search	O(log(n)) 😃	O(n) 😕	O(log(n)) 😃
Delete	O(n) 😕	O(n) 😕	O(log(n)) 😃
Insert	O(n) 😕	O(1) 😃	O(log(n)) 😃

Binary tree terminology

Each node has two children.

The left child of 3 is 2

The right child of 3 is 4

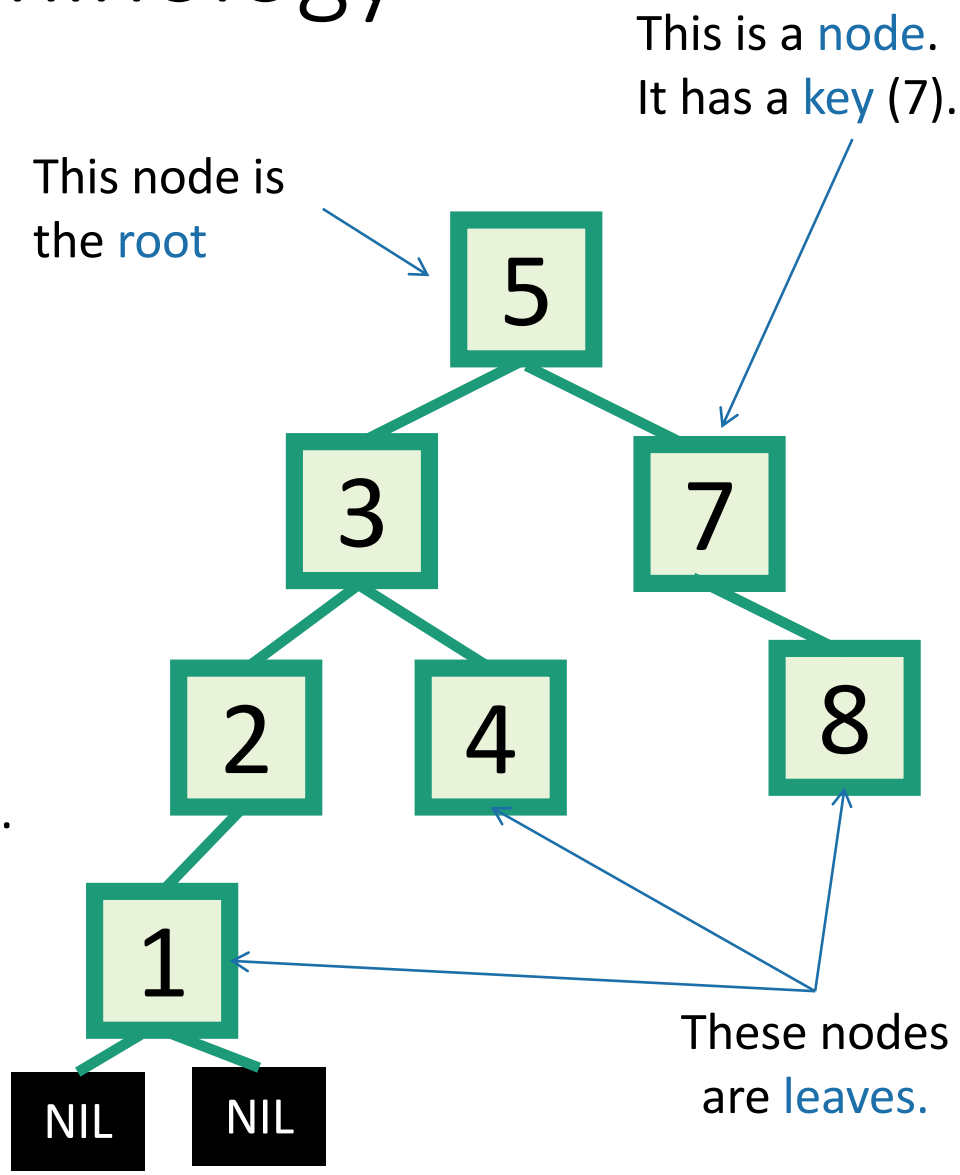
The parent of 3 is 5

2 is a descendant of 5

Each node has a pointer to its left child, right child, and parent.

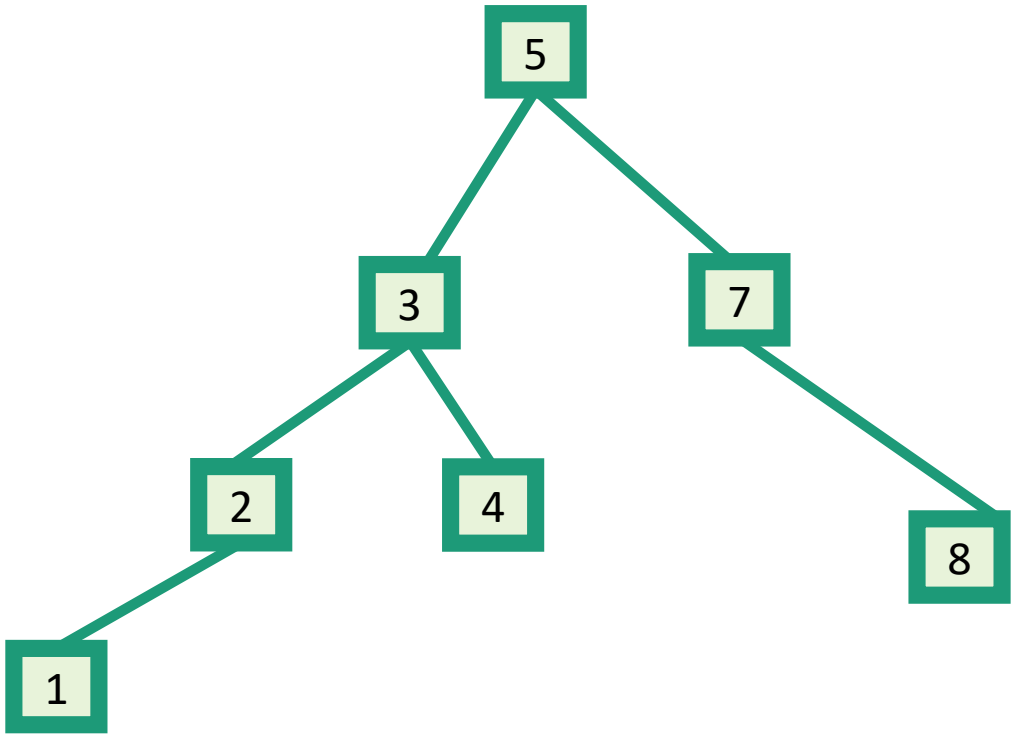
Both children of 1 are NIL.
(I won't usually draw them).

The height of this tree is 3.
(Max length of path from the root to a leaf).



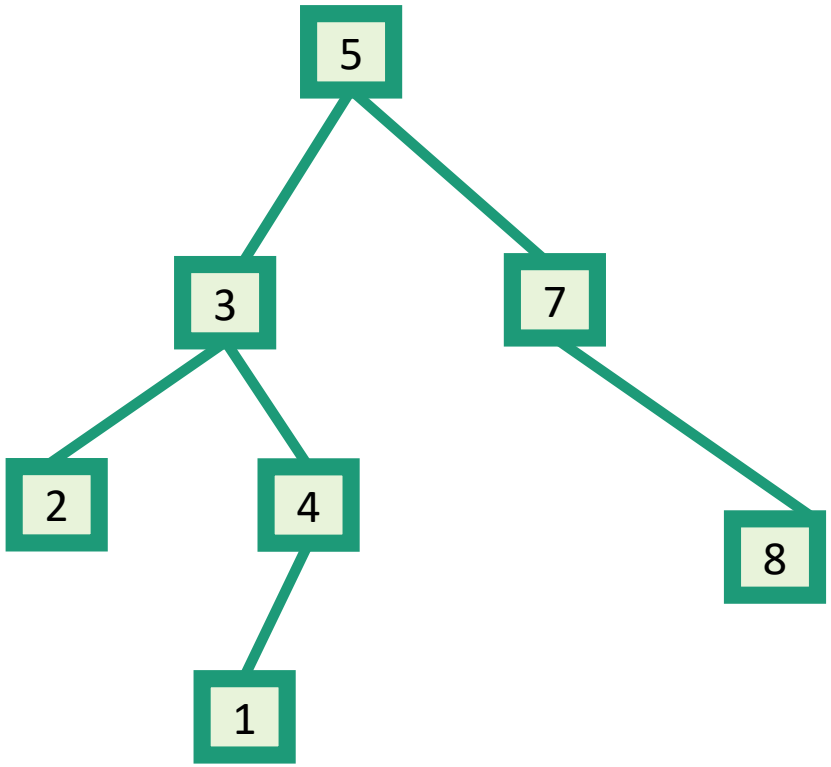
Binary Search Trees

- 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.
- Example BST:



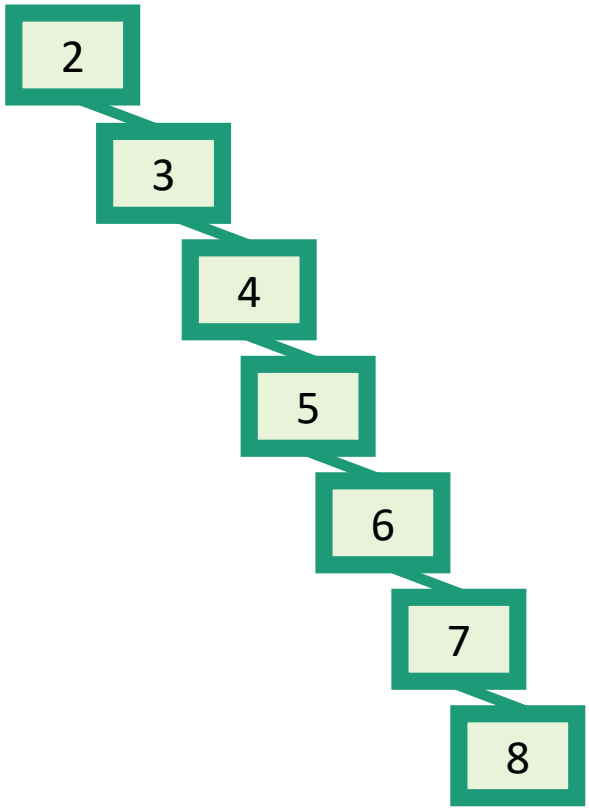
Question

- Is this a BST?



Question

- Is this a BST?



Runtime

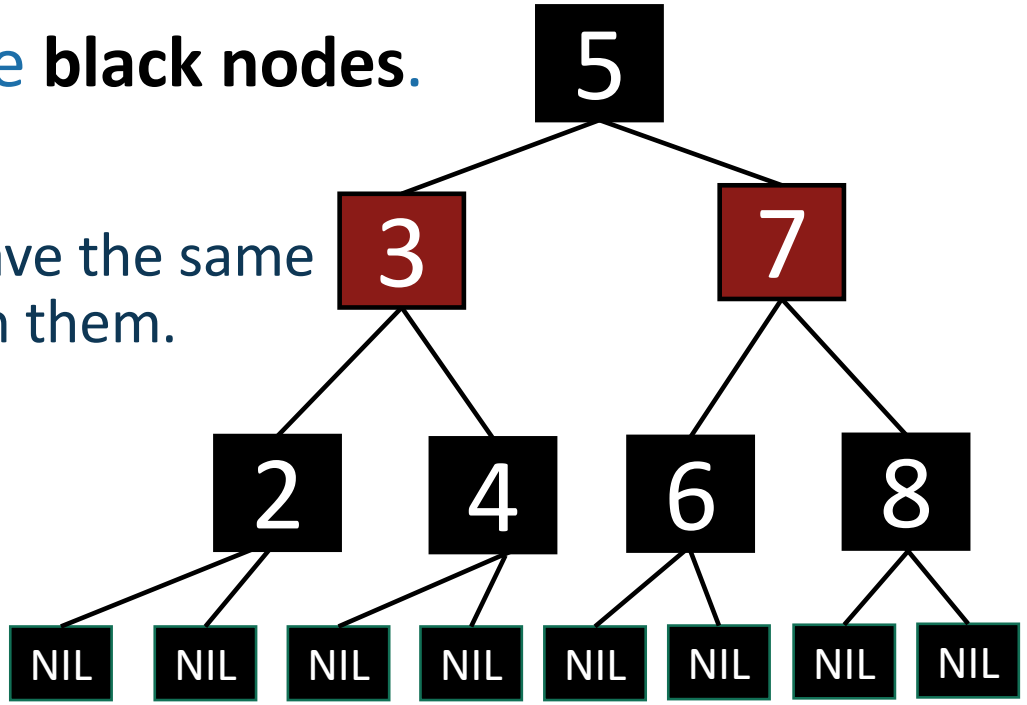
- SEARCH: O(height)
- INSERT: O(height)
- DELETE: O(height)

- But height can be n if we do not make sure the tree is balanced!
  - We want to make sure height is always roughly log(n)

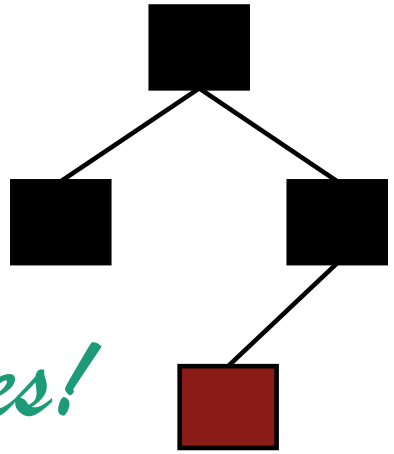
Red-Black Trees

obey the following rules (which are a proxy for balance)

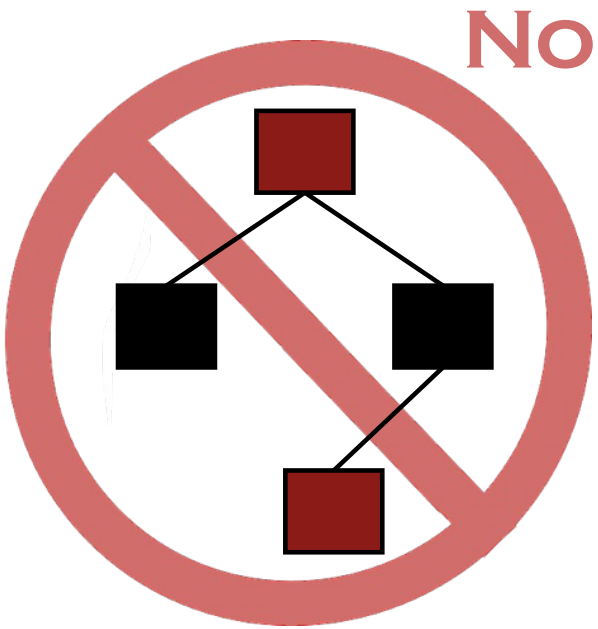
- Every node is colored **red** or **black.**
- The root node is a **black node**.
- NIL children count as **black nodes**.
- Children of a **red node** are **black nodes**.
- For all nodes x:
  - all paths from x to NIL's have the same number of **black nodes** on them.



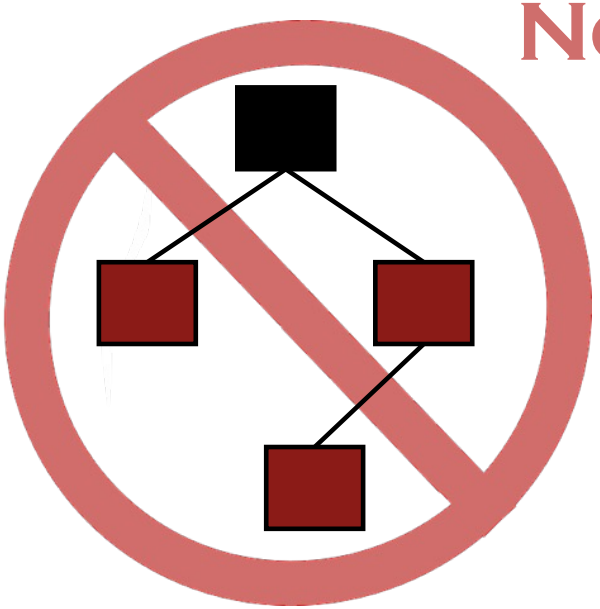
Question



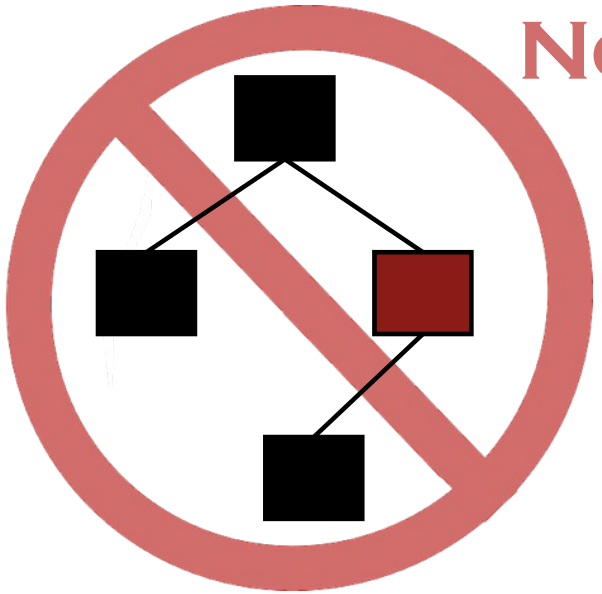
Yes!



No!



No!



No!

Takeaways | BSTs and RB Trees

- BST: Left descendants lower, Right descendants higher
  - Support operations in O(height)
- RB Tree: One particular kind of BST that is guaranteed to be balanced
  - Ensuring that all operations are O(logn)
- Must know their definitions/properties. Do not need to know how insertion/deletion etc works!

Good Luck!

You’re gonna do great!