CSE100: Design and Analysis of Algorithms
Lecture 07 – Selection and Median (cont.)

Feb 8th 2022

More Recursion, Beyond the Master Theorem

Solving Recurrence Relations (review)

- 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)

What have we learned? (review)

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

- Step 1: generate a guess
  - Throw the kitchen sink at it.
- Step 2: try to prove that your guess is correct
  - You may have to leave some constants unspecified till the end – then see what they need to be for the proof to work!!
- Step 3: profit
  - Pretend you didn't do Steps 1 and 2 and write down a nice proof.

The k-SELECT problem (review)

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

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

For today, assume all arrays have distinct elements.

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)

Being sloppy about floors and ceilings!

Note that the definition of Select is 1-indexed…

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)

9	8	3	6	1	4	2

Idea: divide and conquer!

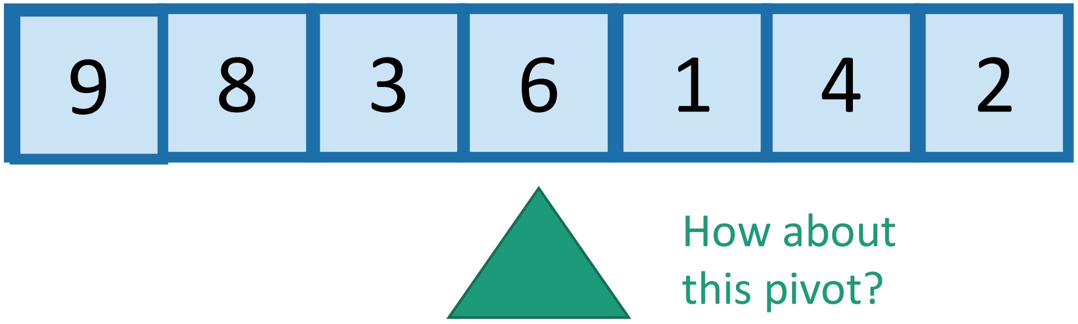
Say we want to find SELECT(A, k)

9	8	3	6	1	4	2

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

Idea: divide and conquer!

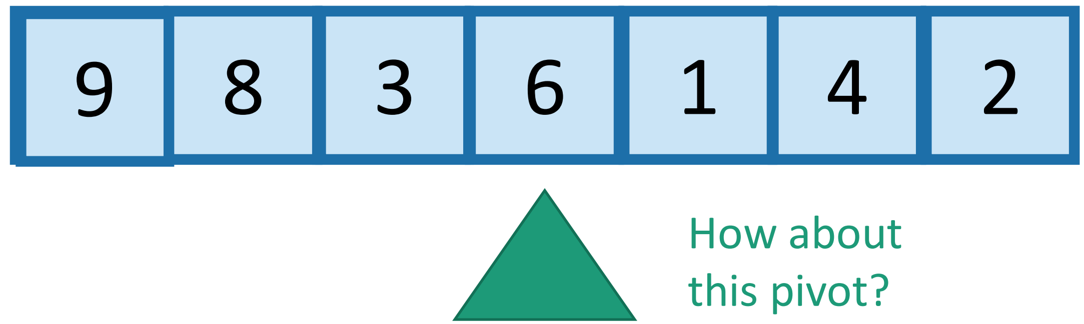
Say we want to find SELECT(A, k)



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

Idea: divide and conquer!

Say we want to find SELECT(A, k)

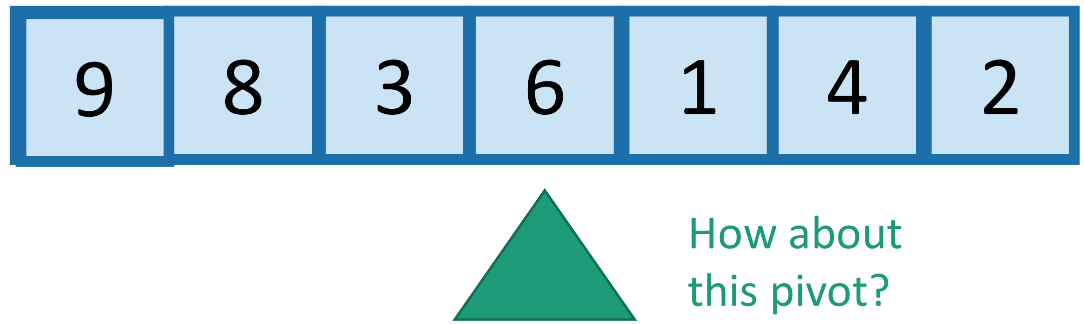


First, pick a "pivot." We'll see how to do this later.

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

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.

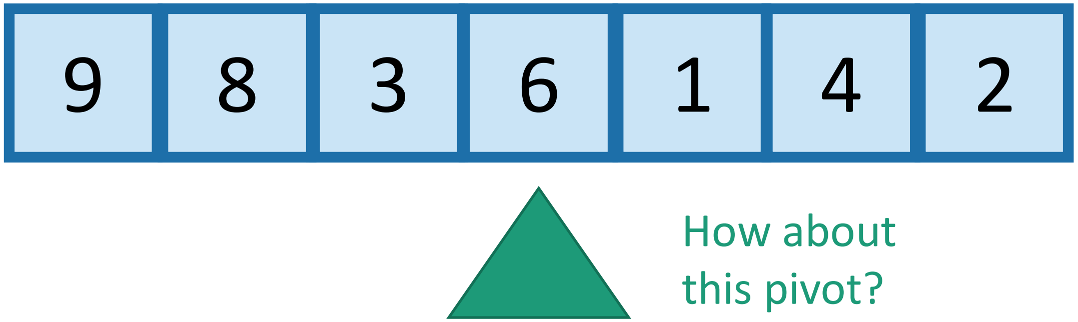
Next, partition the array into "bigger than 6" or "less than 6"

L = array with things smaller than A[pivot]

R = array with things larger than A[pivot]

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.

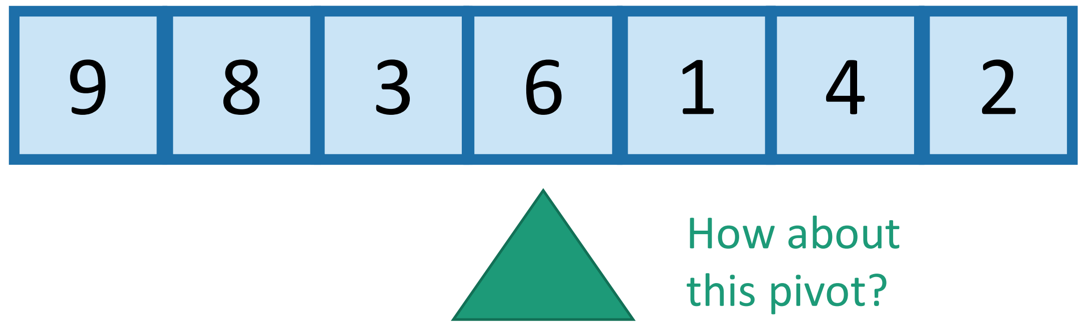
Next, partition the array into "bigger than 6" or "less than 6"

L = array with things smaller than A[pivot]

R = array with things larger than A[pivot]

Idea: divide and conquer!

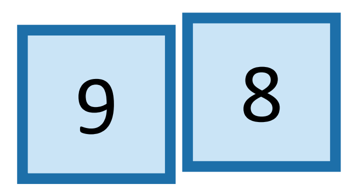
Say we want to find SELECT(A, k)



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"

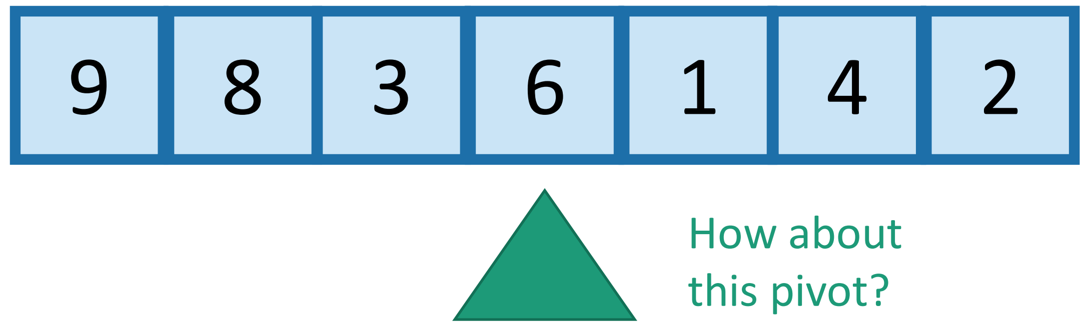


L = array with things smaller than A[pivot]

R = array with things larger than A[pivot]

Idea: divide and conquer!

Say we want to find SELECT(A, k)

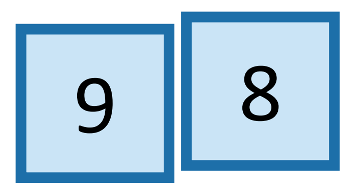


How about this pivot?

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

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

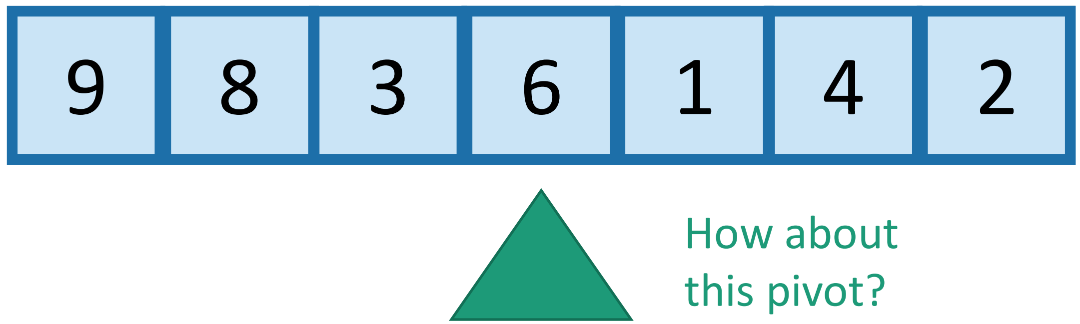
L = array with things smaller than A[pivot]



R = array with things larger than A[pivot]

Idea: divide and conquer!

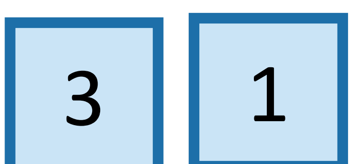
Say we want to find SELECT(A, k)



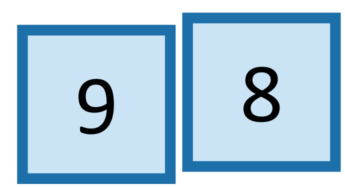
How about this pivot?

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

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



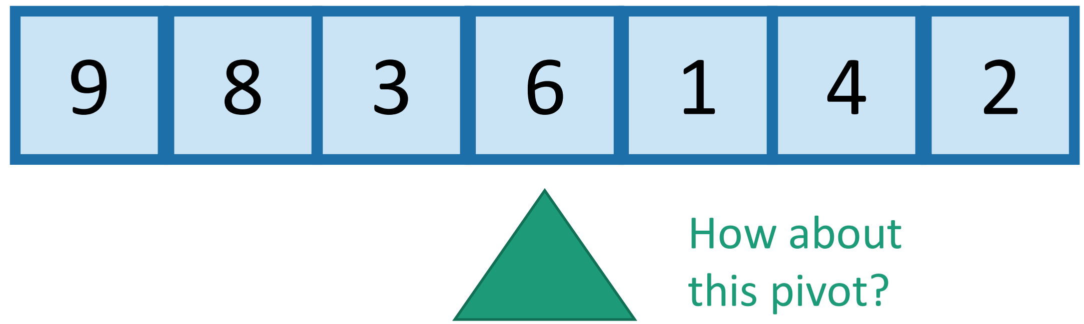
L = array with things smaller than A[pivot]



R = array with things larger than A[pivot]

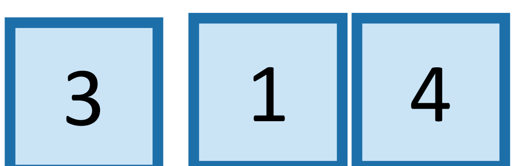
Idea: divide and conquer!

Say we want to find SELECT(A, k)

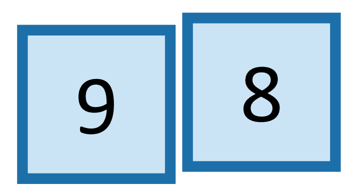


First, pick a "pivot." We'll see how to do this later.

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



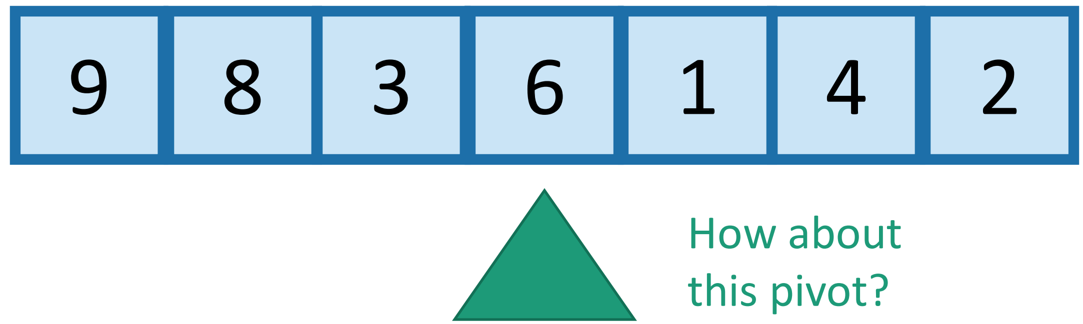
L = array with things smaller than A[pivot]



R = array with things larger than A[pivot]

Idea: divide and conquer!

Say we want to find SELECT(A, k)

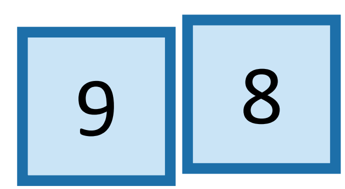


How about this pivot?

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

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

L = array with things smaller than A[pivot]



R = array with things larger than A[pivot]

Idea: divide and conquer!

Say we want to find SELECT(A, k)

9	8	3	6	1	4	2

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).

3	1	4	2

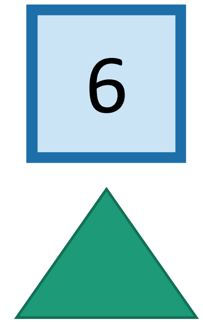
L = array with things smaller than A[pivot]

9	8

R = array with things larger than A[pivot]

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).

3	1	4	2

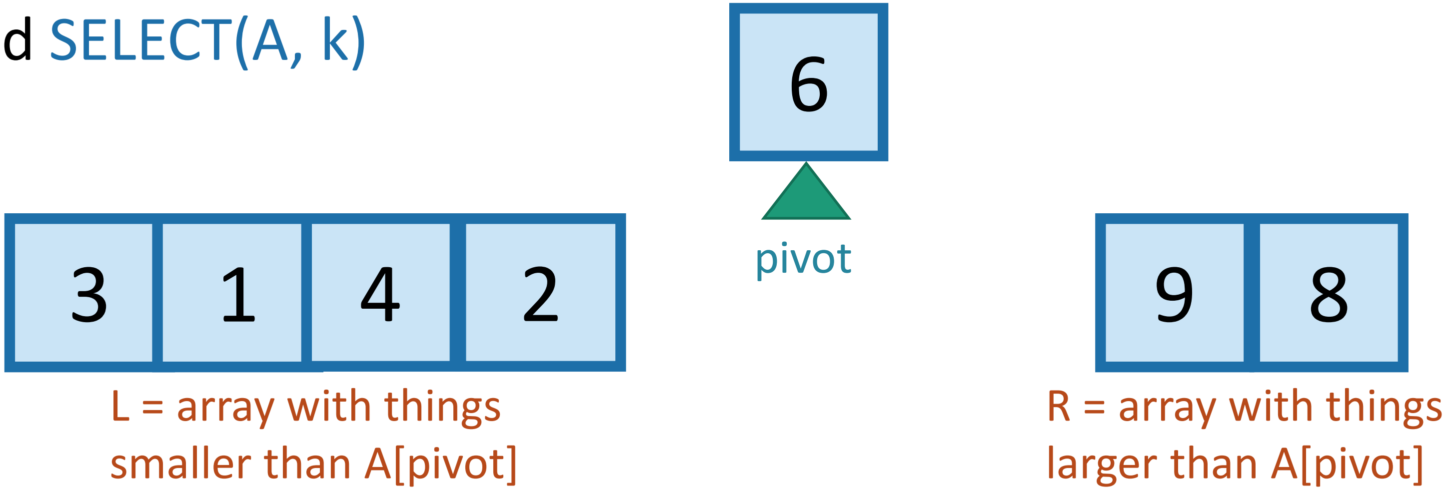
L = array with things smaller than A[pivot]

9	8

R = array with things larger than A[pivot]

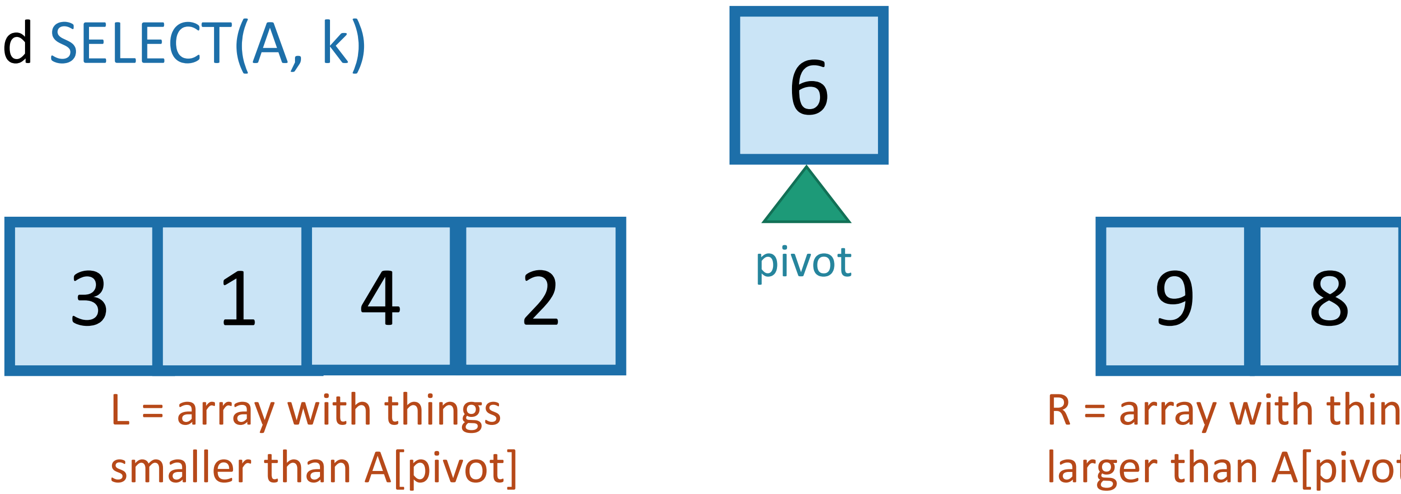
Idea continued…

Say we want to find SELECT(A, k)



Idea continued…

Say we want to find SELECT(A, k)



- 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

- SELECT(A, 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))

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

Base Case: If the 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

Correctness

There seems to be some whitespace not yet used.

That's better.

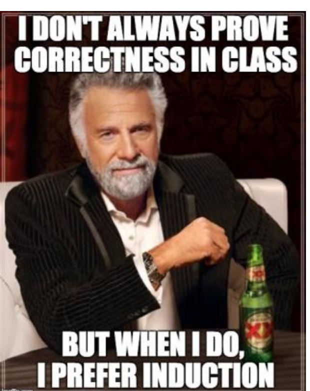
- Recursion invariant:

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.

- 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)

Note: something like this is totally acceptable on your exams (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.

- Conclusion ("Termination")
- By induction, the recursion invariant holds for n + 1, which means that SELECT(A,k) is correct.

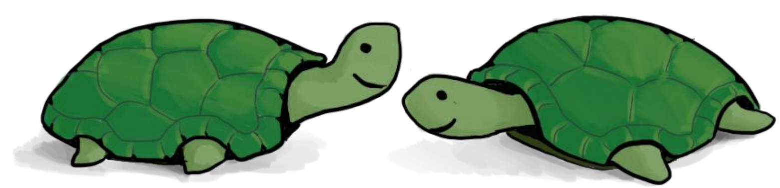
What is the running time?

Assuming we pick the pivot in time O(n)…

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

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

What would be a "good" pivot?
What would be a "bad" pivot?



Think-Pair-Share Terrapins

The best way would be to always pick the pivot so that len(L) = k-1. But say we don't have control over k, just over how we pick the pivot.

The ideal pivot

- We split the input exactly in half:
  - len(L) = len(R) = (n-1)/2

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

Jedi master Yoda

- Let's pretend that's the case and use the **Master Theorem**!
  - $T(n) \leq T\left(\frac{n}{2}\right) + O(n)$
  - So, a = 1, b = 2, d = 1
  - $T(n) \leq O\left(n^d\right) = O(n)$

<u>Note</u>: This is a rhetorical point for intuition in lecture. It is **NOT OKAY** as a final solution on your exam.

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

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

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
- $T(n) \leq T\left(\cancel{\frac{n}{2}}\right) + O(n)$
- $T(n) = O(n)$
- That would be great!

(we can even assume something a little weaker)

- SELECT(A, 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)

Recall the Master Theorem

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

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

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

In our case:

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

Lucky the Lackadaisical Lemur

How about runtime?

- Let's try a recurrence relation...
- $T(n) = \begin{cases} T(len(L)) + O(n) & len(L) < k - 1 \\ T(len(R)) + O(n) & len(L) > k - 1 \\ O(n) & len(L) = k - 1 \end{cases}$
- What is len(L), len(R)?
- Let's **pretend** that len(L) is about ~~n/2~~. 7n/10
- $T(n) \le T\left(\cancel{\frac{n}{2}}\right) + O(n)$
- $T(n) = O(n)$
- That would be great! !!!!!

- SELECT(A, 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)

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~~.

Can we get away with n-1?

- $T(n) \leq T\left(\cancel{\frac{n}{2}}\right) + O(n)$

- SELECT(A, 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)

Recall the Master Theorem

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

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

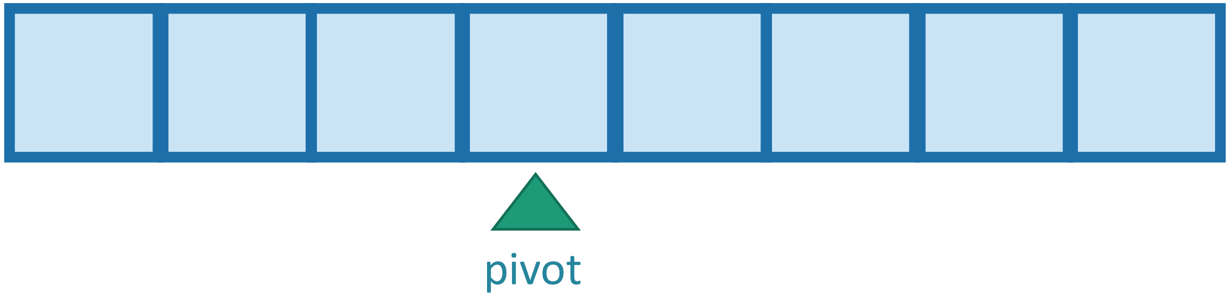
How about runtime?

- 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~~.
- $T(n) \leq T\left(\cancel{\frac{n}{2}}\right) + O(n)$
- $T(n) = O\left(n^2\right)$
- Not good enough ☹!

Can we get away with n-1?

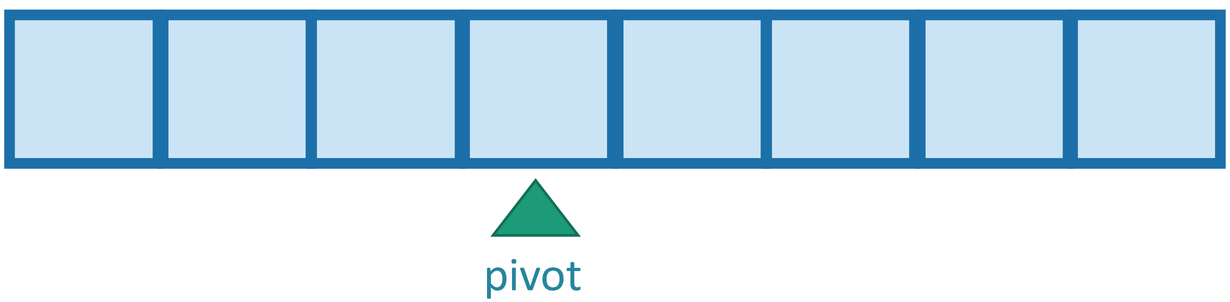
- SELECT(A, k):
    - **If** len(A) <= 50:
        - A = MergeSort(A)
        - Return A[k]
    - p = CHOOSEPIVOT(A)
    - L, A[p], R = PARTITION(A,p)
    - **If** len(L) = k - 1:
        - **Return** A[p]
    - **Else If** len(L) > k - 1:
        - **Return** SELECT(L, k)
    - **Else if** len(L) <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

The worst pivot



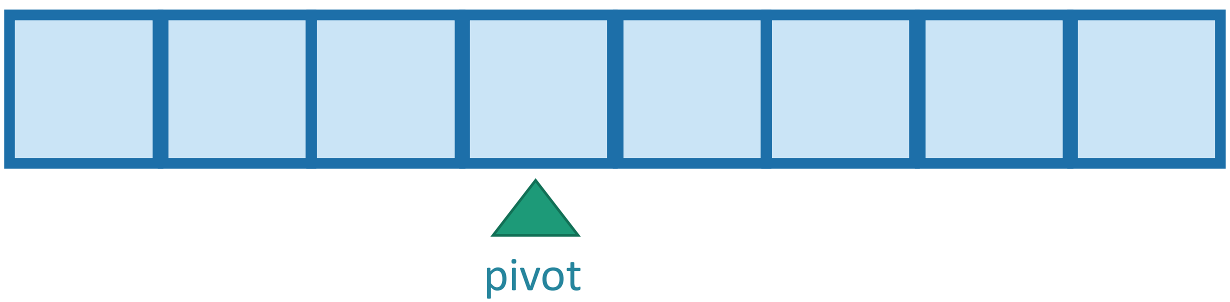
The worst pivot

- Say our choice of pivot doesn't depend on A.



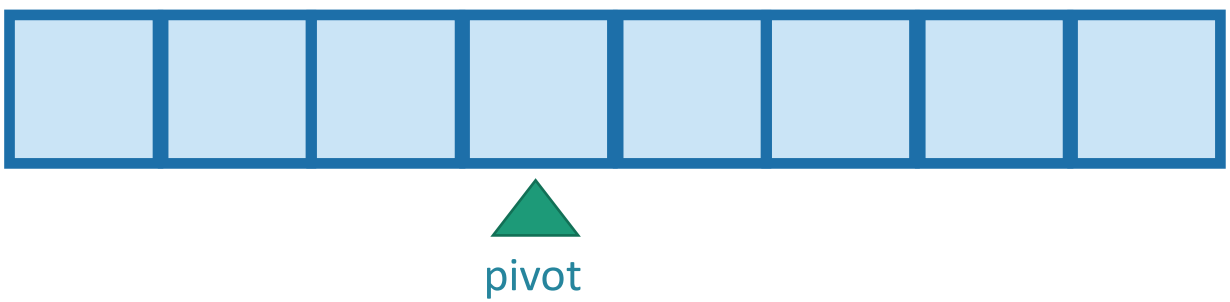
The worst pivot

- 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.



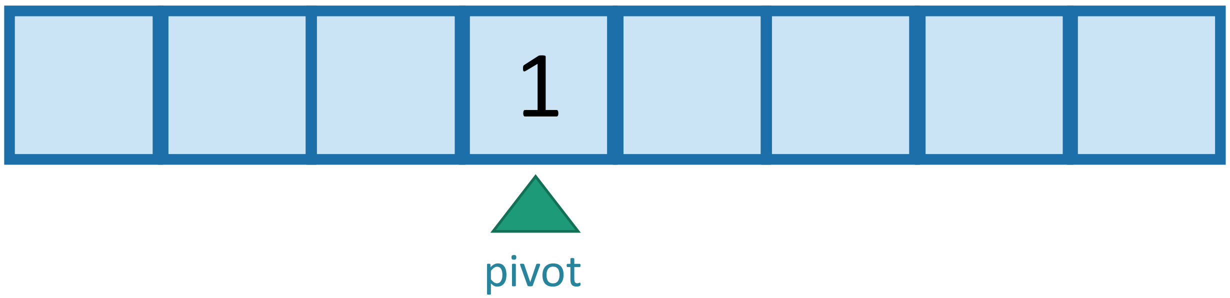
The worst pivot

- 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.



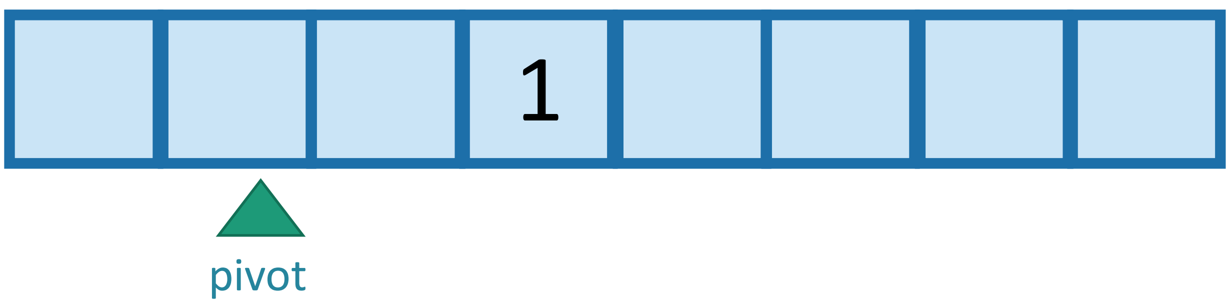
The worst pivot

- 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.



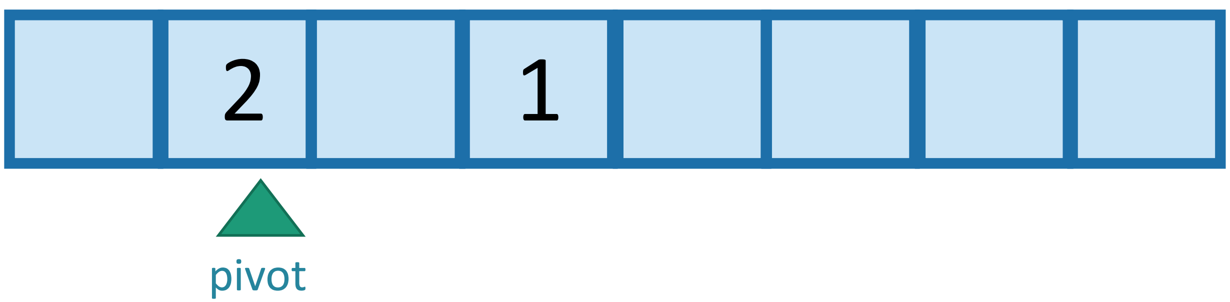
The worst pivot

- 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.



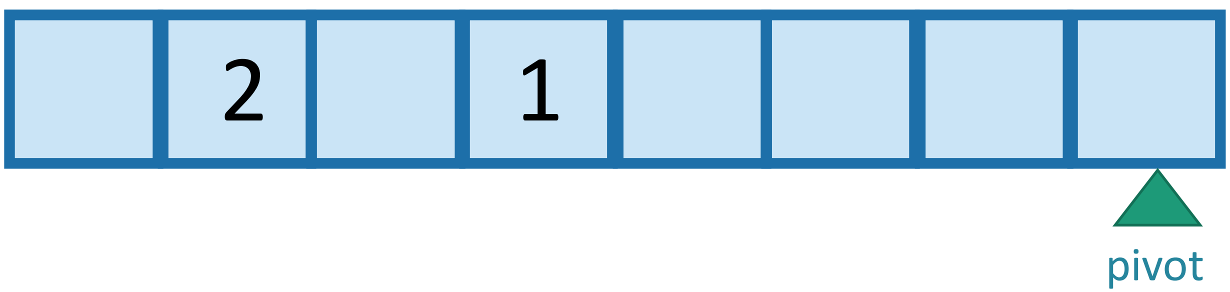
The worst pivot

- 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.



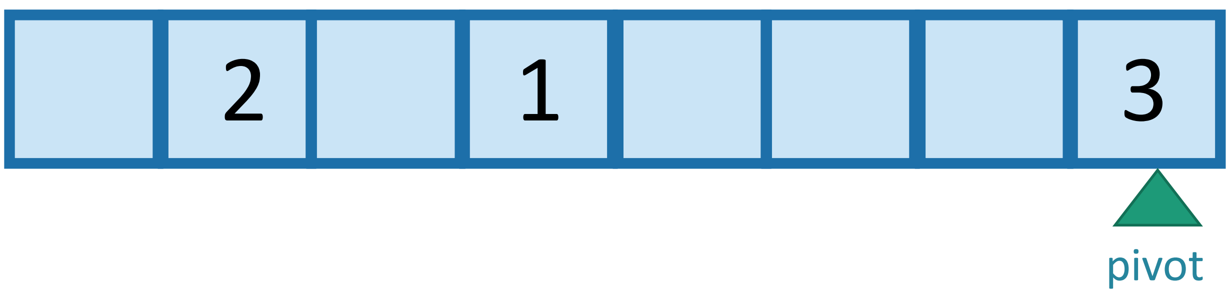
The worst pivot

- 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.

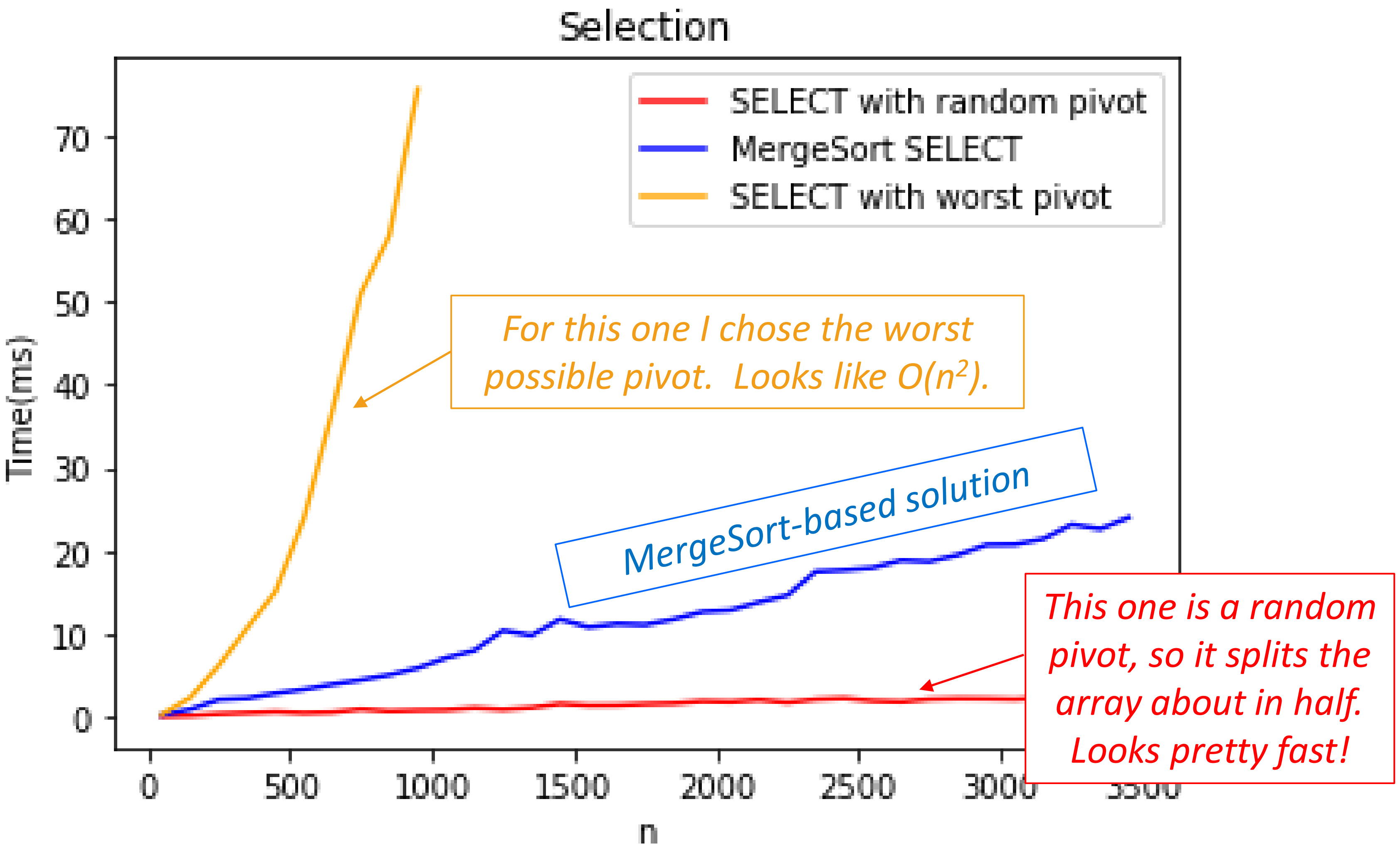


The worst pivot

- 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!
- (More on this later with randomized algos)

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!

Moral of this extremely shady logic

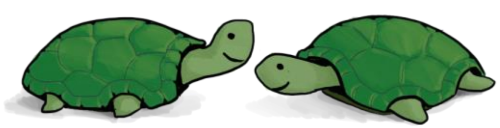
- If we can pick a pivot so that L and R **somewhat** balanced (even like 7n/10), then we're doing great. Otherwise, no good.
- **Try 1: Let's pick the pivot to be the median!**
  - Then L and R are always n/2. (or $\left\lfloor\frac{n}{2}\right\rfloor$ or $\left\lceil\frac{n}{2}\right\rceil$).
- **Problem:** 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.

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 < len(L) < 7n/10
  - 3n/10 < len(R) < 7n/10
- If we could do that (let's say, in time O(n)), the **Master Theorem** would say:
  - $T(n) \leq T\left(\frac{7n}{10}\right) + O(n)$



Think-Pair-Share Terrapins!

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

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

A good enough pivot

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

Lucky the lackadaisical lemur

- We split the input not quite in half:
  - 3n/10 < len(L) < 7n/10
  - 3n/10 < 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
  - $T(n) \leq O\left(n^d\right) = O(n)$

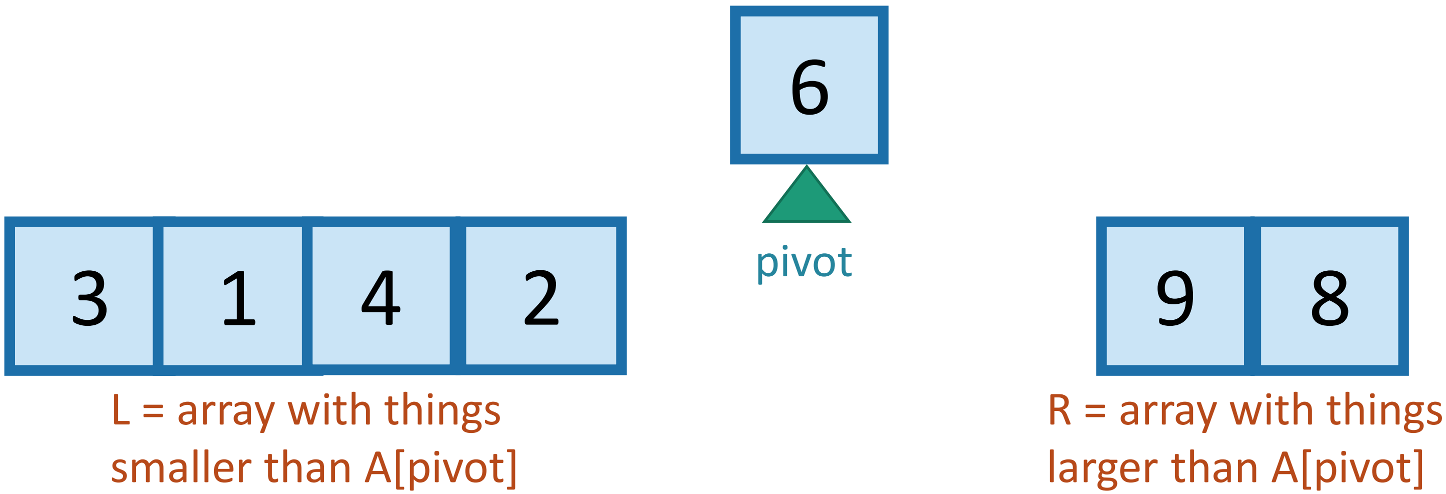
STILL GOOD!

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

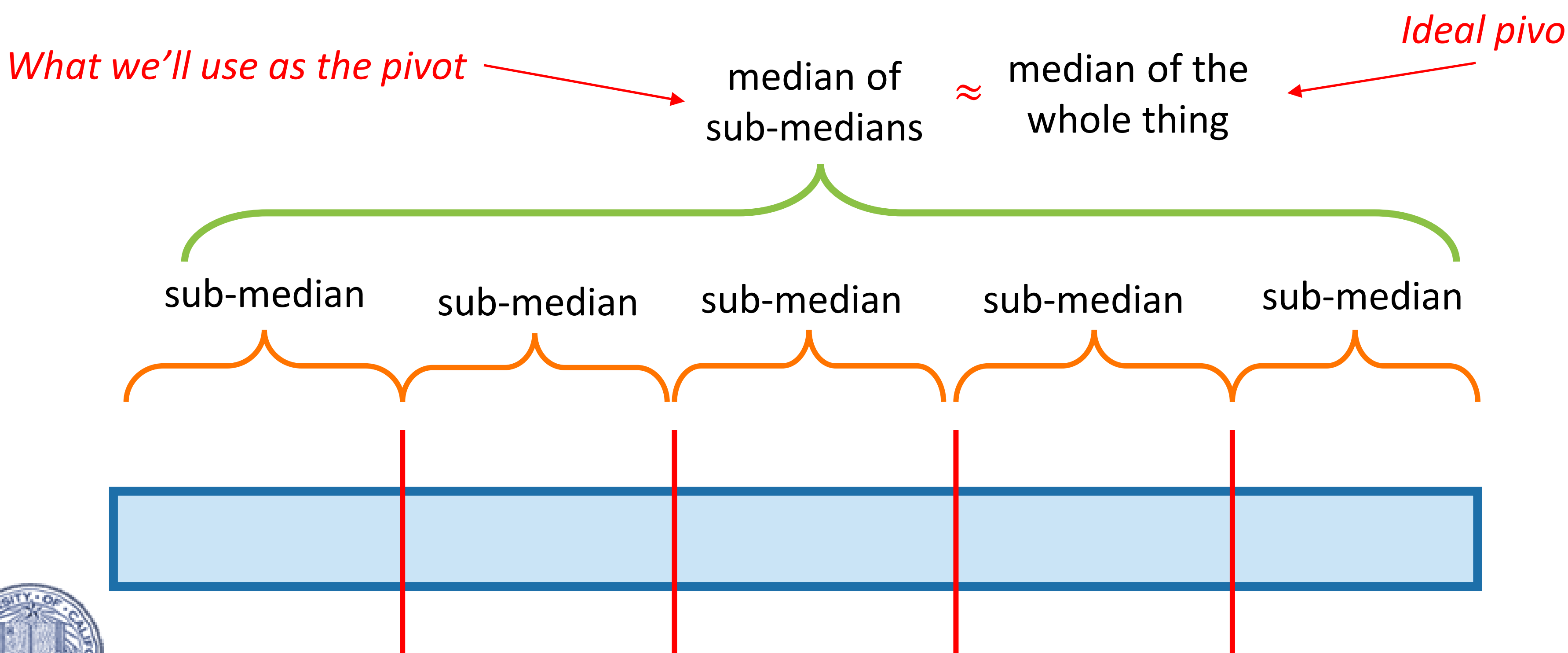
- In time O(n), pick the pivot so that



$$\frac{3n}{10} < \mathrm{len}(L) < \frac{7n}{10} \qquad \frac{3n}{10} < \mathrm{len}(R) < \frac{7n}{10}$$

Another divide-and-conquer alg!

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



*we will make this a bit more precise.

How to pick the pivot

- 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, …, 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)

How to pick the pivot

- 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, …, 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)

1	8	9	3	15	5	9	1	3	4	12	2	1	5	20	15	13	2	4	6	12	1	15	22	3

How to pick the pivot

- 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, …, 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)

1	8	9	3	15

5	9	1	3	4

12	2	1	5	20

15	13	2	4	6

12	1	15	22	3

How to pick the pivot

- **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

1	8	9	3	15
5	9	1	3	4
12	2	1	5	20
15	13	2	4	6
12	1	15	22	3

- 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)

How to pick the pivot

- 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

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.

1	8	9	3	15

5	9	1	3	4

12	2	1	5	20

15	13	2	4	6

12	1	15	22	3

- 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)

How to pick the pivot

- 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, …, p_m] , m/2)
  - **return** p

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.

1	8	9	3	15

5	9	1	3	4

12	2	1	5	20

15	13	2	4	6

12	1	15	22	3

Pivot is SELECT([8 | 4 | 5 | 6 | 12] , 3) = 6:

- 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)

How to pick the pivot

- 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, …, p_m] , m/2)
  - **return** p

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.

1	8	9	3	15

5	9	1	3	4

12	2	1	5	20

15	13	2	4	6

12	1	15	22	3

Pivot is SELECT([8 | 4 | 5 | 6 | 12] , 3) = 6:

6

1 8 9 3 15 5 9 1 3 4 12 2 1 5 20 15 13 2 4 12 1 15 22 3

- 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)

How to pick the pivot

- 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, …, p_m] , m/2)
  - **return** p

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.

- 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)

1	8	9	3	15

5	9	1	3	4

12	2	1	5	20

15	13	2	4	6

12	1	15	22	3

Pivot is SELECT([8 | 4 | 5 | 6 | 12] , 3) = 6:

1	8	9	3	15	5	9	1	3	4	12	2	1	5	20	15	13	2	4	6	12	1	15	22	3

PARTITION around that 6:

1	3	5	1	3	4	2	1	2	4	1	3	5	6	8	9	15	9	12	20	15	13	12	15	22

How to pick the pivot

- 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

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.

- 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)

1	8	9	3	15

5	9	1	3	4

12	2	1	5	20

15	13	2	4	6

12	1	15	22	3

Pivot is SELECT(8 4 5 6 12 , 3) = 6:

1	8	9	3	15	5	9	1	3	4	12	2	1	5	20	15	13	2	4	6	12	1	15	22	3

PARTITION around that 5:

1	3	5	1	3	4	2	1	2	4	1	3	5	6	8	9	15	9	12	20	15	13	12	15	22

This part is L

This part is R: it's almost the same size as L.

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, …, p_m] , m/2)
  - **return** p

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