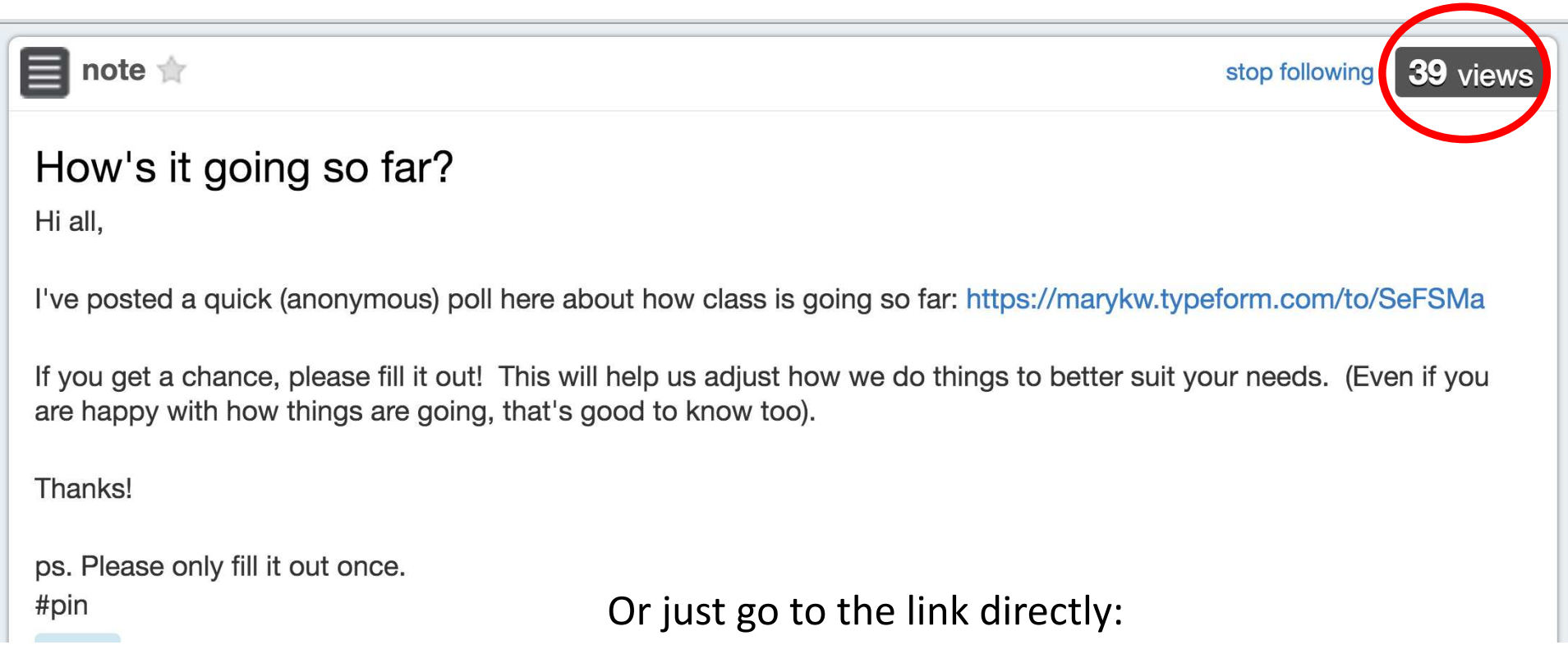
Lecture 4: Selection and Median

While you are waiting for class to start:

Go to Piazza, find this note, fill out the poll!

Not enough!



Or just go to the link directly:
https://marykw.typeform.com/to/SeFSMa

Do it now! You can even do it on your phone!

(and if you can't do it now, then please do it later!)

Announcements!

- Check out the WiCS hackathon:
  - http://web.stanford.edu/group/wics/hackoverflow/spr2017/
  - April 15
- Recitation Sections:
  - Location might change to Gates Basement.
  - (Check the Google Calendar before heading to section.)
- Please fill out the survey on Piazza.
  - I'll update my teaching style based on it, so even if you are happy with how things are, you should register that!
- HW1 due Friday.
  - (And HW2 also posted Friday).

Some final remarks about the master theorem

- 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

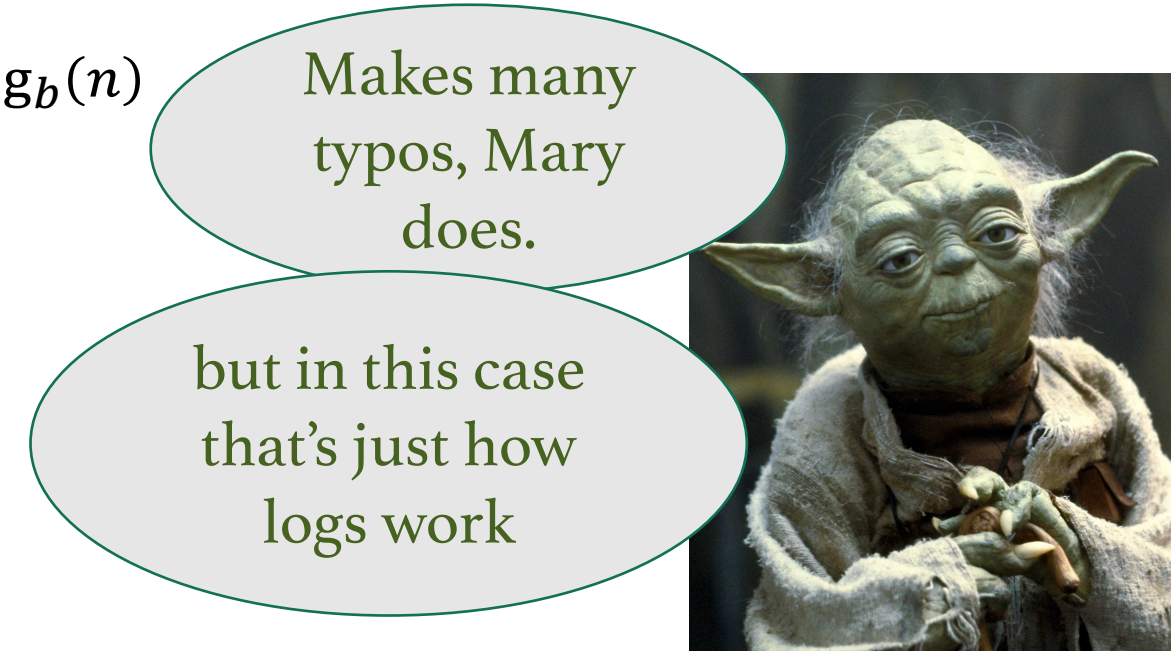
d : need to do n^d work to create all the subproblems and combine their solutions.

Jedi master Yoda

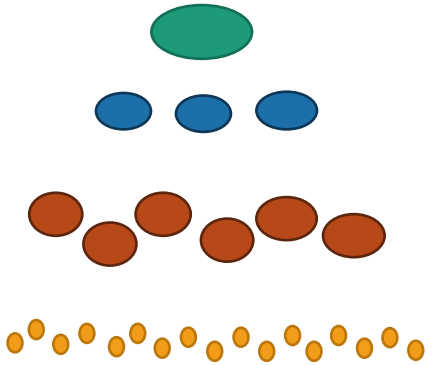
Last time…

- Ended on the board with $O\left(a^{\log_b(n)}\right)$
- The Theorem says $O\left(n^{\log_b(a)}\right)$
- Is it a typo?
- Good exercise:

$$n^{\log_b(a)} = a^{\log_b(n)}$$



Some intuition about the cases



Work at level t: $O(n^d (a/b^d)^t)$

- Case 1: $a = b^d$
  - The recursion tree has the same amount of work at every level. (Like MergeSort).
- Case 3: $a > b^d$
  - The tree branches really quickly compared to work per problem! The bulk of the work is done at the bottom of the tree. (Like Karatsuba).
- Case 2: $a < b^d$
  - The work done shrinks way faster than we branch new problems. The bulk of the work is done at the root of the tree. (We haven't seen this yet but we will today).

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

Today: more recursion, beyond the Master Theorem.

- The Master Theorem only works when all sub-problems are the same size.
- That's not always the case.
- Today we'll see an example where the Master Theorem won't work.
- We'll use something called the substitution method instead.

I can handle all the recurrence relations that look like $T(n) = a \cdot T\left(\frac{n}{b}\right) + O(n^d)$. Before this theorem I was but the learner. Now I am the master.

Only a master of evil*, Darth.

*More precisely, only a master of same-size sub-problems…still pretty handy, actually.

The problem we will solve

A is an array of size n, k is in {1,…,n}

- SELECT(A, k):
  - Return the k'th smallest element of A.

7	4	3	8	1	5	9	14

- SELECT(A, 1) = 1
- SELECT(A, 2) = 3
- SELECT(A, 3) = 4
- SELECT(A, 8) = 14

- SELECT(A, 1) = MIN(A)
- SELECT(A, n/2) = MEDIAN(A)
- SELECT(A, n) = MAX(A)

Assuming n is even! Otherwise need some ceilings or floors.

We're gonna do it in time O(n)

- Let's start with MIN(A) aka SELECT(A, 1).
- MIN(A):
  - ret = ∞
  - **For** i=1, .., n:
    - If A[i] < ret:
      - ret = A[i]
  - **Return** ret

This stuff is O(1)

This loop runs O(n) times

- Time O(n). Yay!

How about SELECT(A,2)?

- SELECT2(A):
  - ret = ∞
  - minSoFar = ∞
  - **For** i=1, .., n:
    - If A[i] < ret and A[i] < minSoFar:
      - ret = minSoFar
      - minSoFar = A[i]
    - Else if A[i] < ret and A[i] >= minSoFar:
      - ret = A[i]
  - **Return** ret

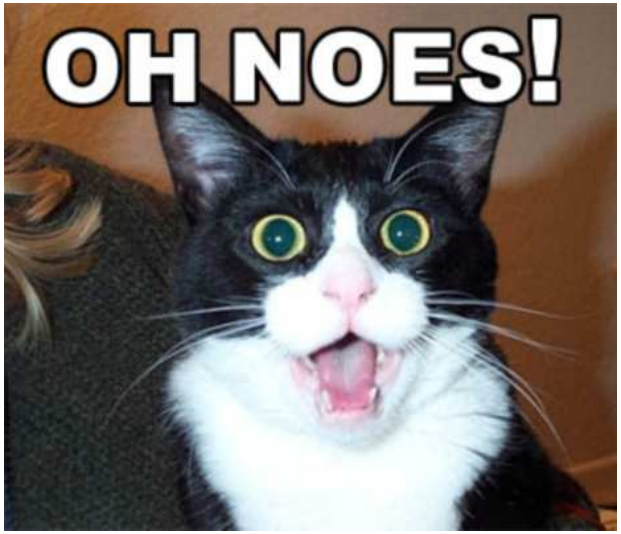
(The actual algorithm here is not very important because this won't end up being a very good idea…)

Still O(n)

SO FAR SO GOOD.

SELECT(A, n/2) aka MEDIAN(A)?

- MEDIAN(A):
  - ret = ∞
  - minSoFar = ∞
  - secondMinSoFar = ∞
  - thirdMinSoFar = ∞
  - fourthMinSoFar = ∞
  - ….



- This is not a good idea for large k (like n/2 or n).
- Basically this is just going to turn into something like INSERTIONSORT…and that was $O(n^2)$.

A much better idea for large k

- SELECT(A, k):
  - A = MergeSort(A)
  - **return** A[k]

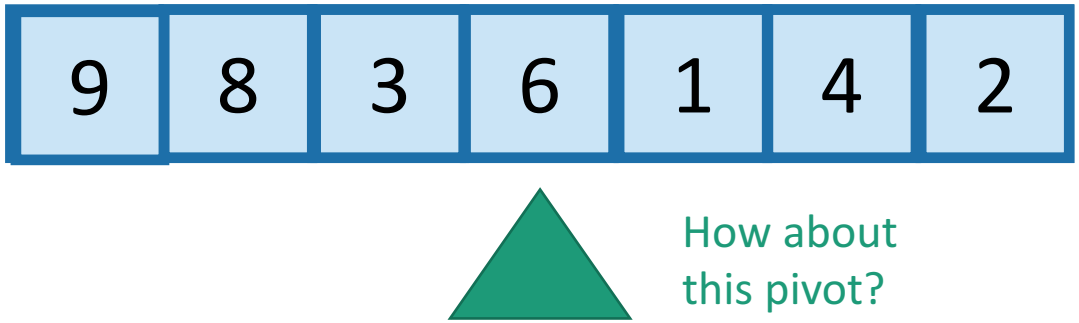
- Running time is O(n log(n)).
- So that's the benchmark….

Can we do better?

We're hoping to get O(n)

Idea: recursion

Say we want to find SELECT(A, k)



First, pick a "pivot." We'll see how to do this later.

Next, partition the array into "bigger than 6" or "less than 6"

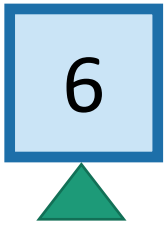
This PARTITION step takes time O(n). (Notice that we don't sort each half).

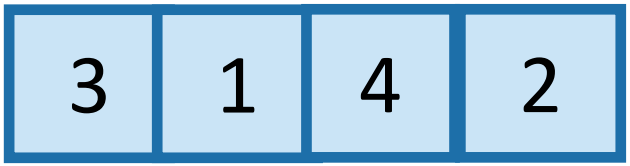
L = array with things smaller than A[pivot]

R = array with things larger than A[pivot]

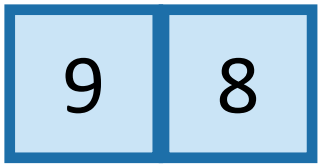
Idea continued…

Say we want to find SELECT(A, k)





L = array with things smaller than A[pivot]



R = array with things larger than A[pivot]

- If k = 5 = len(L) + 1:
  - We should return A[pivot]
- If k < 5:
  - We should return SELECT(L, k)
- If k > 5:
  - We should return SELECT(R, k – 5)

This suggests a recursive algorithm

(still need to figure out how to pick the pivot…)

Let's make that a bit more formal

- PARTITION(A, p):
  - L = new array
  - R = new array
  - **For** i=1,…,n:
    - **If** i==p:
      - continue
    - **Else if** A[i] <= A[p]:
      - L.append(A[i])
    - **Else if** A[i] > A[p]:
      - R.append(A[i])
  - **Return** L, A[p], R

initialize L and R

go through elts one at a time…put small ones in L, big ones in R.

- This is the O(n) PARTITION algorithm that we saw before.
- For clarity, I'm just going to initialize two new arrays, L and R. (Assume they are dynamically sized, and that we can append stuff in time O(1), and access any index in time O(1)).
- However, you can implement this (and everything else we will do in this lecture) in-place, without any of these considerations. (Fun exercise! Or see CLRS.)

More formal part II

(picture on board)

- SELECT(A, k):
  - **If** len(A) <= 50:
    - A = MergeSort(A)
    - Return A[k]
  - Choose p in {1,…,n}
  - L, A[p], R = PARTITION(A,p)
  - **If** len(L) = k - 1:
    - **Return** A[p]
  - **Else If** len(L) > k - 1:
    - **Return** SELECT(L, k)
  - **Else if** len(L) < k - 1:
    - return SELECT(R, k – len(L) – 1)

We'll see why I chose 50 later. It's pretty arbitrary.

- PARTITION(A, p):
  - L = new array
  - R = new array
  - **For** i=1,…,n:
    - **If** i==p:
      - continue
    - **else If** A[i] <= A[p]:
      - L.append(A[i])
    - **Else if** A[i] > A[p]:
      - R.append(A[i])
  - **Return** L, A[p], R

Correctness

(illustration on board)

There seems to be some whitespace not yet used.

That's better.

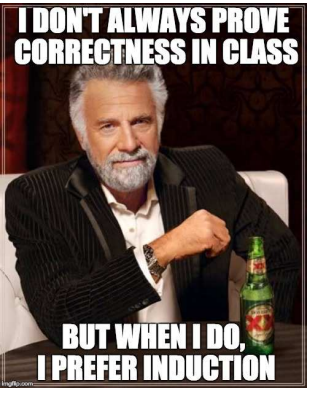
- Recursion invariant:

At the return of each recursive call of size < n, SELECT(A,k) returns the k'th smallest element of A.

- Base case ("Initialization"):
- If len(A) <= 50, then the MergeSort approach is "clearly" correct.
- Inductive step: ("Maintenance")
  - Suppose that the recursion invariant holds for n.
  - Want to show that it holds for n + 1.
  - Three cases:
    - **if len(L) = k-1**, then A[p] is the correct thing to return.
    - **If len(L) > k-1**, then the k'th smallest element of L is the correct thing to return
      - And by induction, this is indeed what we return.
    - **If len(L) < k-1**, then the (k – (len(L) - 1)'st smallest elt of R is the correct thing to return.
      - And by induction, this is indeed what we return.
- Conclusion ("Termination")
- By induction, the recursion invariant holds for n + 1, which means that SELECT(A,k) is correct.

- SELECT(A, p=k):
  - **If** len(A) <= 50:
    - A = MergeSort(A)
    - Return A[k]
  - Choose p in {1,…,n}
  - L, A[p], R = PARTITION(A,p)
  - **If** len(L) = k - 1:
    - **Return** A[p]
  - **Else If** len(L) > k - 1:
    - **Return** SELECT(L, k)
  - **Else if** len(L) < k - 1:
    - return SELECT(R, k – len(L) – 1)

Note: something like this is totally acceptable on your HW (maybe with one more sentence, or an example, saying why those are the right things to return.)



Note: Soon I'm going to stop proving correctness in class – eventually all these arguments will start to look the same.

How about runtime?

- Let's try a recurrence relation…
- $$T(n) = \begin{cases} T\big(len(L)\big) + O(n) & len(L) < k - 1 \\ T\big(len(R)\big) + O(n) & len(L) > k - 1 \\ O(n) & len(L) = k - 1 \end{cases}$$

- What is len(L), len(R)?
- Let's pretend that len(L) is about n/2.
- $T(n) \leq T\left(\frac{n}{2}\right) + O(n)$

- SELECT(A, p=k):
  - **If** len(A) <= 150:
    - A = MergeSort(A)
    - Return A[k]
  - Choose p in {1,…,n}
  - L, A[p], R = PARTITION(A,p)
  - **If** len(L) = k - 1:
    - **Return** A[p]
  - **Else If** len(L) < k - 1:
    - **Return** SELECT(L, k)
  - **Else if** len(L) > k - 1:
    - return SELECT(R, k – len(L) – 1)

Recall the Master Theorem

which totally doesn't apply here, we are cheating by pretending we know the problem size.

Note: This is a rhetorical point for intuition in lecture. It is **NOT OKAY** as a final solution on your HW.

- Suppose $T(n) = a \cdot T\left(\frac{n}{b}\right) + O(n^d)$. Then

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

In our case:

- $T(n) \leq T\left(\frac{n}{2}\right) + O(n)$
- So a = 1, b = 2, d = 1
- $T(n) \leq O(n^d) = O(n)$

Lucky the Lackadaisical Lemur

How about runtime?

- Let's try a recurrence relation…
- $$T(n) = \begin{cases} T\big(len(L)\big) + O(n) & len(L) < k - 1 \\ T\big(len(R)\big) + O(n) & len(L) > k - 1 \\ O(n) & len(L) = k - 1 \end{cases}$$

- What is len(L), len(R)?
- Let's **pretend** that len(L) is about ~~n/2~~. 7n/10
- $T(n) \leq T\left(\frac{n}{2}\right) + O(n)$
- $T(n) = O(n)$
- That would be great!

(we can even assume something a little weaker)

- SELECT(A, p=k):
  - **If** len(A) <= 150:
    - A = MergeSort(A)
    - Return A[k]
  - Choose p in {1,…,n}
  - L, A[p], R = PARTITION(A,p)
  - **If** len(L) = k - 1:
    - **Return** A[p]
  - **Else If** len(L) > k - 1:
    - **Return** SELECT(L, k)
  - **Else if** len(L) < k - 1:
    - return SELECT(R, k – len(L) – 1)

Recall the Master Theorem

which totally doesn't apply here, we are cheating by pretending we know the problem size.

- Suppose $T(n) = a \cdot T\left(\frac{n}{b}\right) + O(n^d)$. Then

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

In our case:

- $T(n) \leq T\left(\frac{7n}{10}\right) + O(n)$
- So a = 1, b = 10/7, d = 1
- $T(n) \leq O(n^d) = O(n)$

Lucky the Lackadaisical Lemur

How about runtime?

- Let's try a recurrence relation…
- $$T(n) = \begin{cases} T\big(len(L)\big) + O(n) & len(L) < k - 1 \\ T\big(len(R)\big) + O(n) & len(L) > k - 1 \\ O(n) & len(L) = k - 1 \end{cases}$$

- What is len(L), len(R)?
- Let's **pretend** that len(L) is about ~~n/2~~ 7n/10.
- $T(n) \leq T\left(\frac{7n}{10}\right) + O(n)$
- $T(n) = O(n)$
- That would be great! !!!!!

- SELECT(A, p=k):
  - **If** len(A) <= 150:
    - A = MergeSort(A)
    - Return A[k]
  - Choose p in {1,…,n}
  - L, A[p], R = PARTITION(A,p)
  - **If** len(L) = k - 1:
    - **Return** A[p]
  - **Else If** len(L) > k - 1:
    - **Return** SELECT(L, k)
  - **Else if** len(L) < k - 1:
    - return SELECT(R, k – len(L) – 1)

How about runtime?

- Let's try a recurrence relation…
- $$T(n) = \begin{cases} T\big(len(L)\big) + O(n) & len(L) < k - 1 \\ T\big(len(R)\big) + O(n) & len(L) > k - 1 \\ O(n) & len(L) = k - 1 \end{cases}$$

- What is len(L), len(R)?
- Let's **pretend** that len(L) is about ~~n/2~~.
- $T(n) \leq T\left(\frac{n}{2}\right) + O(n)$

Can we get away with n-1?

- SELECT(A, p=k):
  - **If** len(A) <= 150:
    - A = MergeSort(A)
    - Return A[k]
  - Choose p in {1,…,n}
  - L, A[p], R = PARTITION(A,p)
  - **If** len(L) = k - 1:
    - **Return** A[p]
  - **Else If** len(L) > k - 1:
    - **Return** SELECT(L, k)
  - **Else if** len(L) < k - 1:
    - return SELECT(R, k – len(L) – 1)

Recall the Master Theorem

which totally doesn't apply here, we are cheating by pretending we know the problem size.

- Suppose $T(n) = a \cdot T\left(\frac{n}{b}\right) + O(n^d)$. Then

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

In our case:

- $T(n) \leq T(n-1) + O(n)$
- So a = 1, b = 1/(1 -1/n) , d = 1
- $T(n) \leq O(n)$ still?
- NO!!! b needs to be independent of n for the master thm to work. Actual running time is O(n^2).

Lucky the Lackadaisical Lemur

How about runtime?

- Let's try a recurrence relation…
- $$T(n) = \begin{cases} T\big(len(L)\big) + O(n) & len(L) < k - 1 \\ T\big(len(R)\big) + O(n) & len(L) > k - 1 \\ O(n) & len(L) = k - 1 \end{cases}$$

- What is len(L), len(R)?
- Let's **pretend** that len(L) is about ~~n/2~~.
- $T(n) \leq T\left(\frac{n}{2}\right) + O(n)$
- $T(n) = O(n^2)$
- Not good enough ☹!

Can we get away with n-1?

- SELECT(A, p=k):
  - **If** len(A) <= 150:
    - A = MergeSort(A)
    - Return A[k]
  - Choose p in {1,…,n}
  - L, A[p], R = PARTITION(A,p)
  - **If** len(L) = k - 1:
    - **Return** A[p]
  - **Else If** len(L) > k - 1:
    - **Return** SELECT(L, k)
  - **Else if** len(L) < k - 1:
    - return SELECT(R, k – len(L) – 1)

Moral of this extremely shady logic

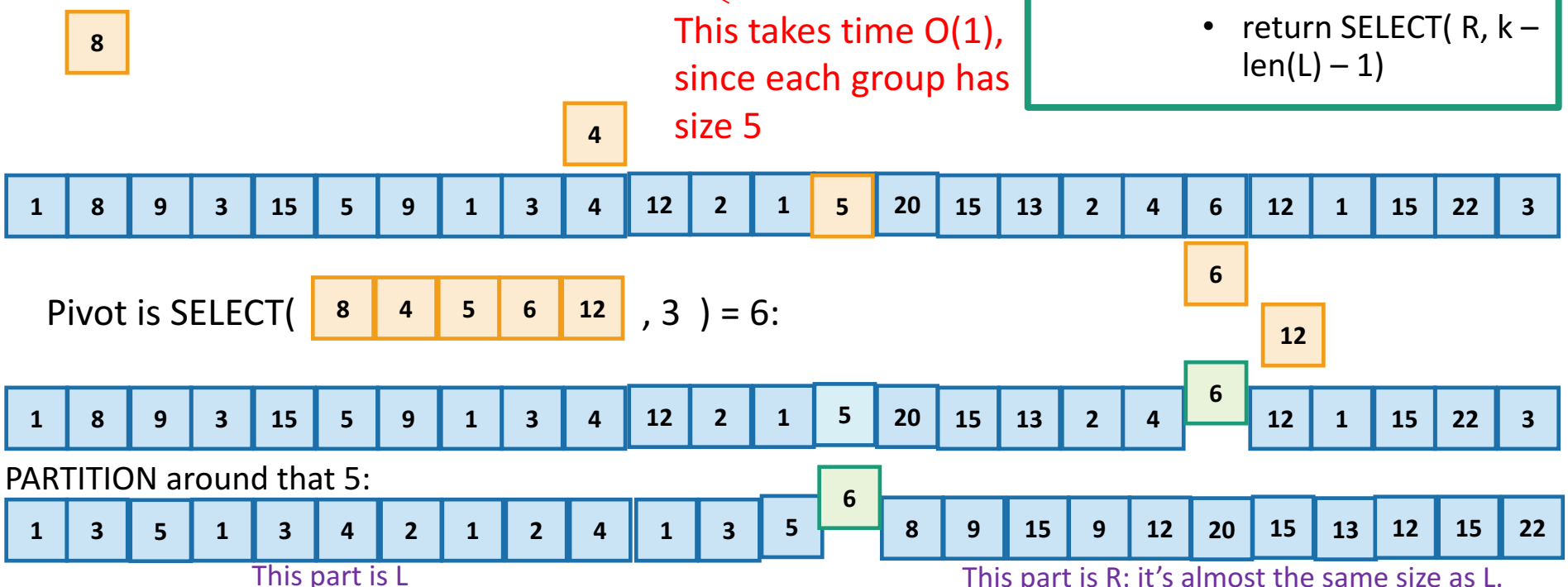
- If we can pick a pivot so that L and R **somewhat** balanced (even like 7n/10), then we're doing great. Otherwise, no good.
- **Try 1: Let's pick the pivot to be the median!**
  - Then L and R are always n/2. (or $\left\lfloor \frac{n}{2} \right\rfloor$ or $\left\lceil \frac{n}{2} \right\rceil$).
- **Problem:** That's exactly the problem we're trying to solve to begin with.
- **Solution:**
  - We can't find the median of n things (yet), but we can *recursively* find the median of n/5 things…
  - that will give us something "close enough" to the median that we can (rigorously) apply the previous analysis.

How to pick the pivot

- CHOOSEPIVOT(A):
  - Split A into m = $\lceil \frac{n}{5} \rceil$ 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, …, p_m] , m/2)
  - **return** p

- SELECT(A, p=k):
  - **If** len(A) <= 50:
    - A = MergeSort(A)
    - Return A[k]
  - p = CHOOSEPIVOT(A)
  - L, A[p], R = PARTITION(A,p)
  - **If** len(L) = k - 1:
    - **Return** A[p]
  - **Else If** len(L) < k - 1:
    - **Return** SELECT(L, k)
  - **Else if** len(L) > k - 1:
    - return SELECT(R, k – len(L) – 1)

This takes time O(1), since each group has size 5



So this gives the whole algorithm

- SELECT(A, p=k):
  - **If** len(A) <= 50:
    - A = MergeSort(A)
    - Return A[k]
  - p = CHOOSEPIVOT(A)
  - L, A[p], R = PARTITION(A,p)
  - **If** len(L) = k - 1:
    - **Return** A[p]
  - **Else If** len(L) > k - 1:
    - **Return** SELECT(L, k)
  - **Else if** len(L) < k - 1:
    - return SELECT(R, k – len(L) – 1)

- PARTITION(A, p):
  - L = new array
  - R = new array
  - **For** i=1,…,n:
    - **If** i==p, continue
    - **Else If** A[i] <= A[p]:
      - L.append(A[i])
    - **Else if** A[i] > A[p]:
      - R.append(A[i])
  - **Return** L, A[p], R

Note: We use recursion in two ways! Both in SELECT itself, and in CHOOSEPIVOT.

- 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** p

- Does it work?
- Yes, our proof before worked for any pivoting strategy.

What's the runtime?

Outline for the rest of today

- Lemma: each of the arrays L and R are pretty balanced.
- The substitution method:
  - A "guess-and-check" approach to solving recurrence relations.
  - Works even when the Master Theorem doesn't.
- We will use the substitution method to see that SELECT indeed runs in time O(n).

- (Time permitting, we'll also see how NOT to use the substitution method. If we don't have time in class, look ahead at the slides on your own.)

Lemma: that median-of-medians thing is a good idea.

- **Lemma:** If L and R are as in the algorithm SELECT given above, then

$$|L| \leq \frac{7n}{10} + 5$$

and

$$|R| \leq \frac{7n}{10} + 5$$

Arbitrary numbers!
I thought the whole point of O() was that we'd never have to do that again.

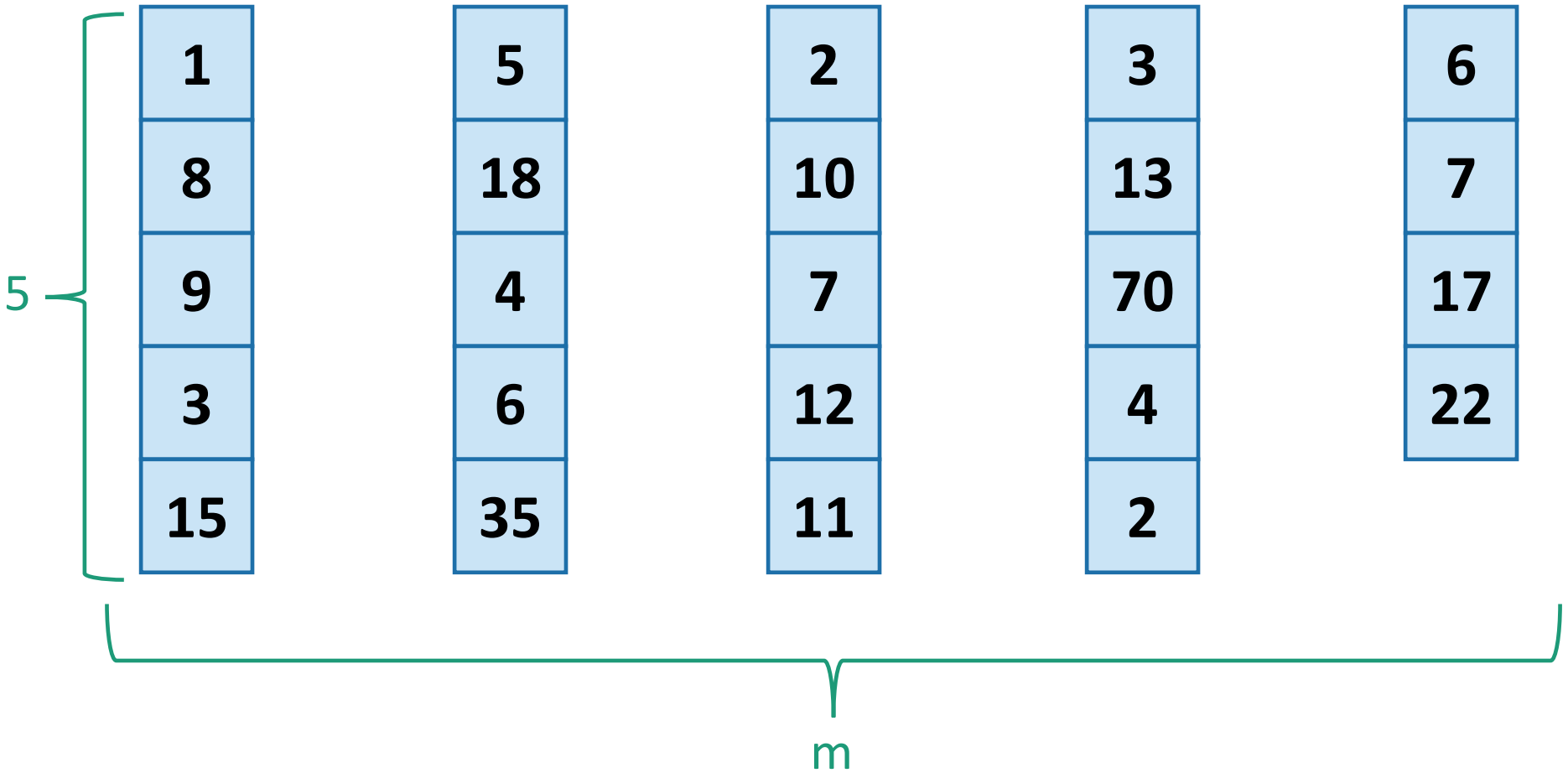
We're only doing it so we can get a (correct) O() bound later…so our choice of numbers **will** be pretty arbitrary.

- Why is this good?
  - It means that things are pretty balanced.
- We will see a proof by picture.
- See CLRS or the Lecture Notes for proof by proof.

Proof by picture generally not okay on HW – need at least a few words. (But pictures are encouraged if it makes things clearer!)

grumpy cat

Proof by picture



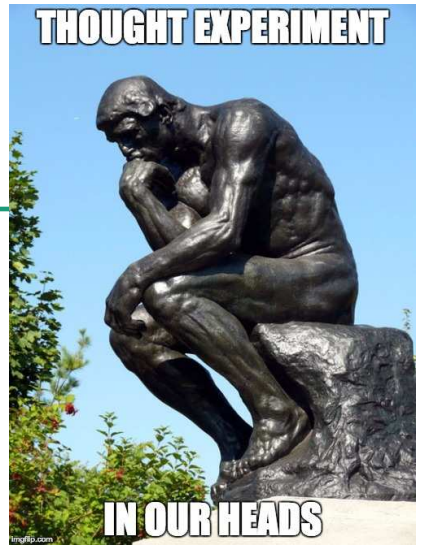
Say these are our m = [n/5] sub-arrays of size at most 5.

Proof by picture

1	4	2	2	6
3	5	7	3	7
8	6	10	4	17
9	18	11	13	22
15	35	12	70	

5

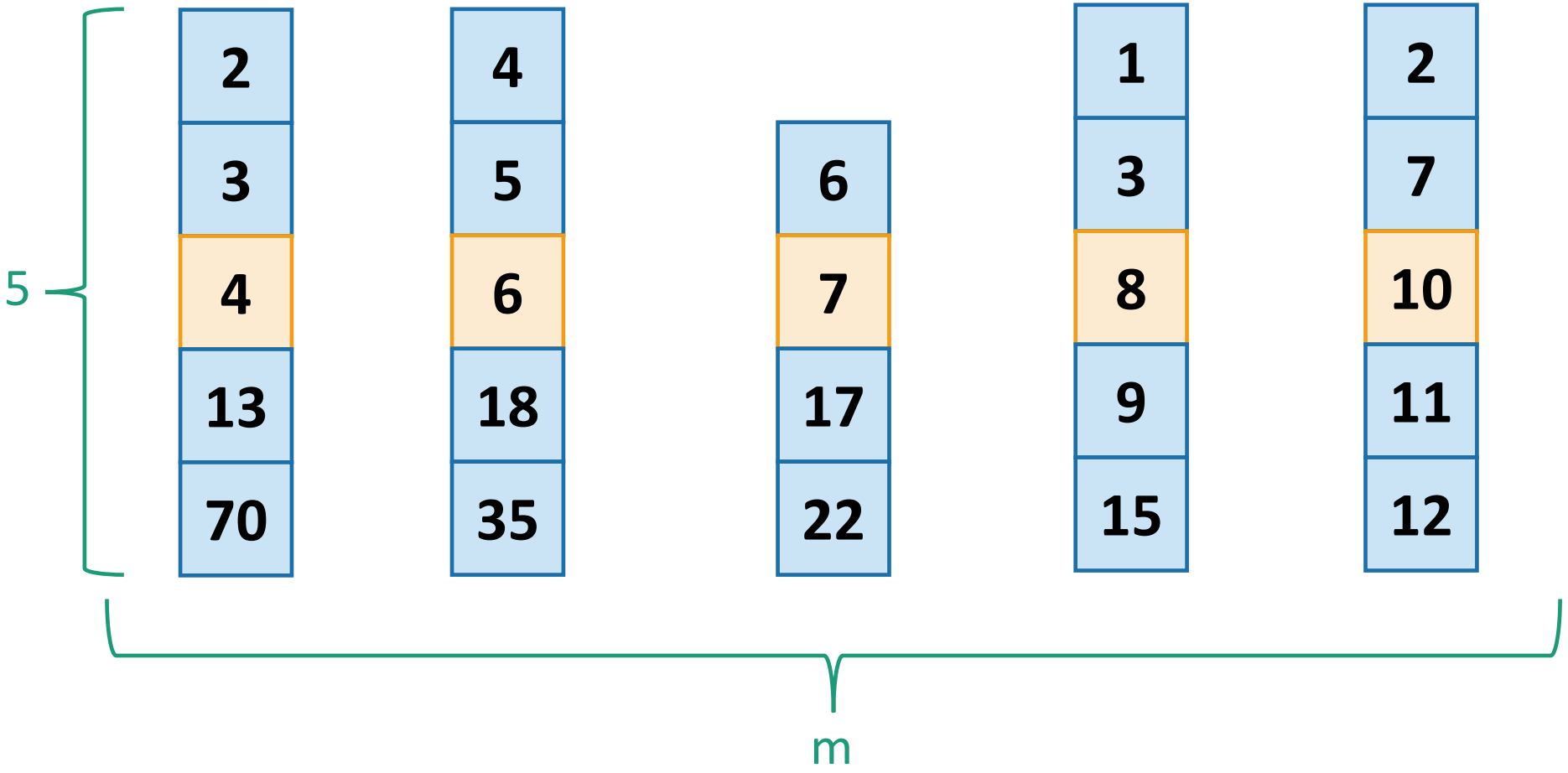
m



In our head, let's sort them.

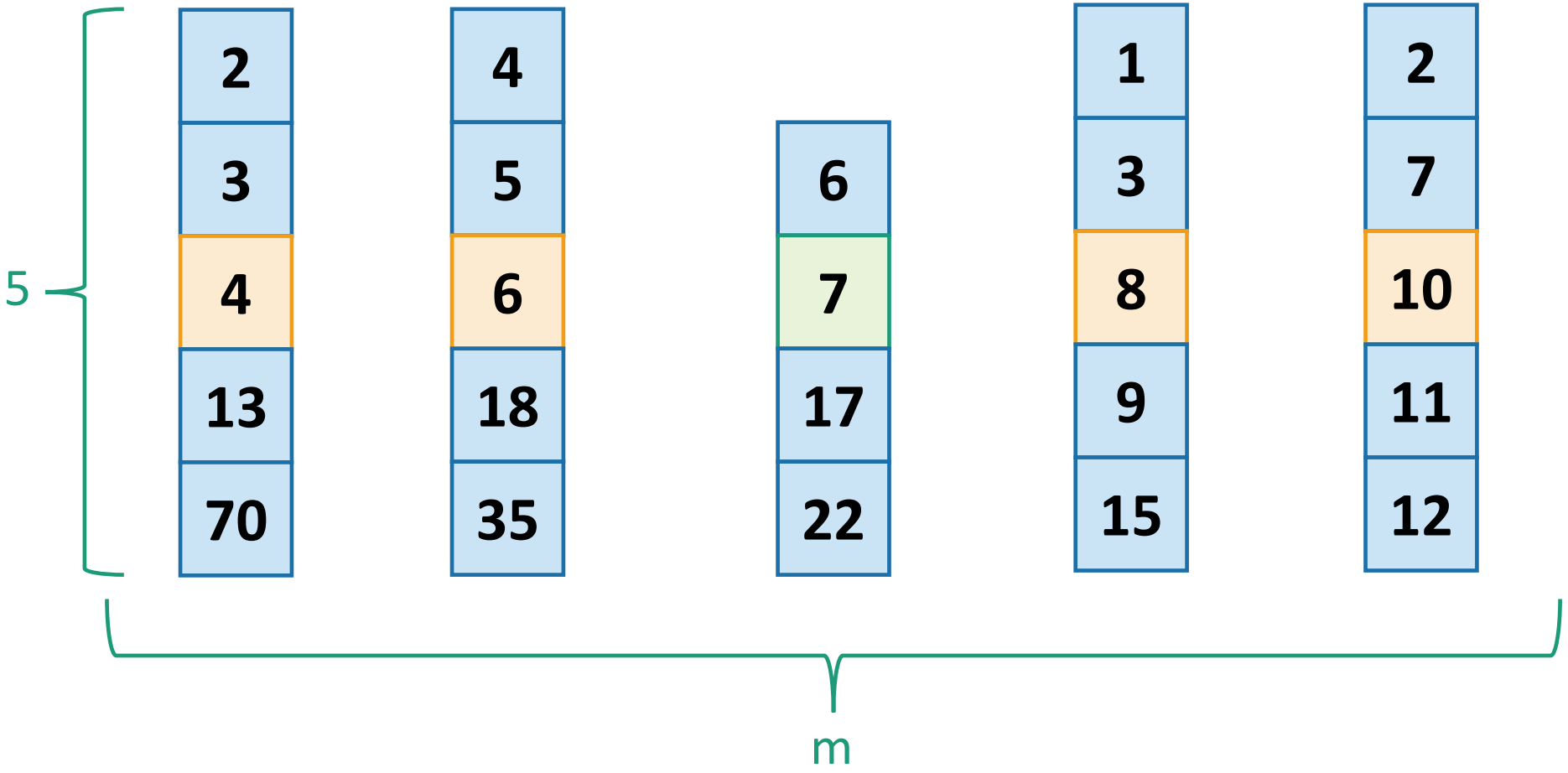
Then find medians.

Proof by picture



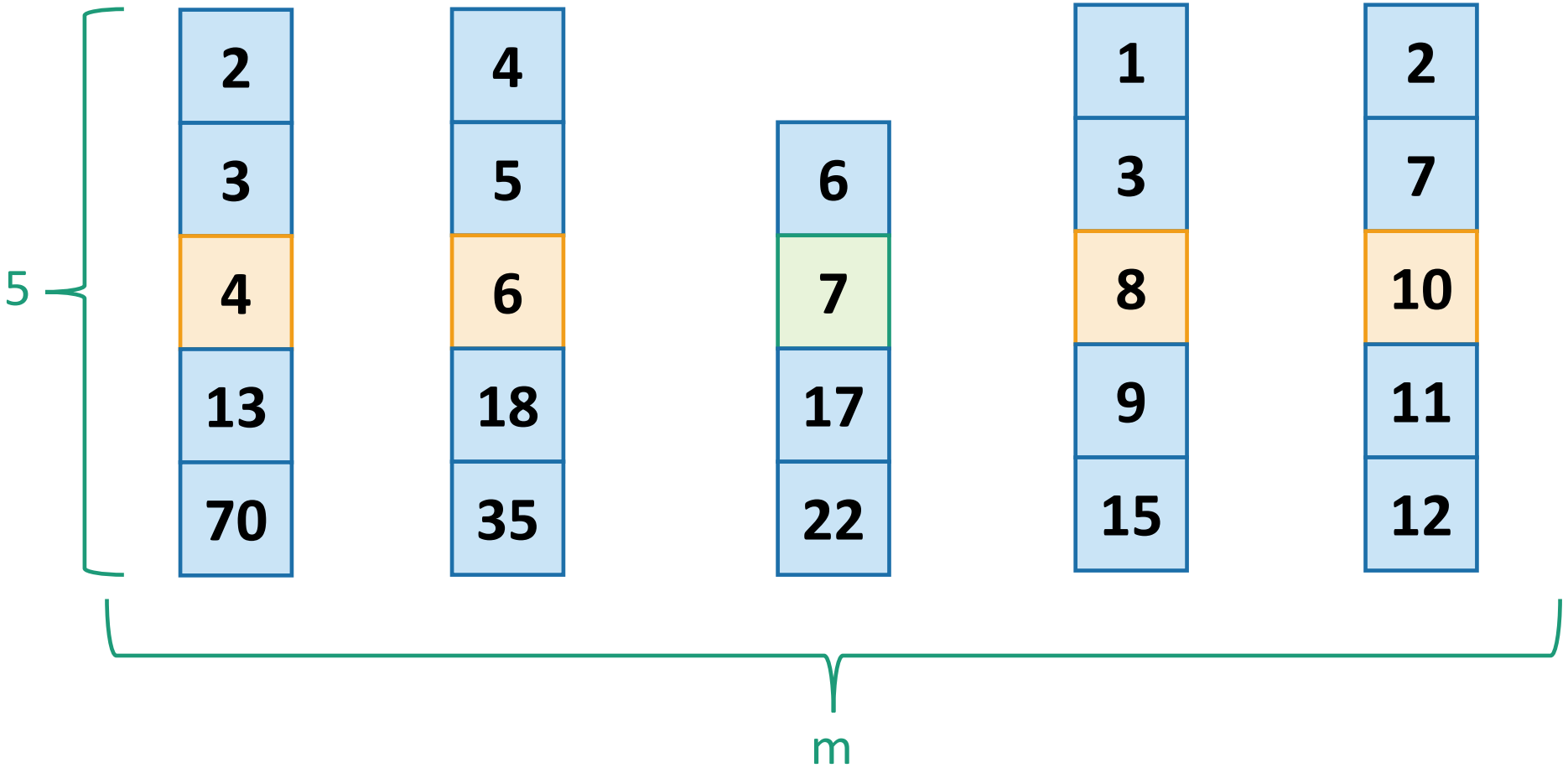
Then let's sort them by the median

Proof by picture



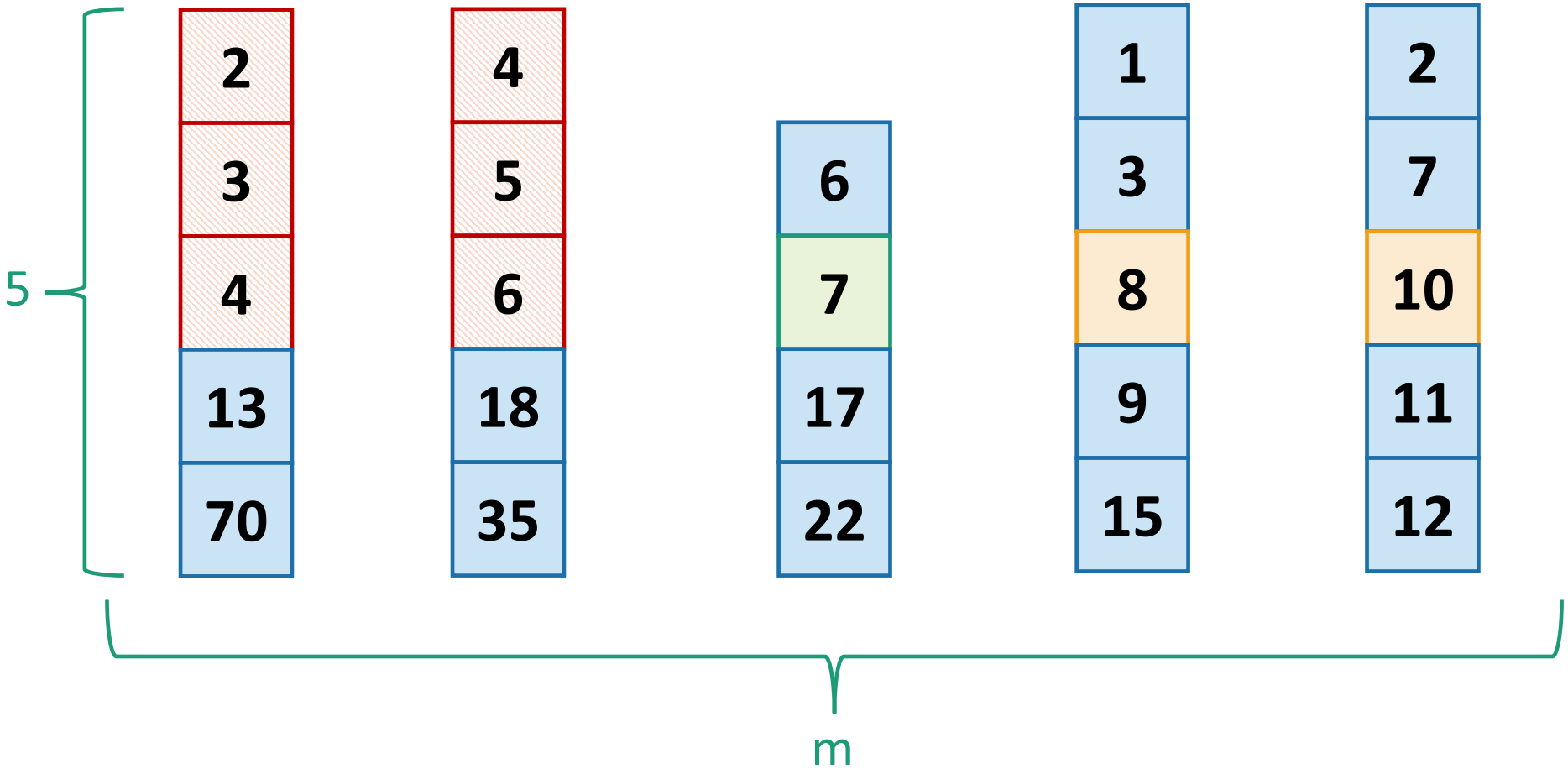
The median of the medians is 7. That's our pivot!

Proof by picture



How many elements are SMALLER than the pivot?

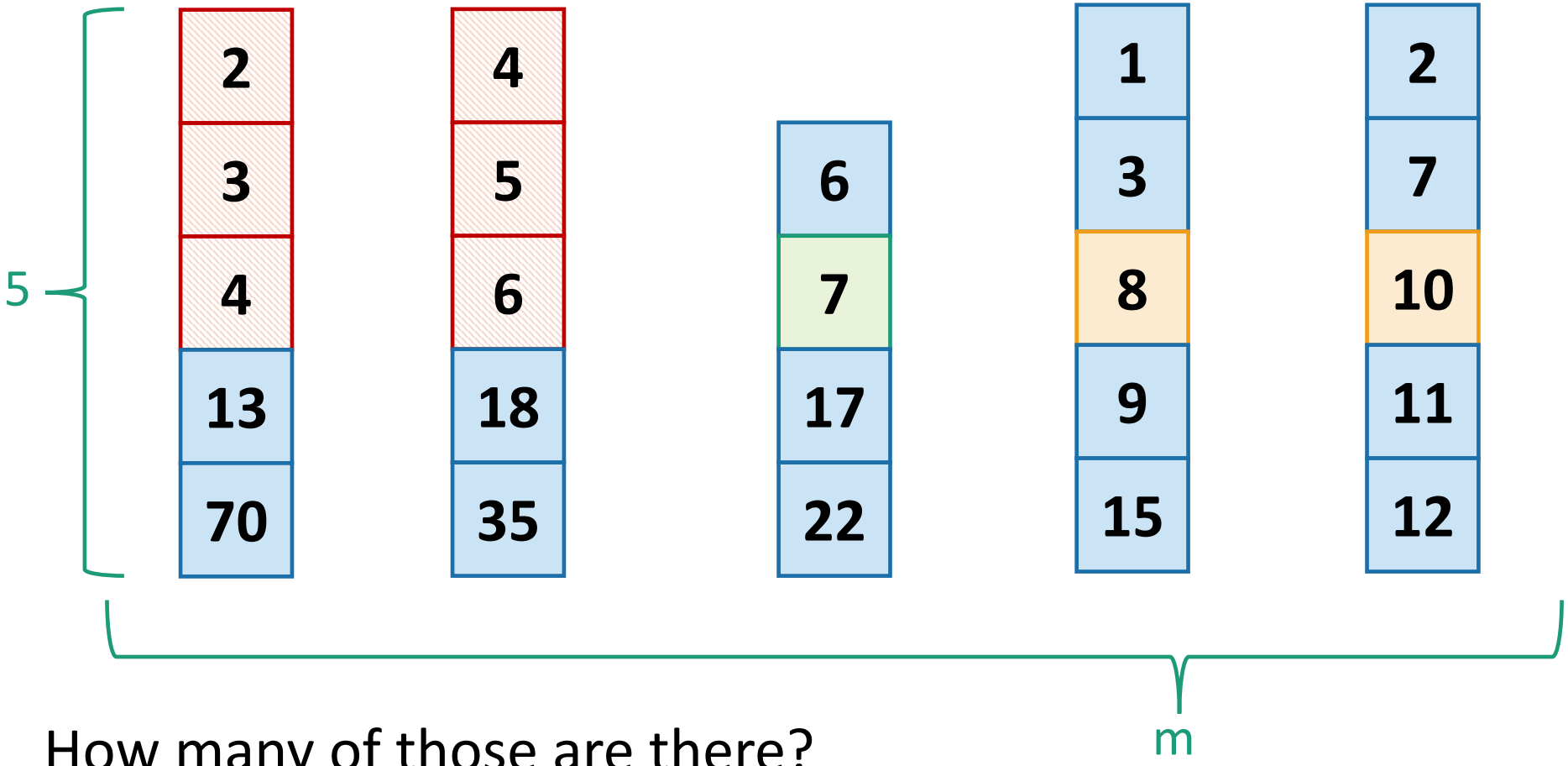
Proof by picture



At least these ones: everything above and to the left.

Proof by picture

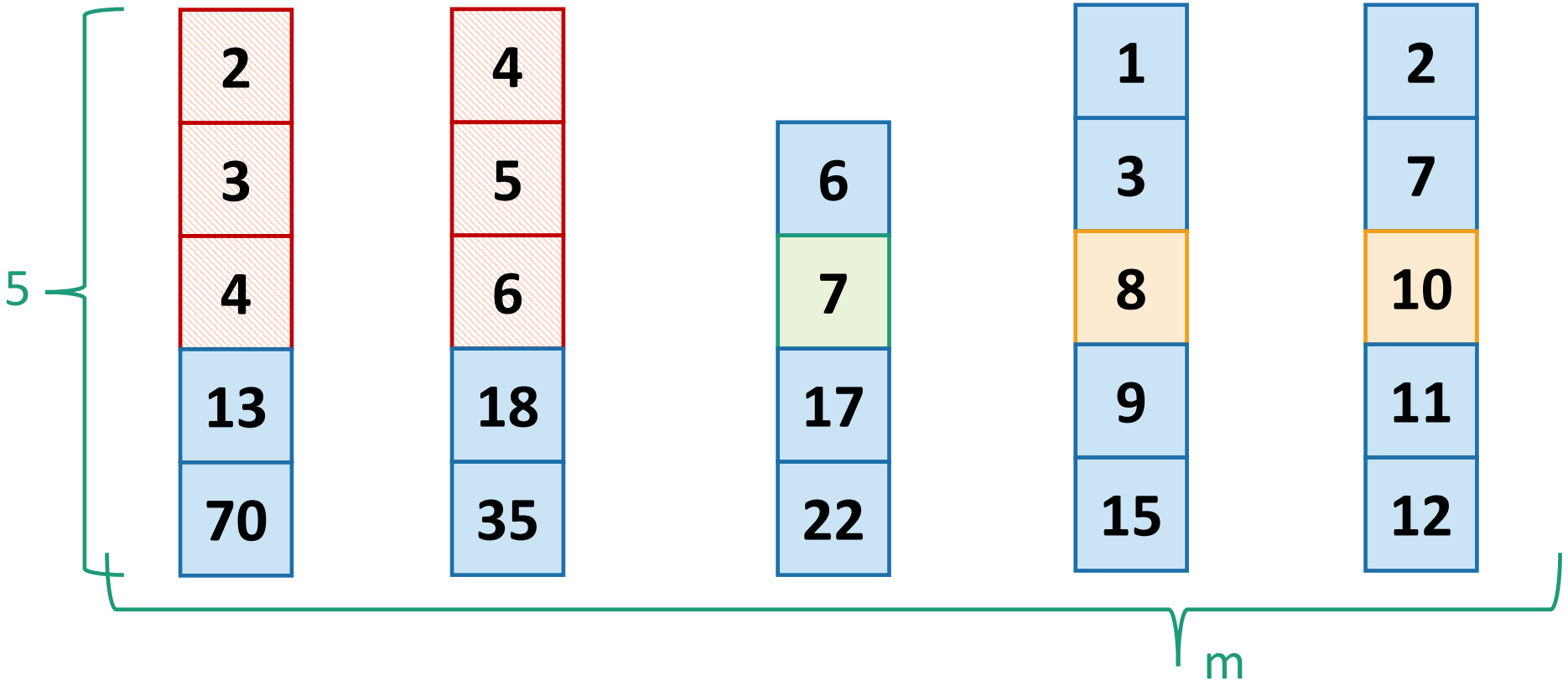
$3 \cdot \left(\left\lceil \frac{m}{2} \right\rceil - 1\right)$ of these, but then one of them could have been the "leftovers" group.



How many of those are there?

at least $3 \cdot \left(\left\lceil \frac{m}{2} \right\rceil - 2\right)$

Proof by picture



So how many are LARGER than the pivot? At most

$$n - 1 - 3\left(\left\lceil\frac{m}{2}\right\rceil - 2\right) \leq \frac{7n}{10} + 5$$

(derivation on board)

Remember $m = \left\lceil\frac{n}{5}\right\rceil$

That was one part of the lemma

- **Lemma:** If L and R are as in the algorithm SELECT given above, then

$$|L| \leq \frac{7n}{10} + 5$$

and

$$|R| \leq \frac{7n}{10} + 5$$

The other part is exactly the same.

Why is this useful?

The substitution method

- Recursion trees can get pretty messy here, since we have a recurrence relation that doesn't nicely break up our big problem into sub-problems of the same size.
- Instead, we will try to:
  - Make a guess
  - Check using an inductive argument
- This is called the substitution method.

Substitution method

work to call recursive sub-problems and merge them back

- Suppose that $T(n) \leq c \cdot f(n) + \sum_{i=1}^{r} T(n_i)$

Work in r different sub-problems, which might have different sizes.

- Let's guess the solution is

$$T(n) \leq \begin{cases} d \cdot g(n_0) & if \quad n \leq n_0 \\ d \cdot g(n) & if \quad n > n_0 \end{cases} \qquad (*)$$

- (aka, guessing T(n) = O(g(n)))
- We'll prove this by induction, with the inductive hypothesis (*) for all smaller n's.

In our case

The cn is the O(n) work done at each level for PARTITION

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

The T(7n/10 + 5) is for the recursive call to SELECT for either L or R.

The T(n/5) is for the recursive call to get the median in FINDPIVOT

- Inductive hypothesis:
- $T(n) \leq \begin{cases} d \cdot 50 & if \ \ n \leq 50 \\ d \cdot n & if \ \ n > 50 \end{cases}$

(aka, T(n) = O(n)).

for d = 10c.

So let's prove this.

$$(*)\quad T(n) \le \begin{cases} d \cdot 50 & if\ \ n \le 50 \\ d \cdot n & if\ \ n > 50 \end{cases}$$

for d = 10c.

- Base case:
  - if n <= 50, our algorithm was: run MergeSort on <=50 things.
  - By maybe allowing c to be a bit bigger, that takes time at most 50d. *(WHY IS THIS OKAY?)*
- Inductive step: Suppose (*) holds for all sizes < n. Then
- $T(n) \le c \cdot n + T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10} + 5\right)$

$$\le c \cdot n + d \cdot \frac{n}{5} + d \cdot \left(\frac{7n}{10} + 5\right)$$

$$\le n\left(c + \frac{d}{5} + \frac{7d}{10}\right) + 5d$$

$$\le n\left(c + \frac{10c}{5} + \frac{70 \cdot c}{10}\right) + 50\,c$$

$$= (9\,n + 50\,)c$$

$$\le 10c \cdot n = d \cdot n \quad \text{whenever n > 50.}$$

This is a pretty pedantic proof! But it's worth being careful about the constants for the substitution method, as we'll see in a bit.

This is why we had the base case at 50 before.

Nearly there!

$$(*)\quad T(n) \leq \begin{cases} d \cdot 50 & if\ \ n \leq 50 \\ d \cdot n & if\ \ n > 50 \end{cases}$$

for d = 10c.

- By induction, this shows that the inductive hypothesis (*) applies for all n.
- Termination: Observe that this is exactly what we wanted to show!
  - There exists a constant d (which depends on the constant c from the running time of PARTITION…) and an n_0 (aka 50) so that for all $n > n_0$, $T(n) <= d\ n$.
  - By definition, T(n) = O(n).
  - Hooray!
- Conclusion: We can implement SELECT (and in particular, MEDIAN) in time O(n).

That was pretty pedantic

- Can't we just use the nice O() notation?
- $T(n) \leq c \cdot n + T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10} + 5\right)$
- Inductive hypothesis: T(n) = O(n).

vs.

$$T(n) \leq \begin{cases} d \cdot 50 & if \ \ n \leq 50 \\ d \cdot n & if \ \ n > 50 \end{cases}$$

- Then the inductive step is just

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

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

Actually that doesn't work

- Consider this "proof" that MERGESORT runs in time O(n). (It doesn't).
- $T(n) \leq 2\,T\left(\frac{n}{2}\right) + c \cdot n$
- Inductive hypothesis: T(n) = O(n).

- Then the inductive step is just

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

- What's wrong???
  - (It turns out the base case is fine).

The problem is being sloppy with O()

- When we use O() in the inductive hypothesis, it might have a different constant "c" in the definition of O() each time the hypothesis is called. As a result, this "constant" might depend on n, which is not allowed in the definition of O().
- Try rigorously:
  - Suppose that $T(n) \leq c \cdot n + 2\,T\left(\frac{n}{2}\right)$
  - Let's guess the solution is $T(n) \leq \begin{cases} d \cdot n_0 & if \quad n \leq n_0 \\ d \cdot n & if \quad n > n_0 \end{cases}$
  - Then the inductive argument would go….
  - $T(n) = 2T\left(\frac{n}{2}\right) + c \cdot n \leq 2 \cdot \frac{dn}{2} + c \cdot n = (d + c)n$
  - We need that to be smaller than dn for the induction to work.
  - No way that's going to happen, since c > 0.
  - So this argument won't work. (Which is good, since the statement is false).

Told you so.

Recap

- We saw a (pretty clever) algorithm to do SELECT in time O(n).
- We proved that it worked using the Substitution Method.
- The Master Theorem wouldn't have worked for this.
- In practice for this algorithm, it's often better to just choose the pivot randomly. You'll see an analysis of that if you take CS265 (randomized algorithms).
- We'll also see some randomized algorithms…

…next time