

# Lecture

## Median and Selection

# Last Time:

## Solving Recurrence Relations

- A **recurrence relation** expresses  $T(n)$  in terms of  $T(\text{less than } n)$
- For example,  $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + 11 \cdot n$
- Two methods of solution:
  1. Master Theorem (aka, generalized “tree method”)
  2. Substitution method (aka, guess and check)

# The Master Theorem

- Suppose  $a \geq 1$ ,  $b > 1$ , and  $d$  are constants (that don't depend on  $n$ ).
- 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.

A powerful  
theorem it is...

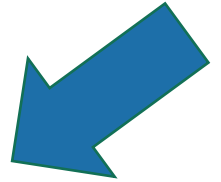


Jedi master Yoda

# The Substitution Method

- Step 1: Guess what the answer is.
- Step 2: Prove by induction that your guess is correct.
- Step 3: Profit.

# The plan for today

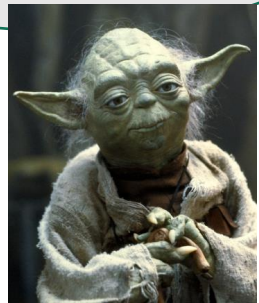


1. More practice with the Substitution Method.
2. k-SELECT problem
3. k-SELECT solution
4. Return of the Substitution Method.

# A fun recurrence relation

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

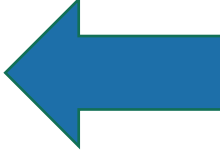
Apply here, the  
Master Theorem does  
NOT.



# What have we learned?

- The substitution method can work when the master theorem doesn't.
  - For example with different-sized sub-problems.

# The Plan

1. More practice with the Substitution Method.
2. k-SELECT problem 
3. k-SELECT solution
4. Return of the Substitution Method.



# The k-SELECT problem

*For today, assume  
all arrays have  
distinct elements.*

A is an array of size n, k is in  $\{1, \dots, n\}$

- **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)

# An $O(n \log(n))$ -time algorithm

- **SELECT**(A, k):

- A = MergeSort(A)
- **return** A[k-1]

*It's k-1 and not k since my pseudocode is 0-indexed and the problem is 1-indexed...*

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

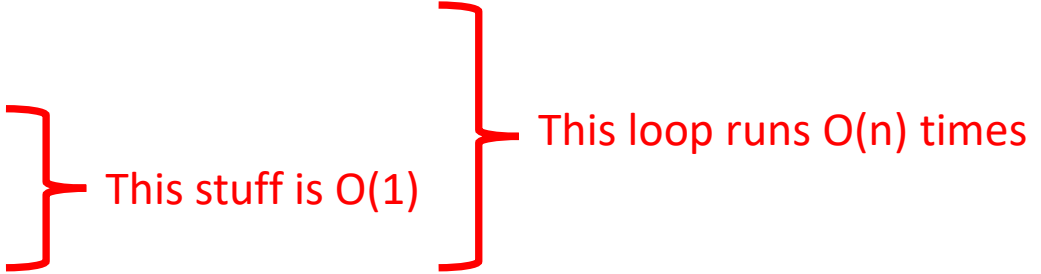
Can we do better?

We're hoping to get  $O(n)$

Show that you can't do better than  $O(n)$ .



# Goal: An $O(n)$ -time algorithm

- **SELECT**(A, 1).
    - (aka, **MIN**(A))
  - **MIN**(A):
    - $\text{ret} = \infty$
    - **For**  $i=0, \dots, n-1$ :
      - If  $A[i] < \text{ret}$ :
        - $\text{ret} = A[i]$
    - **Return**  $\text{ret}$
- 
- Time  $O(n)$ . Yay!

# How about SELECT(A,2)?

- **SELECT2(A):**
  - $ret = \infty$
  - $minSoFar = \infty$
  - **For**  $i=0, \dots, n-1$ :
    - **If**  $A[i] < ret$  and  $A[i] < minSoFar$ :
      - $ret = minSoFar$
      - $minSoFar = A[i]$
    - **Else if**  $A[i] < ret$  and  $A[i] \geq minSoFar$ :
      - $ret = A[i]$
  - **Return**  $ret$

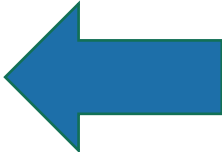
(The actual algorithm here is not very important because this won't end up being a very good idea...)

Still  $O(n)$   
SO FAR SO GOOD.

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

- MEDIAN(A):
  - $ret = \infty$
  - $minSoFar = \infty$
  - $secondMinSoFar = \infty$
  - $thirdMinSoFar = \infty$
  - $fourthMinSoFar = \infty$
  - ....
- 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  $O(n^2)$ .

# The Plan

1. More practice with the Substitution Method.
2. k-SELECT problem
3. k-SELECT solution 
4. Return of the Substitution Method.

# Idea: divide and conquer!

Say we want to  
find `SELECT(A, k)`



How about  
this pivot?

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).

9/16/2024 L = array with things  
smaller than  $A[\text{pivot}]$

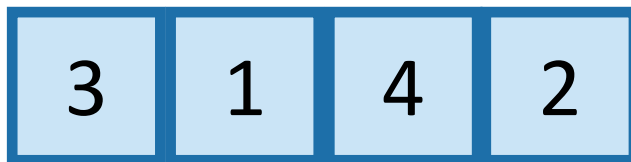
R = array with things  
larger than  $A[\text{pivot}]$

# Idea: divide and conquer!

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”



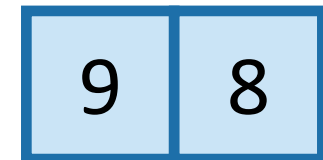
L = array with things  
smaller than A[pivot]

9/16/2024



How about  
this pivot?

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

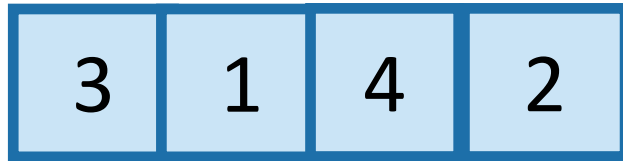


R = array with things  
larger than A[pivot]

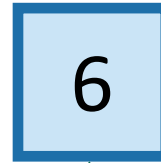


# Idea continued...

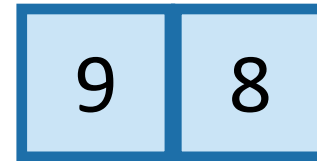
Say we want to  
find `SELECT(A, k)`



L = array with things  
smaller than A[pivot]



pivot



R = array with things  
larger than A[pivot]

- If  $k = 5 = \text{len}(L) + 1$ :
  - 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...)

# Pseudocode

- **getPivot** ( $A$ ) returns some pivot for us.
  - How?? We'll see later...
- **Partition** ( $A, p$ ) splits up  $A$  into  $L, A[p], R$ .

- **Select**( $A, k$ ):
  - **If**  $\text{len}(A) \leq 50$ :
    - $A = \text{MergeSort}(A)$
    - **Return**  $A[k-1]$
  - $p = \text{getPivot}(A)$
  - $L, \text{pivotVal}, R = \text{Partition}(A, p)$
  - **if**  $\text{len}(L) == k-1$ :
    - **return**  $\text{pivotVal}$
  - **Else if**  $\text{len}(L) > k-1$ :
    - **return** **Select**( $L, k$ )
  - **Else if**  $\text{len}(L) < k-1$ :
    - **return** **Select**( $R, k - \text{len}(L) - 1$ )

**Base Case:** If the  $\text{len}(A) = O(1)$ , 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

# What is the running time?

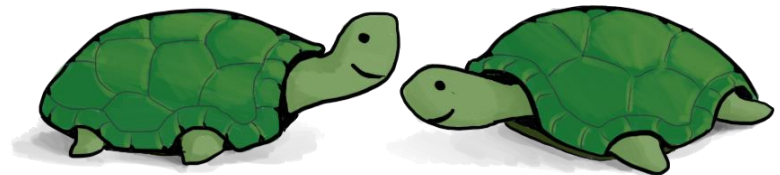
Assuming we pick the pivot in time  $O(n)$ ...

$$\bullet T(n) = \begin{cases} T(\text{len}(\mathbf{L})) + O(n) & \text{len}(\mathbf{L}) > k - 1 \\ T(\text{len}(\mathbf{R})) + O(n) & \text{len}(\mathbf{L}) < k - 1 \\ O(n) & \text{len}(\mathbf{L}) = k - 1 \end{cases}$$

- What are **len(L)** and **len(R)**?
- That depends on how we pick the pivot...

Think: two minutes  
Pair and share: one minute

What would be a “good” pivot?  
What would be a “bad” pivot?



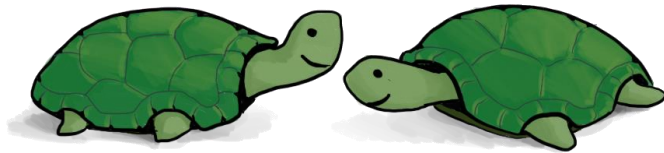
Think-Pair-Share Terrapins



# The ideal pivot

- We split the input exactly in half:
  - $\text{len}(\text{L}) = \text{len}(\text{R}) = (n-1)/2$

What would be the running time in that case?  
(If we could always choose that ideal pivot)



Think: two minutes  
Pair and share: one minute

In case it's helpful...

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

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

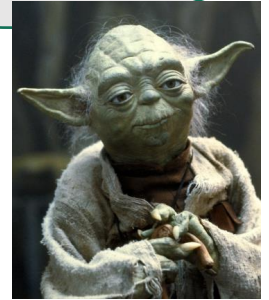
$$T(n) = \begin{cases} T(\text{len}(\text{L})) + O(n) & \text{len}(\text{L}) > k - 1 \\ T(\text{len}(\text{R})) + O(n) & \text{len}(\text{L}) < k - 1 \\ O(n) & \text{len}(\text{L}) = k - 1 \end{cases}$$

# The ideal pivot

- We split the input exactly in half:
  - $\text{len}(L) = \text{len}(R) = (n-1)/2$

Apply here, the Master Theorem does NOT. Making unsubstantiated assumptions about problem sizes, we are.

- Let's pretend that's the case and use the **Master Theorem!**



Jedi master Yoda

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

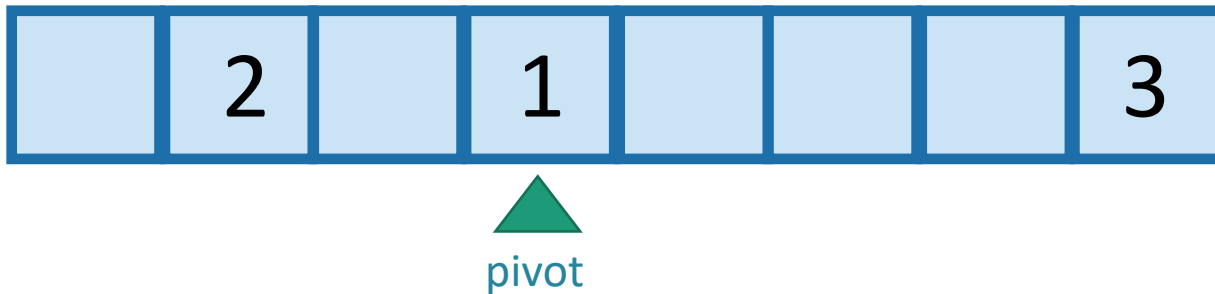
- Suppose  $T(n) = a \cdot T\left(\frac{n}{b}\right) + O(n^d)$ . Then

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

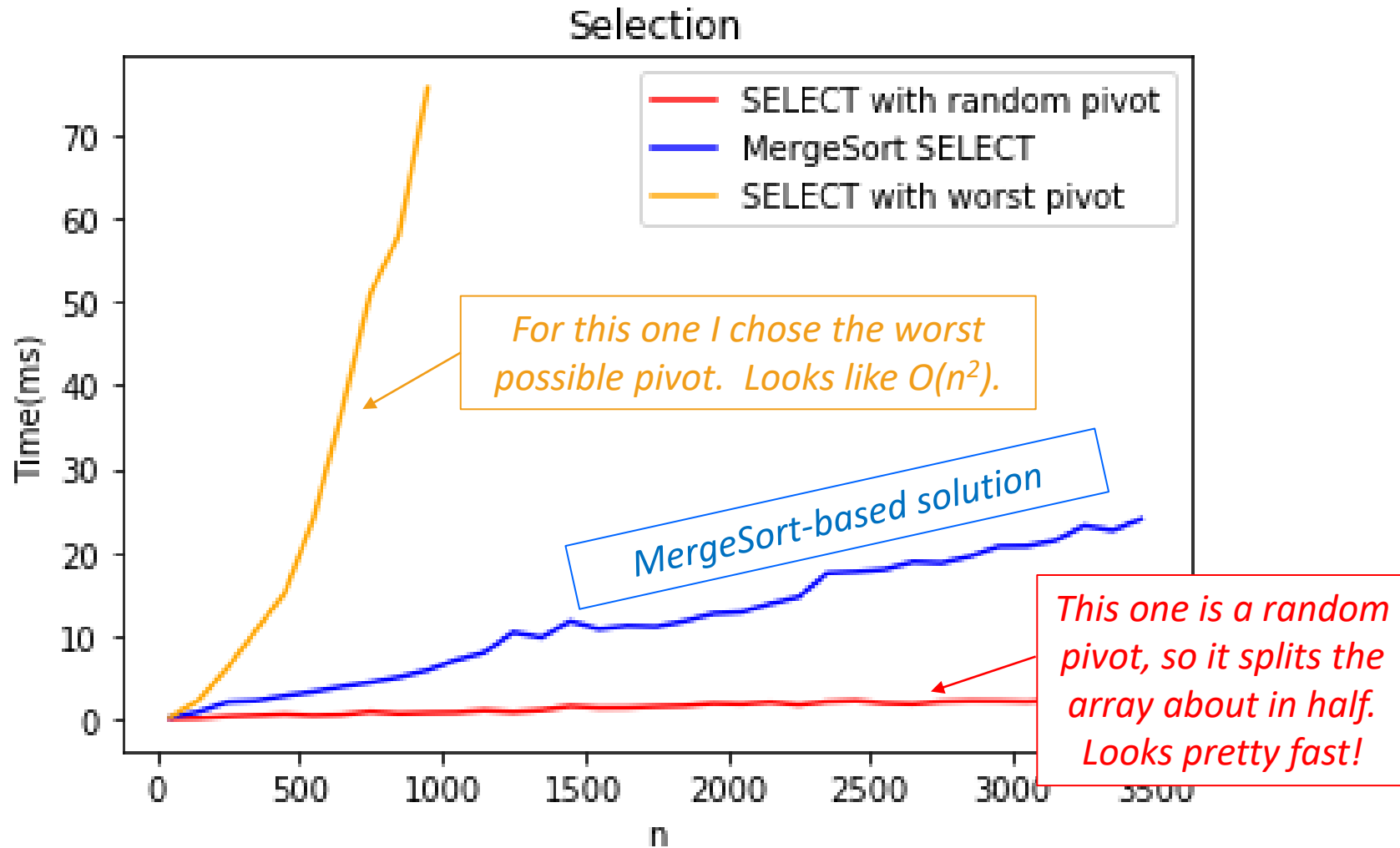
***That would be great!***

# The worst pivot

- Say our choice of pivot doesn't depend on A.
- A bad guy who **knows what pivots we will choose** gets to come up with A.



# The distinction matters!



# How do we pick a good pivot?

- Randomly?
  - That works well if there's no bad guy.
  - But if there is a bad guy who gets to see our pivot choices, that's just as bad as the worst-case pivot.

---

## Aside:

- In practice, there is often no bad guy. In that case, just pick a random pivot and it works really well!






# How do we pick a good pivot?

- For today, let's assume there's this bad guy.
- Reasons:
  - This gives us a very strong guarantee
  - We'll get to see a **really clever algorithm**.
    - Necessarily it will look at A to pick the pivot.
  - We'll get to use the **substitution method**.



# The Plan

1. More practice with the Substitution Method.
  2. k-SELECT problem
  3. k-SELECT solution
    - a) The outline of the algorithm.
    - b) How to pick the pivot.
  4. Return of the Substitution Method.
- 

# Approach

- First, we'll figure out what the ideal pivot would be.
  - But we won't be able to get it.
- Then, we'll figure out what a **pretty good** pivot would be.
  - But we still won't know how to get it.
- Finally, we will see how to get our pretty good pivot!
  - And then we will celebrate. 🎉

# How do we pick our ideal pivot?

- We'd like to live in the ideal world.
- Pick the pivot to divide the input in half.
- Aka, pick the median!
- Aka, pick `SELECT(A, n/2)!`



# How about a good enough pivot?

- We'd like to **approximate** the ideal world.



- Pick the pivot to divide the input **about** in half!
- Maybe this is easier!



# A good enough pivot

We still don't know that we can get such a pivot, but at least it gives us a goal and a direction to pursue!

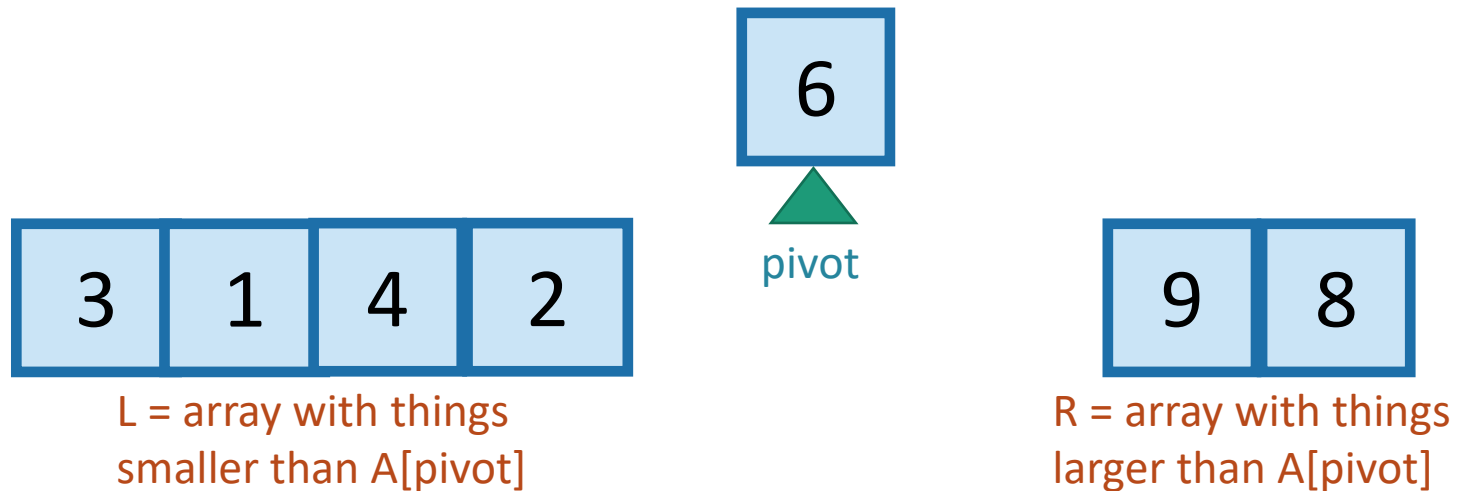


Lucky the lackadaisical lemur

- We split the input not quite in half:
  - $3n/10 < \text{len}(L) < 7n/10$
  - $3n/10 < \text{len}(R) < 7n/10$
- If we could do that (let's say, in time  $O(n)$ ), the **Master Theorem** would say:
  - $T(n) \leq T\left(\frac{7n}{10}\right) + O(n)$
  - So  $a = 1$ ,  $b = 10/7$ ,  $d = 1$
  - $T(n) \leq O(n^d) = O(n)$
- Suppose  $T(n) = a \cdot T\left(\frac{n}{b}\right) + O(n^d)$ . Then
$$T(n) = \begin{cases} O(n^d \log(n)) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

# Goal

- Efficiently pick the pivot so that

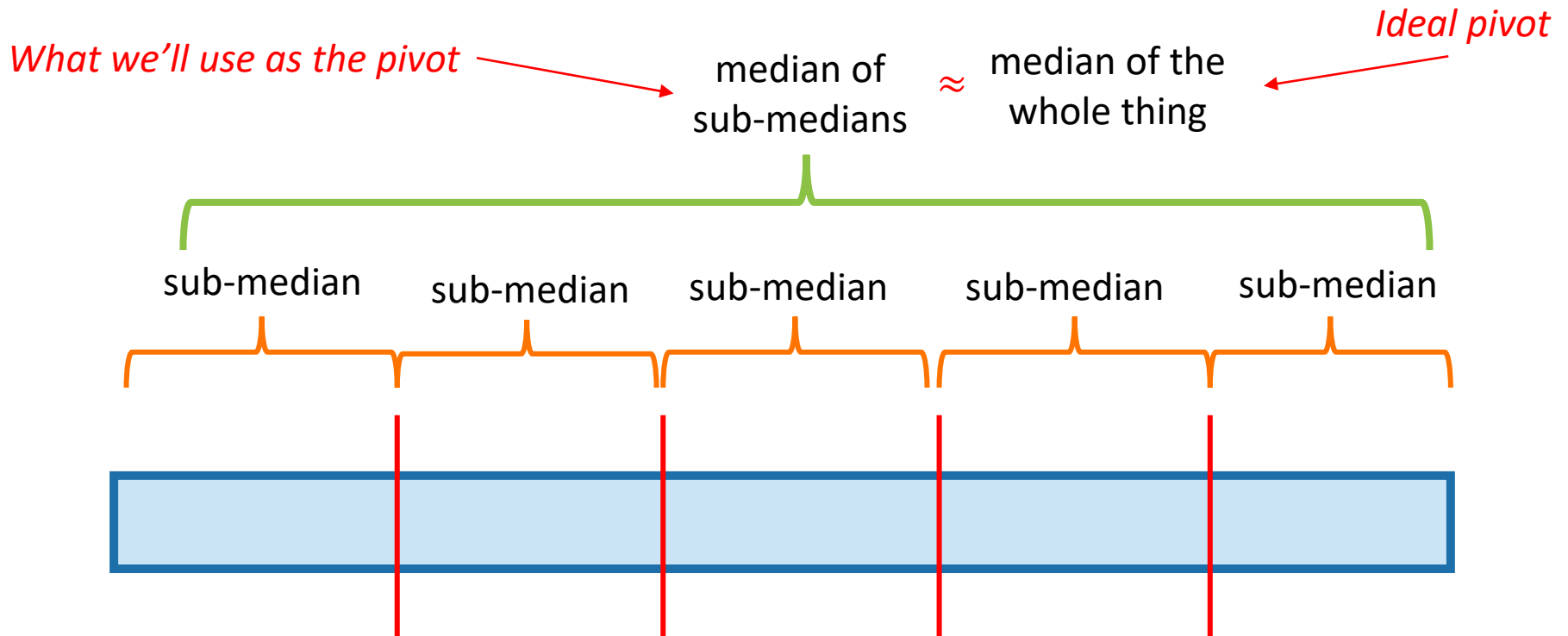


$$\frac{3n}{10} < \text{len}(L) < \frac{7n}{10}$$

$$\frac{3n}{10} < \text{len}(R) < \frac{7n}{10}$$

# Another divide-and-conquer alg!

- We can't solve  $\text{SELECT}(A, n/2)$  (yet)
- But we can divide and conquer and solve  $\text{SELECT}(B, m/2)$  for smaller values of  $m$  (where  $\text{len}(B) = m$ ).
- Lemma\*: The median of sub-medians is close to the median.

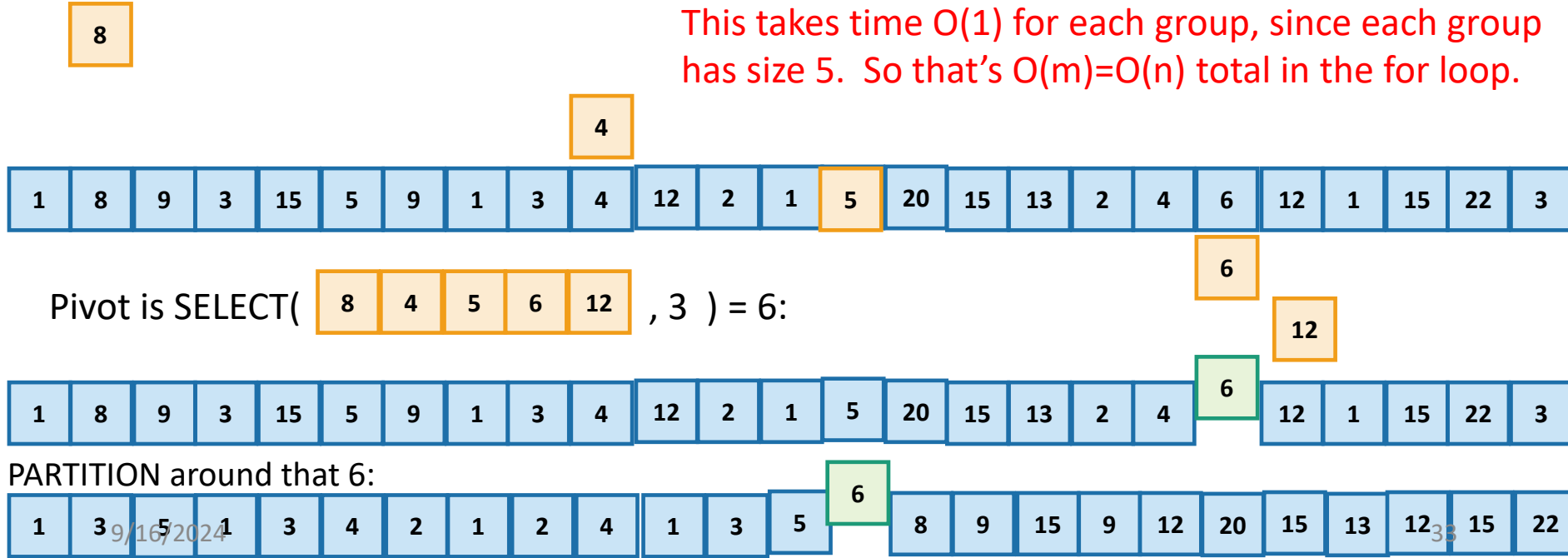




# How to pick the pivot

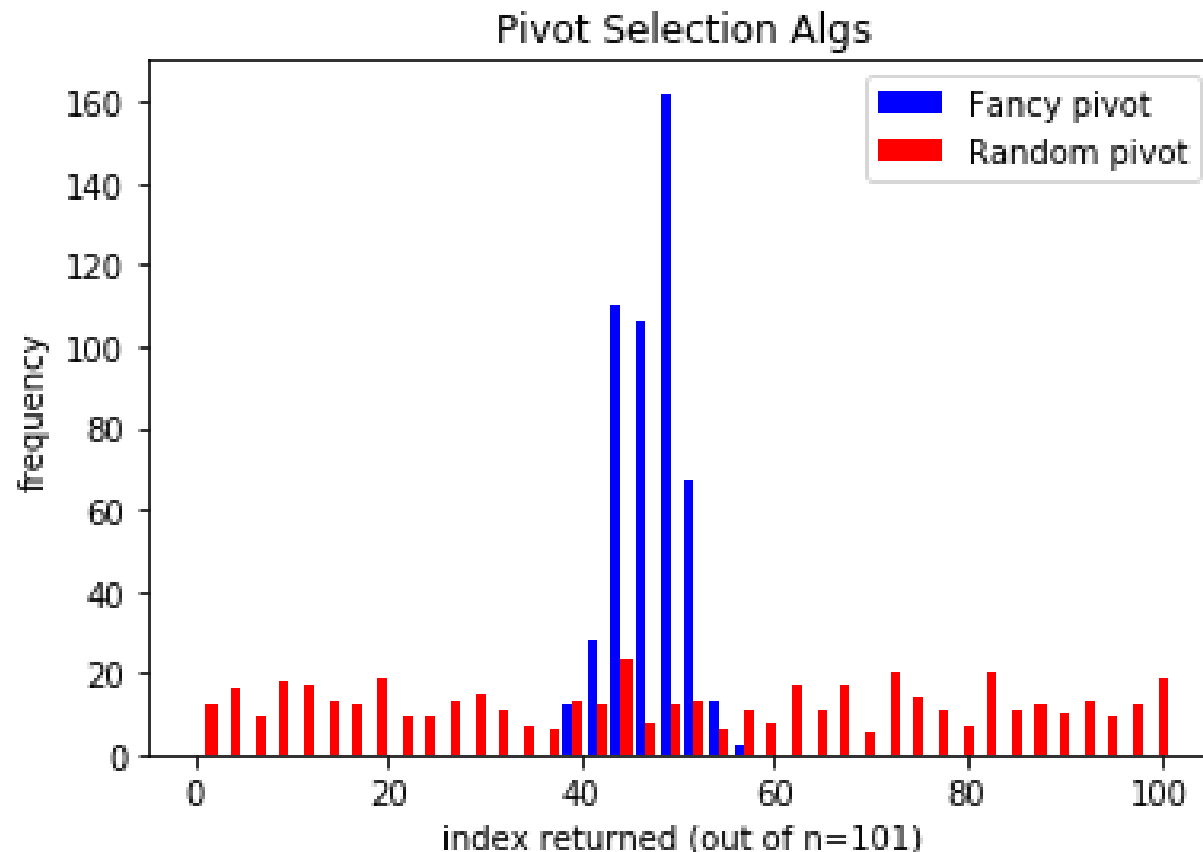
- CHOOSEPIVOT(A):
  - Split A into  $m = \lceil \frac{n}{5} \rceil$  groups, of size  $\leq 5$  each.
  - **For**  $i=1, \dots, m$ :
    - Find the median within the  $i$ 'th group, call it  $p_i$
  - $p = \text{SELECT}( [ p_1, p_2, p_3, \dots, p_m ], m/2 )$
  - **return** the index of  $p$  in A

This takes time  $O(1)$  for each group, since each group has size 5. So that's  $O(m)=O(n)$  total in the for loop.



CLAIM: this works  
divides the array *approximately* in half

- Empirically



CLAIM: this works  
divides the array *approximately* in half

- Formally, we will prove (later):

**Lemma:** If we choose the pivots like this, then

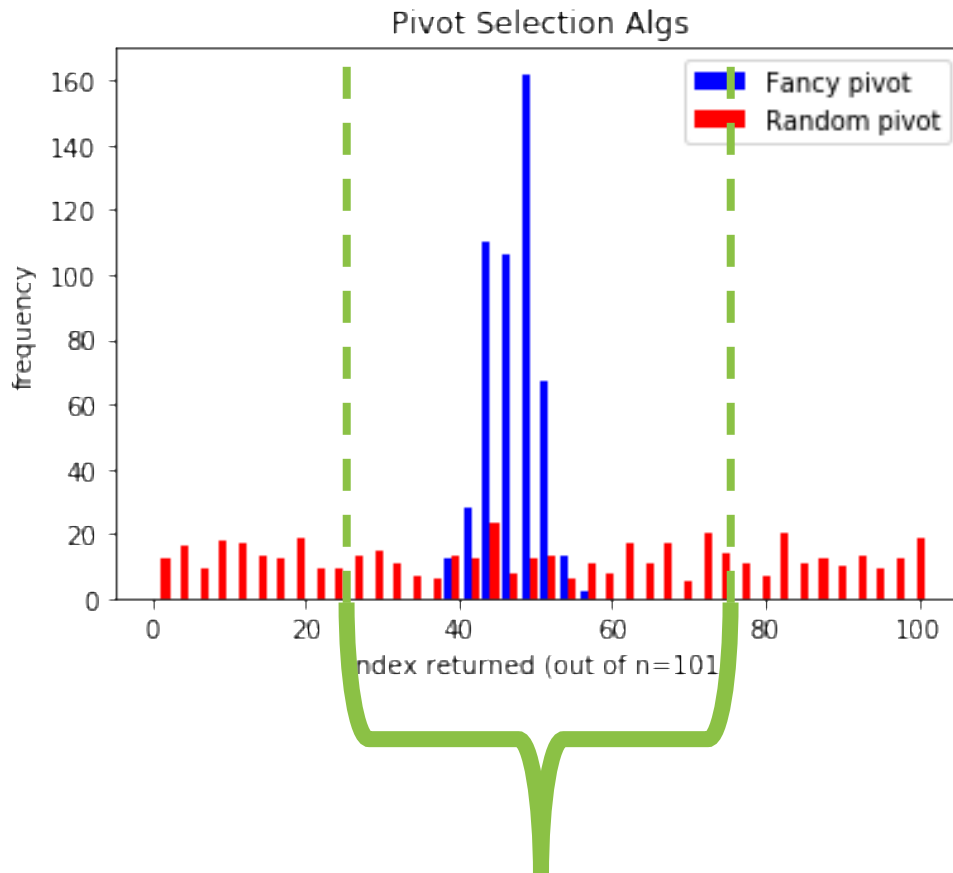
$$|L| \leq \frac{7n}{10} + 5$$

and

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

# Sanity Check

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



That's this window

Actually in practice (on randomly chosen arrays) it looks **even better!**

But this is a worst-case bound.



# How about the running time?

- Suppose the Lemma is true. (It is).

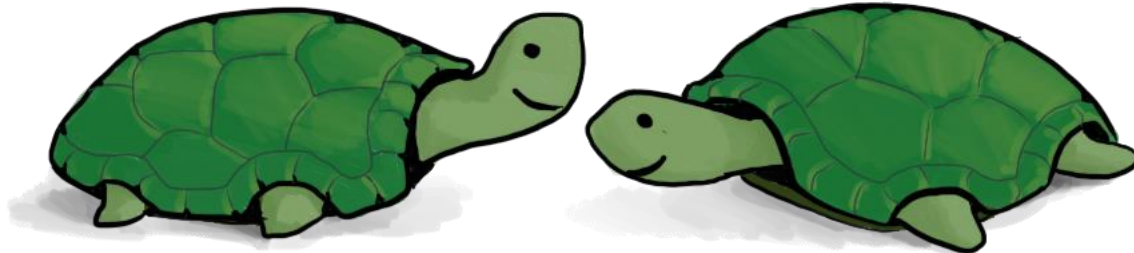
- $|L| \leq \frac{7n}{10} + 5$  and  $|R| \leq \frac{7n}{10} + 5$

- Recurrence relation:

$$T(n) \leq ?$$

Think: 2 minutes

Pair and share: 2 minutes



# How about the running time?

- Suppose the Lemma is true. (It is).

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

- Recurrence relation:

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

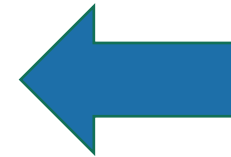
The call to CHOOSEPIVOT makes one further recursive call to SELECT on an array of size  $n/5$ .

Outside of CHOOSEPIVOT, there's at most one recursive call to SELECT on array of size  $7n/10 + 5$ .

We're going to drop the "+5" for convenience, but you can see CLRS for a more careful treatment if you're curious.

# The Plan

1. More practice with the Substitution Method.
2. k-SELECT problem
3. k-SELECT solution
  - a) The outline of the algorithm.
  - b) How to pick the pivot.
4. Return of the Substitution Method.



This sounds like a job for...

# *The Substitution Method!*

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

That's convenient!

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





# Recap of approach

- First, we figured out what the ideal pivot would be.
  - Find the median
- Then, we figured out what a **pretty good** pivot would be.
  - An approximate median
- Finally, we saw how to get our pretty good pivot!
  - Median of medians and divide and conquer!
  - Hooray!

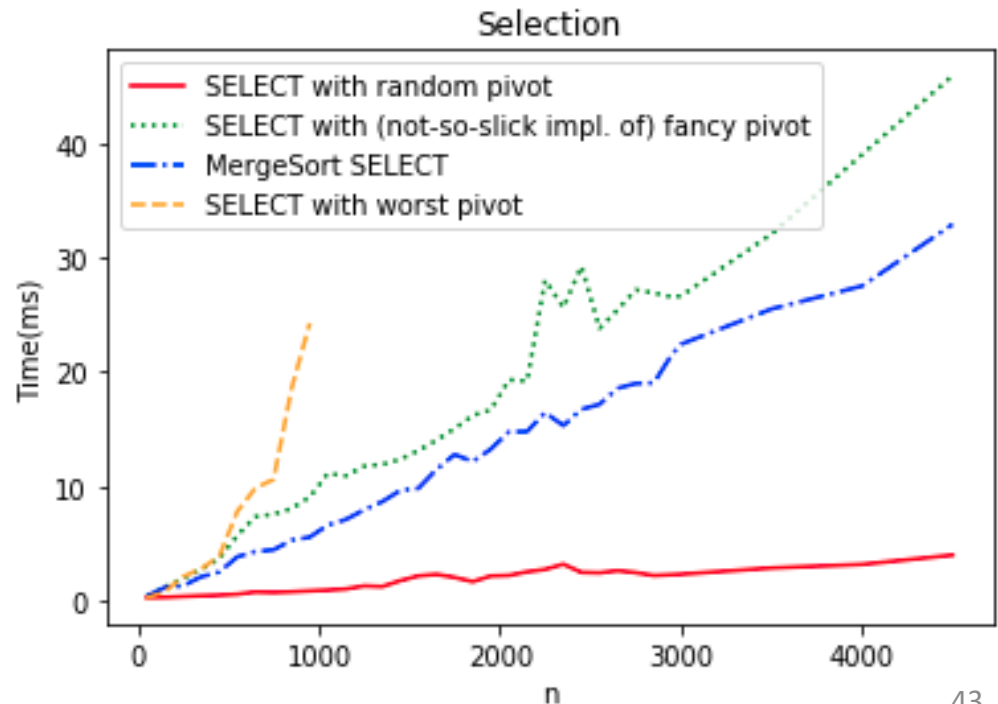
# In practice?

- With my not-very-slick implementation, our fancy version of SELECT is worse than the MergeSort-based SELECT ☹
  - But  $O(n)$  is better than  $O(n\log(n))$ ! How can that be?
  - *What's the constant in front of the  $n$  in our proof? 20? 30?*
- On **non-adversarial** inputs, random pivot choice is much better.

## Moral:

*Just pick a random pivot  
if you don't expect  
nefarious arrays.*

Optimize the implementation of  
SELECT (with the fancy pivot).  
Can you beat MergeSort?



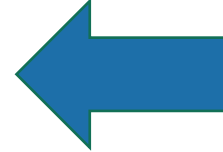
# What have we learned?

## Pending the Lemma

- It is possible to solve SELECT in time  $O(n)$ .
  - Divide and conquer!
- If you want a deterministic algorithm expect that a bad guy will be picking the list, **choose a pivot cleverly.**
  - More divide and conquer!
- If you don't expect that a bad guy will be picking the list, in practice it's better just to **pick a random pivot.**

# The Plan

1. More practice with the Substitution Method.
2. k-SELECT problem
3. k-SELECT solution
  - a) The outline of the algorithm.
  - b) How to pick the pivot.
4. Return of the Substitution Method.
5. (Optional) Proof of that Lemma.



# If time, back to the Lemma

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

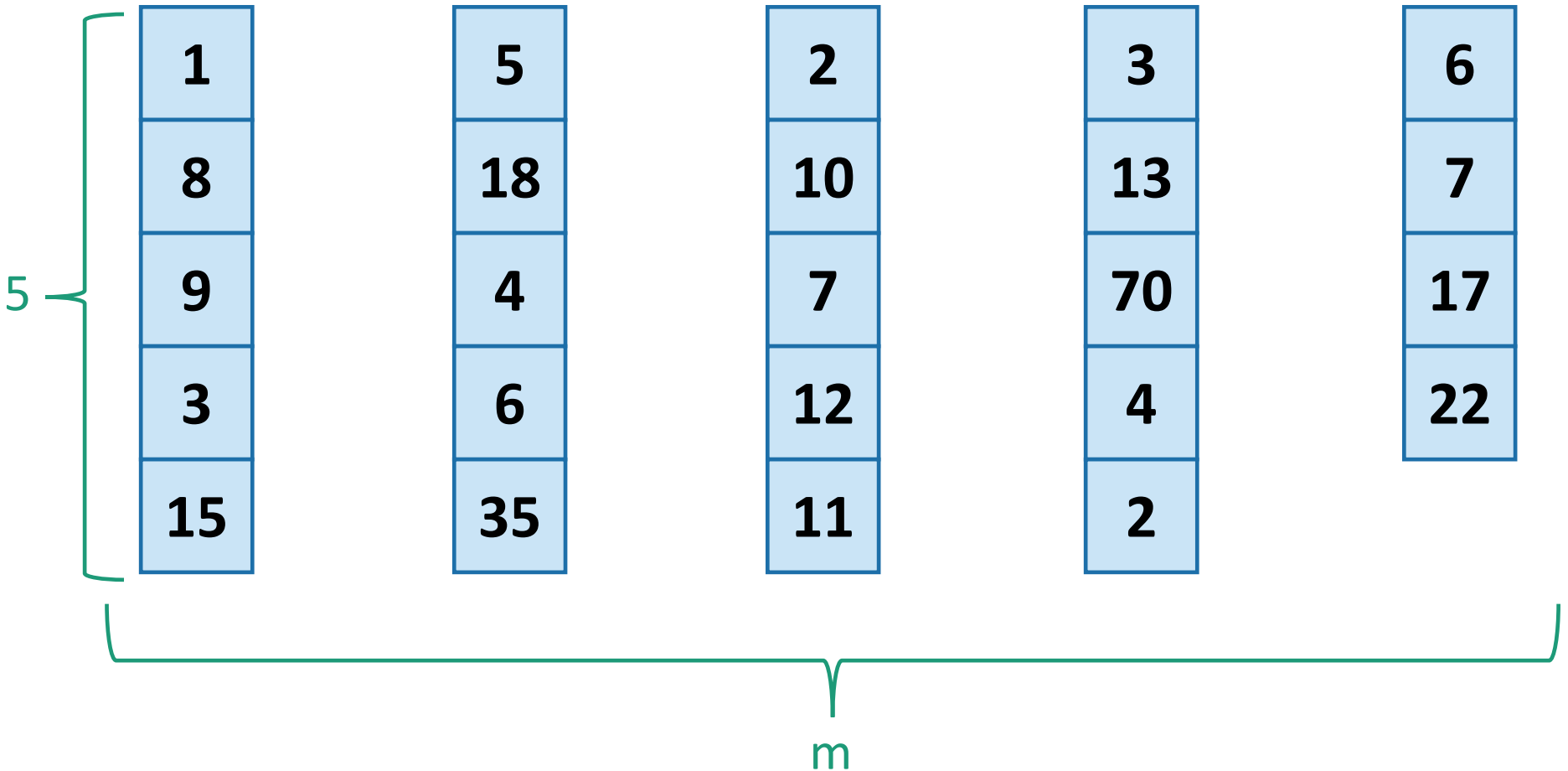
$$|L| \leq \frac{7n}{10} + 5$$

and

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

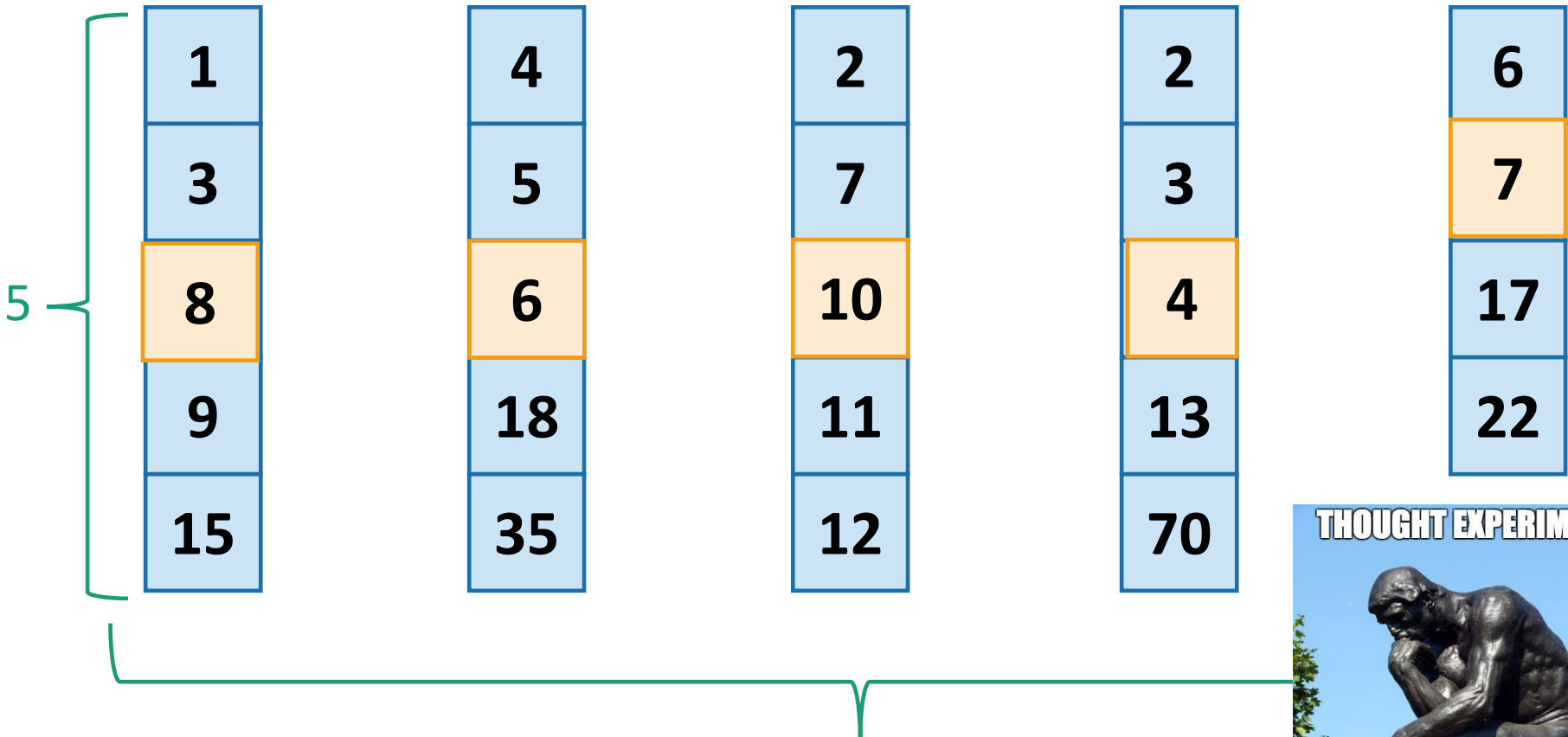
- We will see a **proof by picture**.

# Proof by picture

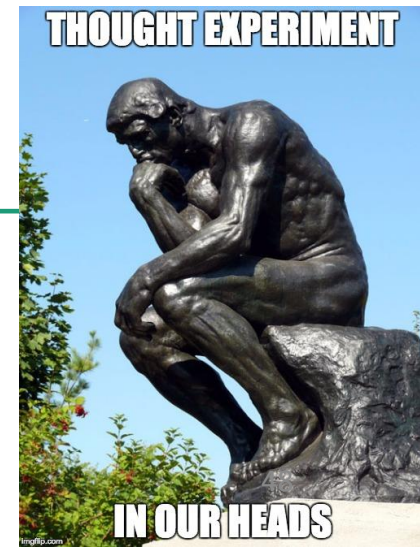


Say these are our  $m = \lceil n/5 \rceil$  sub-arrays of size at most 5.

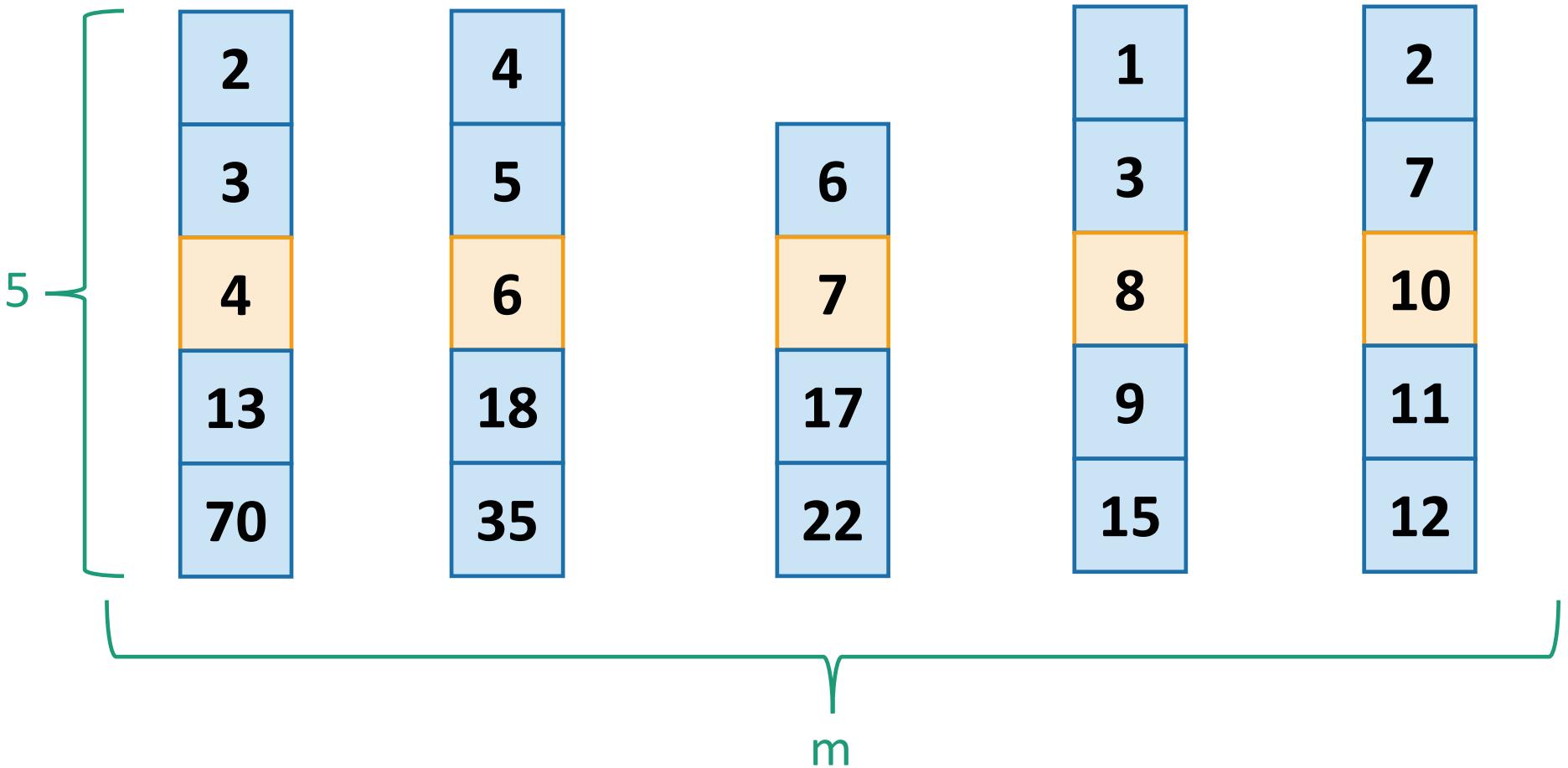
# Proof by picture



In our head, let's sort them.  
Then find medians.



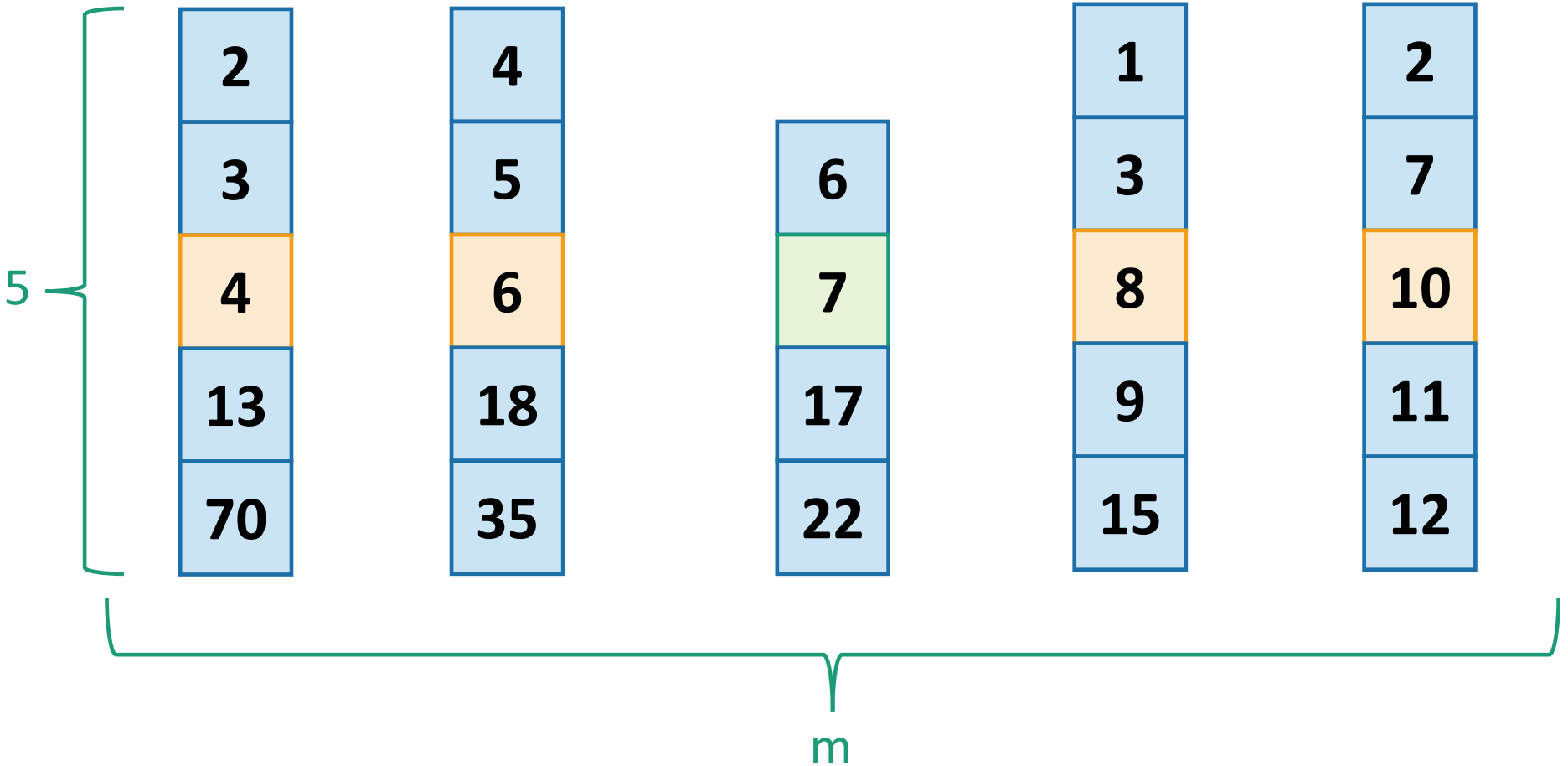
# Proof by picture



Then let's sort them by the median



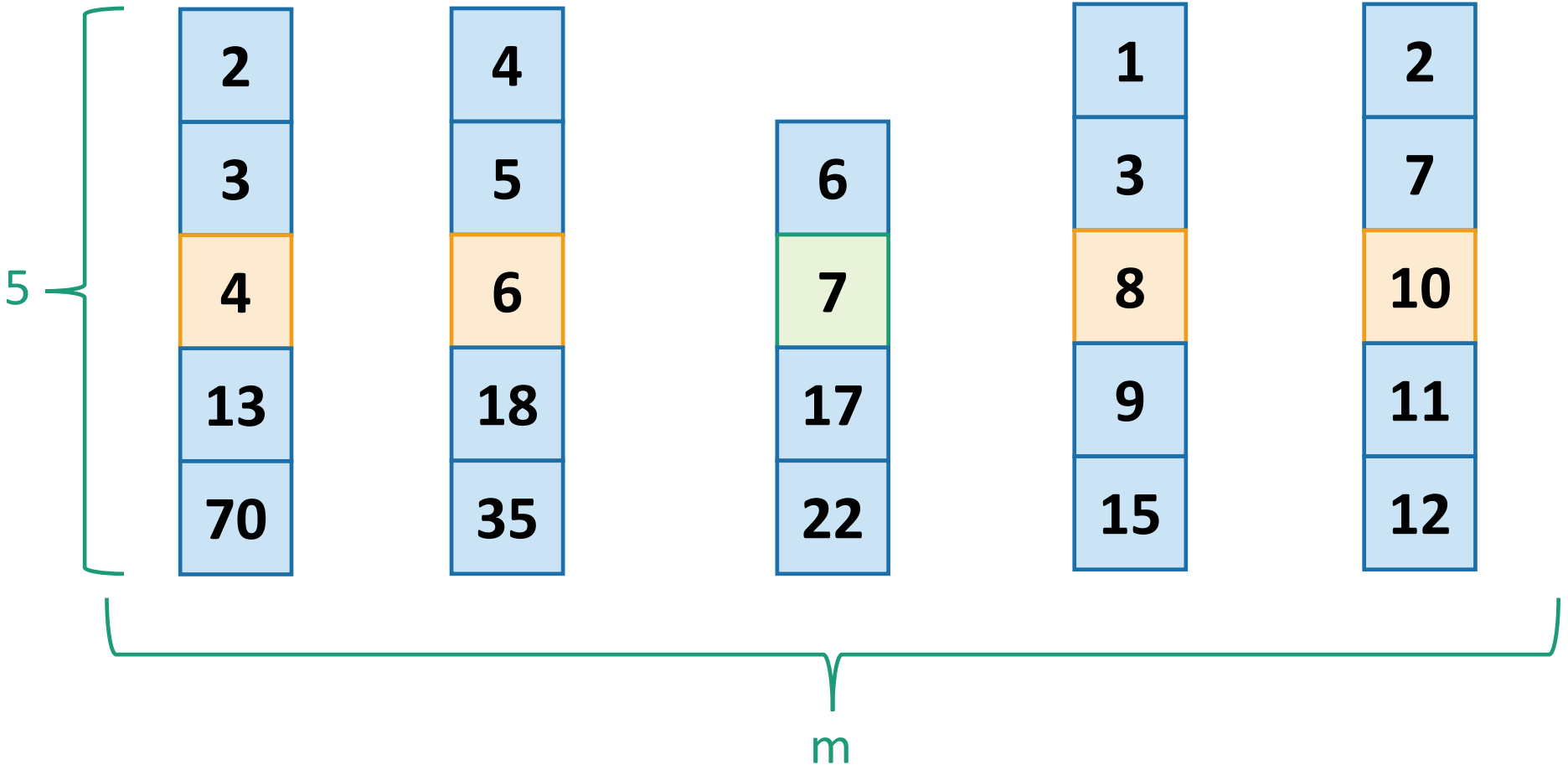
# Proof by picture



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

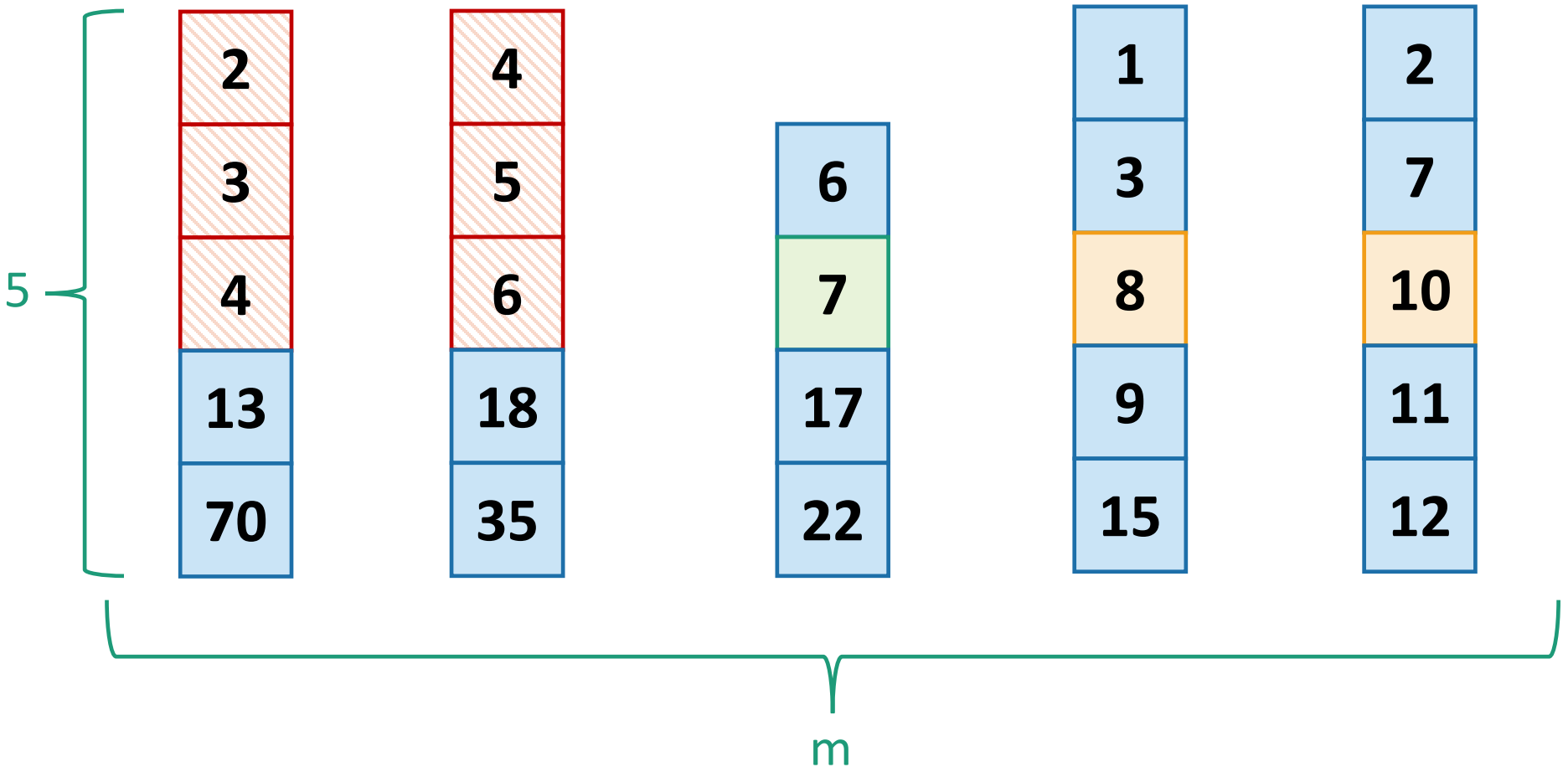
# Proof by picture

We will show that lots of elements are smaller than the pivot, hence not too many are larger than the pivot.



How many elements are SMALLER than the pivot?

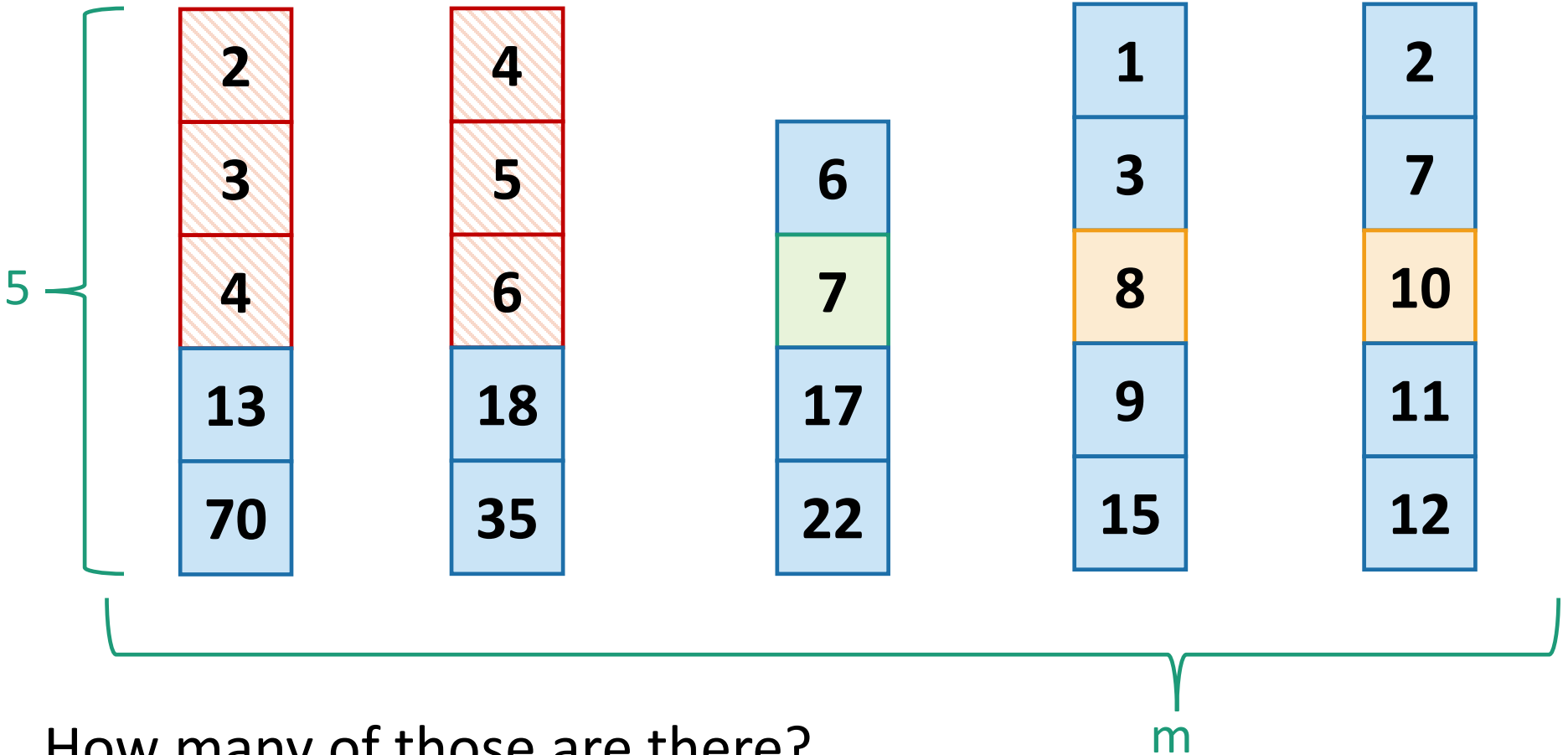
# Proof by picture



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

# Proof by picture

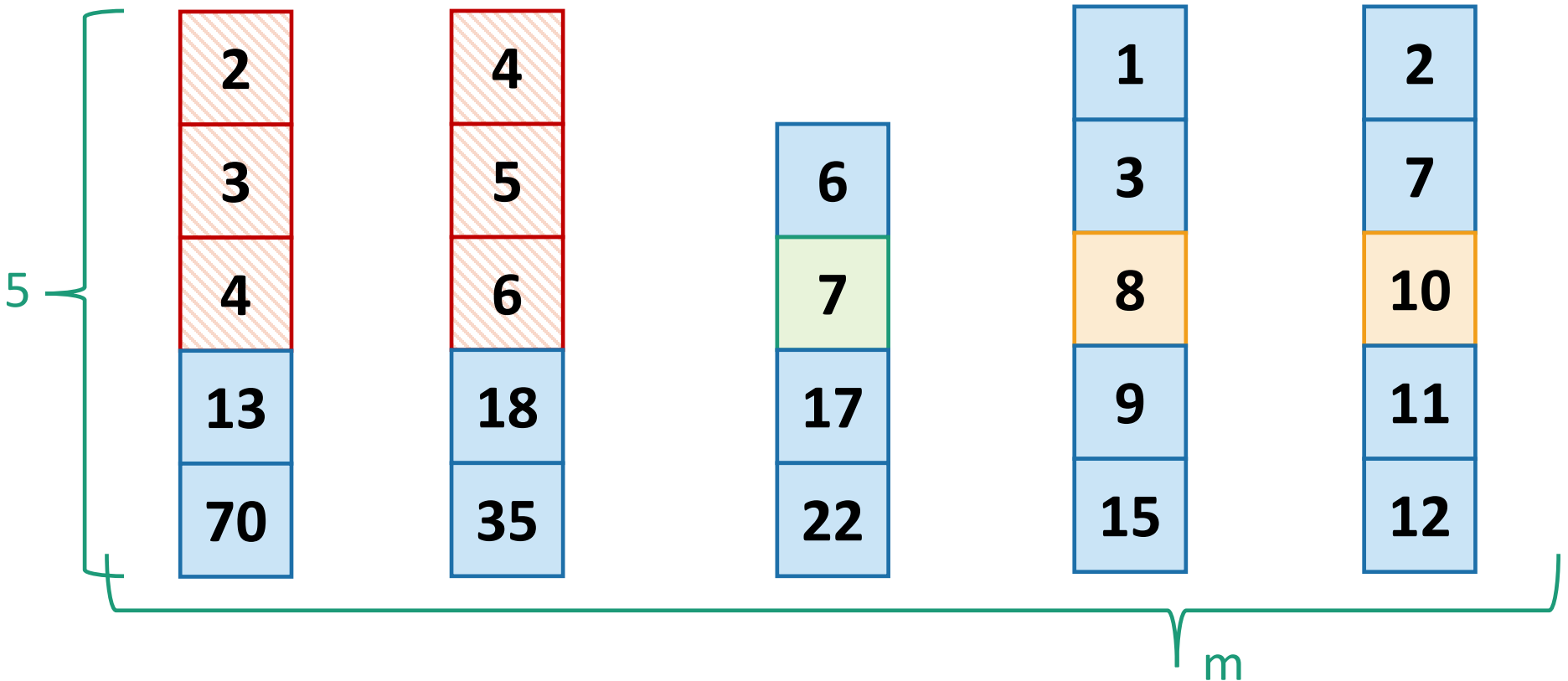
$3 \cdot \left(\left\lceil \frac{m}{2} \right\rceil - 1\right)$  of these, but then one of them could have been the “leftovers” group.



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...

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

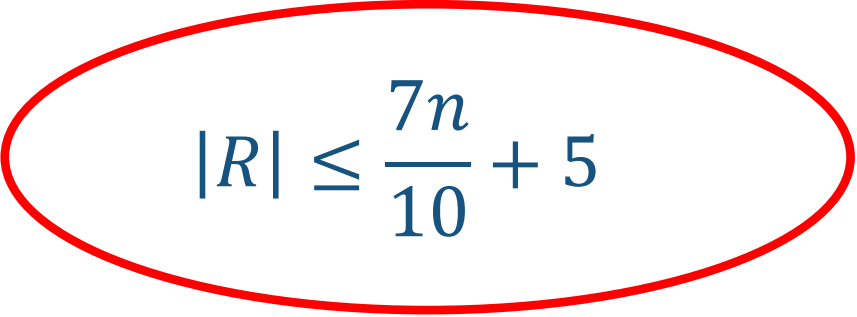
Remember  
 $m_{54} = \left\lceil \frac{n}{5} \right\rceil$

# 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

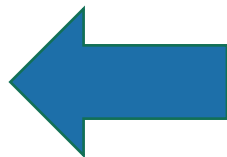

$$|R| \leq \frac{7n}{10} + 5$$

The other part is exactly the same.

# The Plan

1. More practice with the Substitution Method.
2. k-SELECT problem
3. k-SELECT solution
  - a) The outline of the algorithm.
  - b) How to pick the pivot.
4. Return of the Substitution Method.
5. (If time) Proof of that Lemma.

Recap



# Recap

- Substitution method can work when the master theorem doesn't.
- One place we needed it was for SELECT.
  - Which we can do in time  $O(n)$ !



# Next time

- QuickSort!



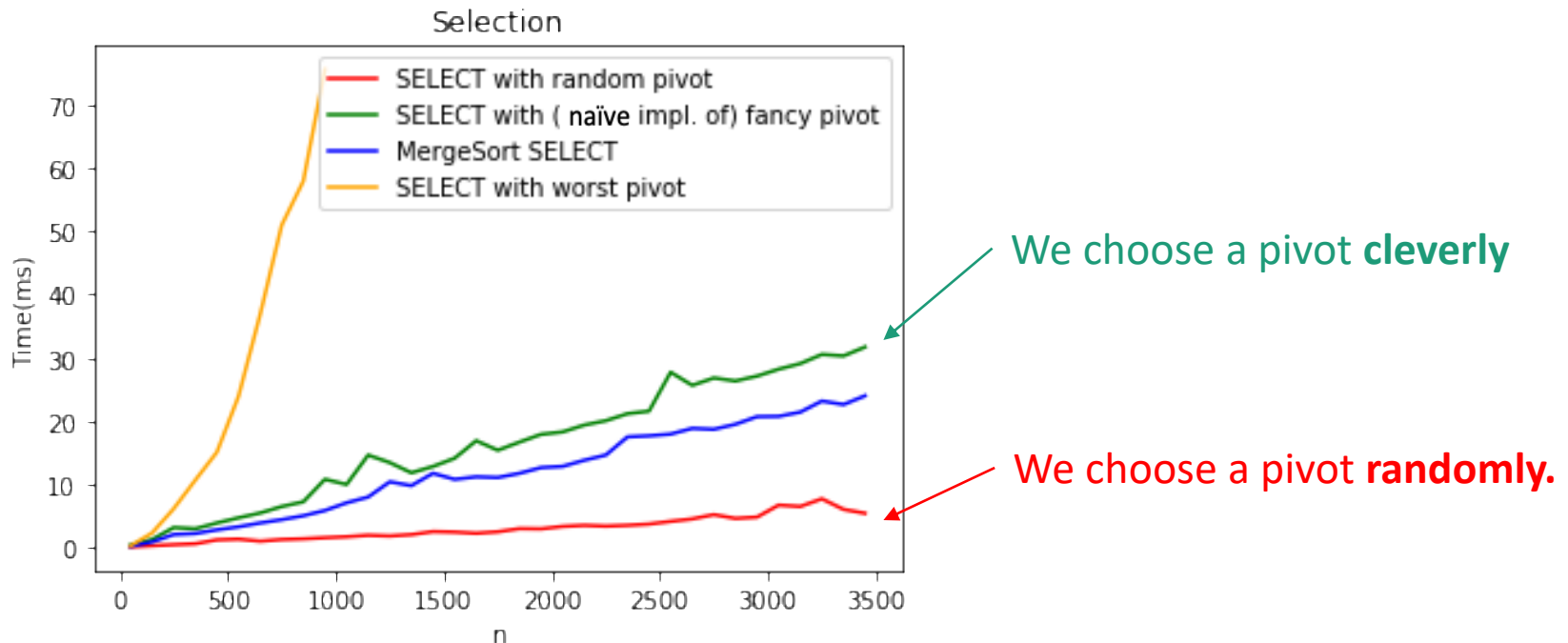
# Lecture

---

## QuickSort

# Last time

- We saw a divide-and-conquer algorithm to solve the **Select** problem in time  $O(n)$  in the worst-case.
- It all came down to picking the pivot...



# Randomized algorithms

- We make some random choices during the algorithm.
- We hope the algorithm works.
- We hope the algorithm is fast.


For today we will look at algorithms that always work and are probably fast.

e.g., **Select** with a random pivot is a randomized algorithm.

- Always works (aka, is correct).
- Probably fast.




# Today

- How do we analyze randomized algorithms?
- A few randomized algorithms for sorting.
  - **BogoSort** 
  - QuickSort
- **BogoSort** is a pedagogical tool.
- **QuickSort** is important to know. (in contrast with BogoSort...)



# Today

- How do we analyze randomized algorithms?
- A few randomized algorithms for sorting.
  - **BogoSort**
  - **QuickSort** 
- **BogoSort** is a pedagogical tool.
- **QuickSort** is important to know. (in contrast with BogoSort...)



# a better randomized algorithm:

## QuickSort

- Expected runtime  $O(n \log(n))$ .
- Worst-case runtime  $O(n^2)$ .
- In practice works great!

# Quicksort

We want to sort this array.

For the rest of the lecture, assume all elements of A are distinct.

First, pick a “pivot.”  
**Do it at random.**



Next, partition the array into  
“bigger than 5” or “less than 5”

random pivot!

This PARTITION step takes time  $O(n)$ .  
(Notice that we don't sort each half).  
[same as in SELECT]

Arrange them like so:

L = array with things smaller than A[pivot]

R = array with things larger than A[pivot]

Recurse on L and R:





# PseudoPseudoCode for what we just saw

- QuickSort(A):
  - **If**  $\text{len}(A) \leq 1$ :
    - **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)

Assume that all elements  
of A are distinct. How  
would you change this if  
that's not the case?



# Running time?

- $T(n) = T(|L|) + T(|R|) + O(n)$
- In an ideal world...
  - if the pivot splits the array exactly in half...

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

- We've seen that a bunch:

$$T(n) = O(n \log(n)).$$



# The expected running time of QuickSort is $O(n \log(n))$ .

## Proof:<sup>\*</sup>

- $E[|L|] = E[|R|] = \frac{n-1}{2}$ .
  - The expected number of items on each side of the pivot is half of the things.

# Aside

why is  $E[|L|] = \frac{n-1}{2}$  ?

- $E[|L|] = E[|R|]$ 
  - by symmetry
- $E[|L| + |R|] = n - 1$ 
  - because L and R make up everything except the pivot.
- $E[|L|] + E[|R|] = n - 1$ 
  - By linearity of expectation
- $2E[|L|] = n - 1$ 
  - Plugging in the first bullet point.
- $E[|L|] = \frac{n-1}{2}$ 
  - Solving for  $E[|L|]$ .

# The expected running time of QuickSort is $O(n \log(n))$ .

## Proof:<sup>\*</sup>

- $E[|L|] = E[|R|] = \frac{n-1}{2}$ .
  - The expected number of items on each side of the pivot is half of the things.
- If that occurs, the running time is  $T(n) = O(n \log(n))$ .
  - Since the relevant recurrence relation is  $T(n) = 2T\left(\frac{n-1}{2}\right) + O(n)$
- Therefore, the expected running time is  $O(n \log(n))$ .



# Red flag

- **Slow** Sort(A):
  - If  $\text{len}(A) \leq 1$ :
  - return

We can use the same argument to prove something false.

- **Pick the pivot x to be either max(A) or min(A), randomly**
  - \\ We can find the max and min in  $O(n)$  time

- 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)

- **Slow** Sort(L)

- **Slow** Sort(R)

- Same recurrence relation:

$$T(n) = T(|L|) + T(|R|) + O(n)$$

- We still have  $E[|L|] = E[|R|] = \frac{n-1}{2}$
- But now, one of  $|L|$  or  $|R|$  is always  $n-1$ .
- You check: Running time is  $\Theta(n^2)$ , with probability 1.

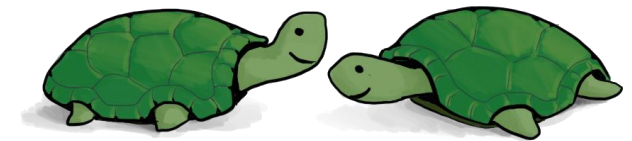
# The expected running time of SlowSort is $O(n \log(n))$ .

What's wrong???

2 minutes: think

1 minute: pair and share

## Proof:<sup>\*</sup>



- $E[|L|] = E[|R|] = \frac{n-1}{2}$ .
  - The expected number of items on each side of the pivot is half of the things.
- If that occurs, the running time is  $T(n) = O(n \log(n))$ .
  - Since the relevant recurrence relation is  $T(n) = 2T\left(\frac{n-1}{2}\right) + O(n)$
- Therefore, the expected running time is  $O(n \log(n))$ .

# What's wrong?

- $E[|L|] = E[|R|] = \frac{n-1}{2}$ .
  - The expected number of items on each side of the pivot is half of the things.
- If that occurs, the running time is  $T(n) = O(n \log(n))$ .
  - Since the relevant recurrence relation is  $T(n) = 2T\left(\frac{n-1}{2}\right) + O(n)$
- Therefore, the expected running time is  $O(n \log(n))$ .

***That's not how  
expectations work!***



- The running time in the “expected” situation is not the same as the expected running time.
- Sort of like how  $E[X^2]$  is not the same as  $(E[X])^2$



# What have we learned?

- The expected running time of QuickSort is  $O(n \log(n))$

# Worst-case running time

- Suppose that an adversary is choosing the “random” pivots for you.
- Then the running time might be  $O(n^2)$ 
  - Eg, they’d choose to implement SlowSort
  - In practice, this doesn’t usually happen.



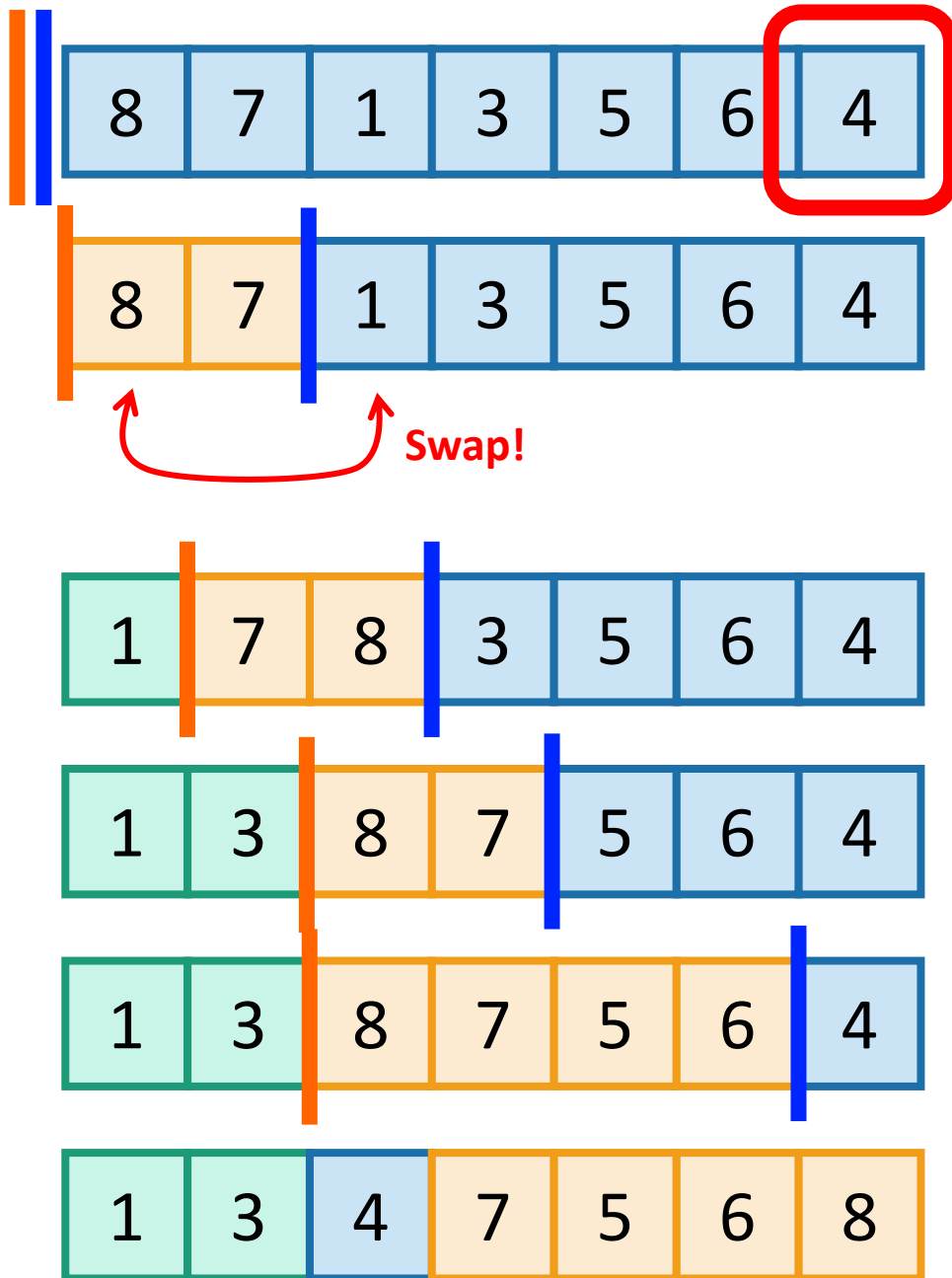
# How should we implement this?

- Our pseudocode is easy to understand and analyze, but is not a good way to implement this algorithm.

- QuickSort(A):
  - If  $\text{len}(A) \leq 1$ :
    - **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)



- Instead, implement it **in-place** (without separate L and R)

# A better way to do Partition




## Pivot

Choose it randomly, then swap it with the last one, so it's at the end.

Initialize  and 

Step  forward.

When  sees something smaller than the pivot, **swap** the things ahead of the bars and increment both bars.

Repeat till the end, then put the pivot in the right place.

# QuickSort vs MergeSort

\*What if you want  $O(n \log(n))$  worst-case runtime and stability? Check out “Block Sort” on Wikipedia!

	QuickSort (random pivot)	MergeSort (deterministic)
Running time	<ul style="list-style-type: none"><li>Worst-case: <math>O(n^2)</math></li><li>Expected: <math>O(n \log(n))</math></li></ul>	Worst-case: $O(n \log(n))$
Used by	<ul style="list-style-type: none"><li>Java for primitive types</li><li>C qsort</li><li>Unix</li><li>g++</li></ul>	<ul style="list-style-type: none"><li>Java for objects</li><li>Perl</li></ul>
In-Place? (With $O(\log(n))$ extra memory)	Yes, pretty easily	Not easily* if you want to maintain both stability and runtime. (But pretty easily if you can sacrifice runtime).
Stable?	No	Yes
Other Pros	Good cache locality if implemented for arrays	Merge step is really efficient with linked lists

Understand this

These are just for fun.  
(Not on exam).

# Today

- How do we analyze randomized algorithms?
- A few randomized algorithms for sorting.
  - **BogoSort**
  - **QuickSort**
- **BogoSort** is a pedagogical tool.
- **QuickSort** is important to know. (in contrast with BogoSort...)



Recap



# Recap

- How do we measure the runtime of a **randomized algorithm**?

- Expected runtime
- Worst-case runtime



- **QuickSort** (with a random pivot) is a randomized sorting algorithm.
  - In many situations, QuickSort is nicer than MergeSort.
  - In many situations, MergeSort is nicer than QuickSort.

Code up QuickSort and MergeSort in a few different languages, with a few different implementations of lists A (array vs linked list, etc). What's faster?

(This is an exercise best done in C where you have a bit more control than in Python).

