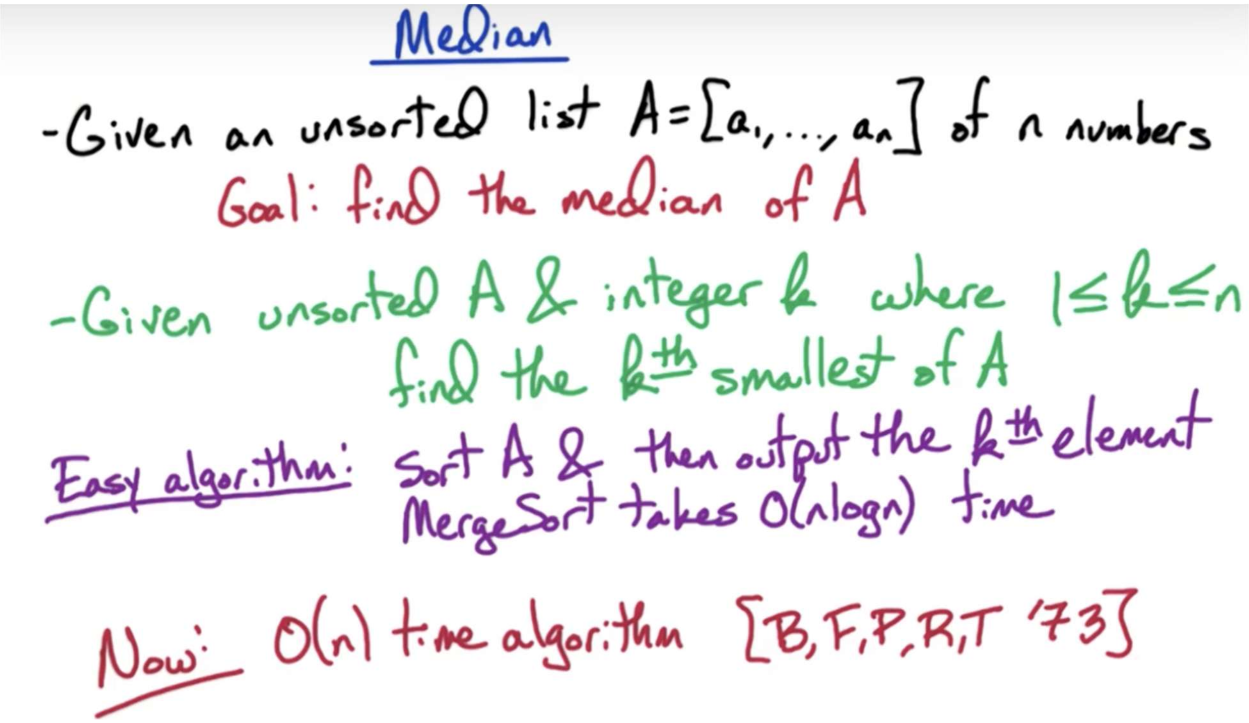
Lesson: DC2: Linear-Time Median

(Slide 1) Median Problem

***



Median

-Given an unsorted list $A=[a_1,\ldots,a_n]$ of n numbers

Goal: find the median of A

-Given unsorted A & integer k where $1\le k\le n$

find the k^{th} smallest of A

Easy algorithm: Sort A & then output the k^{th} element

MergeSort takes $O(n\log n)$ time

Now: $O(n)$ time algorithm [B,F,P,R,T '73]

Let's look now at another nice example of Divide and Conquer. This is the problem of finding the median.

The input to the problem are n numbers, and these are an arbitrary order. So we assume the list is unsorted and input is given to us as this one dimensional array, capital A. Our goal is to find the median of A, this is the middle element.

When n is even, it's not exactly clear what we mean by the median. So for concreteness, let's define the median as the ceiling(n/2)th smallest element of A.

So when n is odd. So n = 2 l + 1 for some integer l. Then in this case, the median is the (l+1)st smallest and there are exactly l which are, at most, this median element, and there is at least l which are at least this median element.

It will be useful for us to solve a more general problem: Instead of finding the median element, we want to find the kth smallest, where k is an input given to us.

More specifically, we're going to look at the following problem: We're given an unsorted list A, just as before, and we're also given an integer k, where k is between 1 and n. Our goal is to find the kth smallest of A. So, if we set k=n/2, then that'll be the median. Now if A happens to be sorted, then it's easy to find the kth smallest. We just output the kth element of the sorted list.

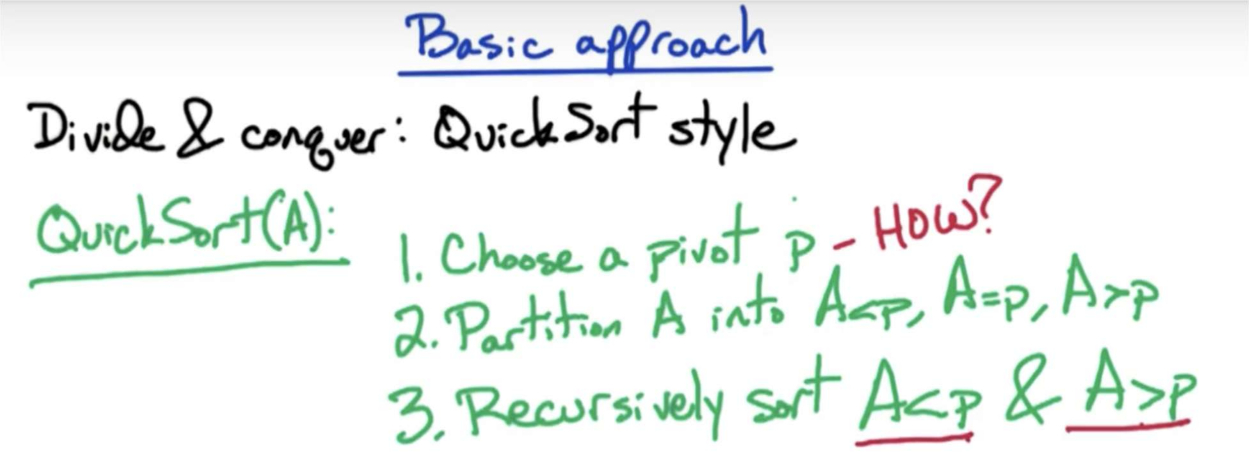
So that gives us a very trivial algorithm for solving this problem. Given an arbitrary A, we simply sort A and then we output the kth element of this sorted list.

How long does this algorithm take? Well, mergesort or several other algorithms take O(n log n) to sort A. So the total runtime of this algorithm will be O(n log n).

Now, is it possible to find the kth smallest without first sorting A? That's what we're going to do now. And, in fact, we'll find the kth smallest in O(n) instead of O(n log n). What we're going to do now is this very clever Divide and Conquer algorithm for finding the kth smallest in O(n).

This algorithm is due to Blum, Floyd, Pratt, Rivest and Tarjan from 1973. The story is that they figured it out over lunch. Once you see the algorithm and it's quite clever, it will be quite impressive that they just figured this out just over one lunchtime.

(Slide 2) Quicksort
***



Basic approach

Divide & conquer: QuickSort style

QuickSort(A):

1. Choose a pivot p - HOW?
2. Partition A into $A<p$, $A=p$, $A>p$
3. Recursively sort $\underline{A<p}$ & $\underline{A>p}$

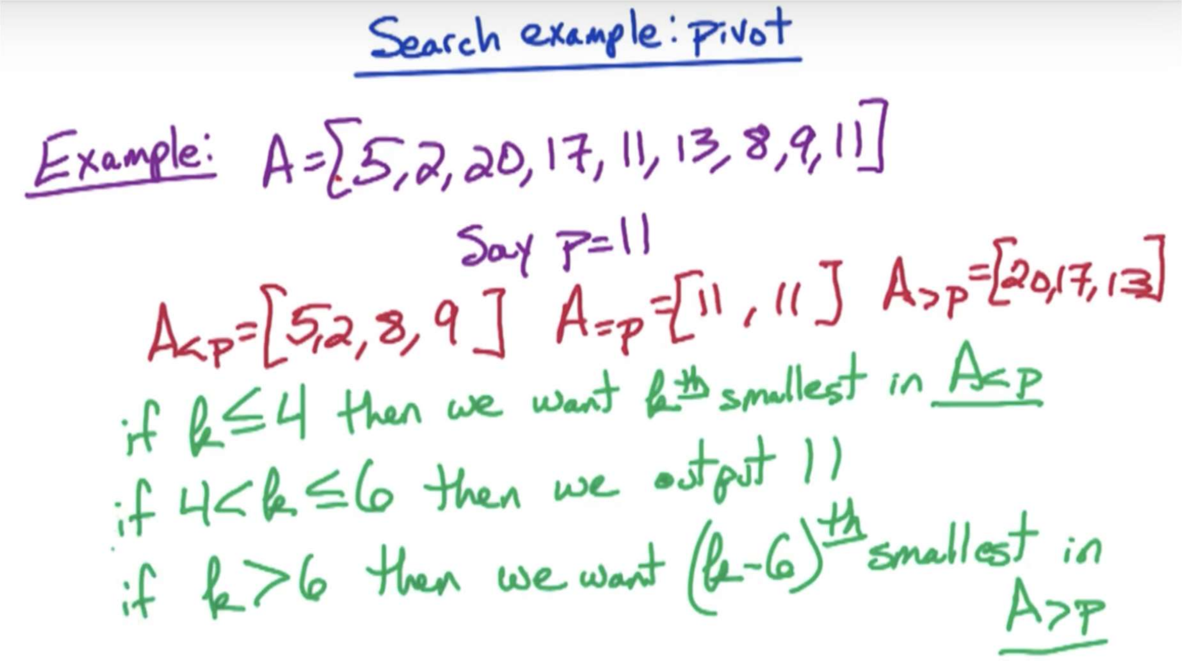
Now we're going to use divide and conquer to solve this case smallest problem. And, our algorithm … the basic approach is going to be quite reminiscent of the quicksort algorithm. Let me remind you about the quicksort algorithm and then we'll see the modifications for our approach.

So we're looking at quicksort for sorting this unsorted list A. The first step in the quicksort algorithm is to choose a pivot, p. Then we partition the array A into three buckets based on their relation to the pivot p. One bucket is for those elements strictly smaller than p, those equal to p, and those strictly bigger than p. We do one scan through the array A and we put each element into one of these three buckets. Then we recursively run quicksort on the small elements and the big elements. We take that output and our final output is the sorted list of the small elements, followed by the equal elements, followed by the big elements.

Now recall the whole challenge in quicksort is how to choose a good pivot. If we choose a terrible pivot, such as the smallest element or the largest element, then one of these two lists is going to be of size n-1. It's just going to go down by one element and then the running time of our algorithm is going to be O(n^2).

So what's a good pivot for quicksort? It's the median, or something close to the median. For our problem we're going to have the same challenge. How to choose a good pivot? That's going to be the main task. But quicksort ideally runs in O(n log n). We're aiming for an O(n) algorithm. The key is that we don't have to recursively consider both A < p and A > p. We only have to recursively search in one of these two lists. Let's look at a specific example to see exactly what I mean for this third step.

(Slide 3) Search Example
***



Search example: pivot

Example: $A=[5,2,20,17,11,13,8,9,11]$

Say $p=11$

$A_{<p}=[5,2,8,9]$ $A_{=p}=[11,11]$ $A_{>p}=[20,17,13]$

if $k \le 4$ then we want k^{th} smallest in $A_{<p}$

if $4 < k \le 6$ then we output 11

if $k > 6$ then we want $(k-6)^{th}$ smallest in $A_{>p}$

Let's look at the following example with an array of n numbers and let's choose a specific pivot. Let's say the pivot is number 11.

So we make these three buckets: A<p, A=p and A>p. And we're going to take a scan through the array and we're going to put each of these elements in one of these three buckets. 5 is smaller than the pivot, 2 is smaller than the pivot and so on. 11 happens to appear twice, so it appears twice in this equal list. And these are the final buckets.

Now, what is the k we're searching for? Depending on the k we're searching for, the kth smallest will be in one of these three lists. And we can figure out where it resides based on how large these lists are in comparison to k. The smallest list is those of size 4.

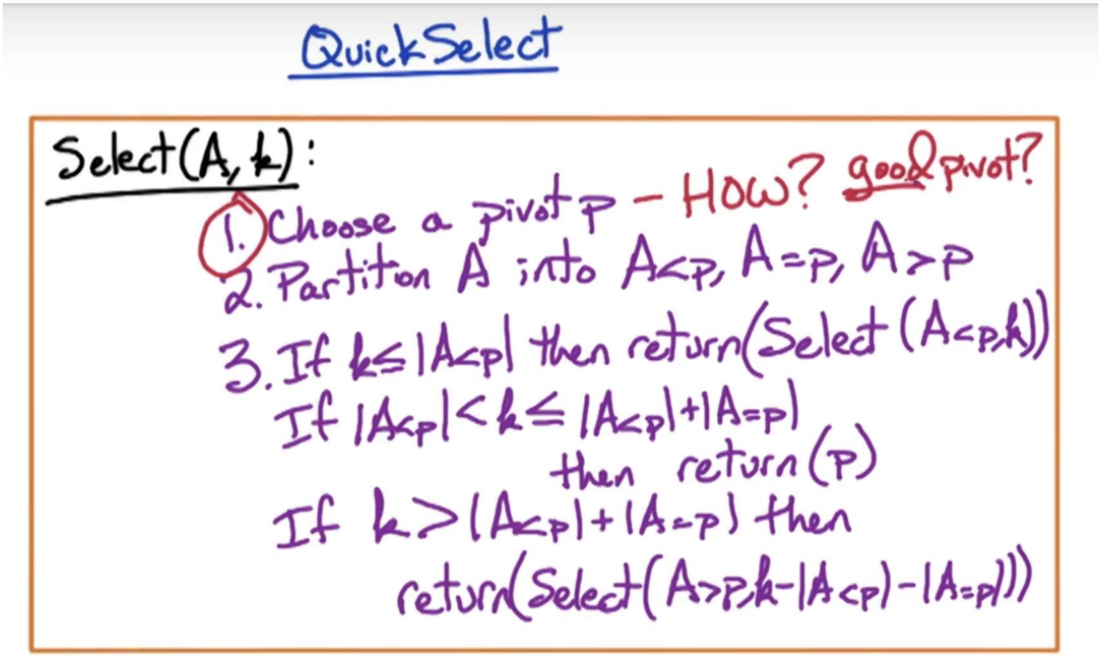
- So if k <= 4 then we know the kth smallest is in this list. So in this case, what we can do is we can recursively search for the kth smallest in this list and we can discard these two other lists.
- Now if k is 5 or 6, then what do we know? Then we know the kth smallest is in this list and then, therefore, it's equal to 11. So we don't need to recurse at all, we can just output 11.
- Finally, if k is bigger than 6, then what do we know? We know that the kth smallest is in this list, so we don't search for the kth smallest in this list because we discarded six

elements from these lists. So we're going to search for the k-6th smallest in this list, A>p.

Now, the key is that we're always recursing on at most one list, either the small list or the big list. Or, in the middle case, we don't even have to recurse at all; whereas, QuickSort has to recursively sort these two lists, A < p and A > p. In this case, we only have to recursively search in one of the two lists.

Now let's go ahead and detail this for the general case and let's write the pseudocode to make sure it's all clear.

(Slide 4) QuickSelect
***



QuickSelect

```
Select(A, k):
    1. Choose a pivot p – HOW? good pivot?
    2. Partition A into $A_{<p}, A_{=p}, A_{>p}$
    3. If $k \le |A_{<p}|$ then return(Select($A_{<p}$, k))
       If $|A_{<p}| < k \le |A_{<p}| + |A_{=p}|$
               then return(p)
       If $k > |A_{<p}| + |A_{=p}|$ then
           return(Select($A_{>p}$, $k - |A_{<p}| - |A_{=p}|$))
```

So we're given this unsorted list A and we've given an parameter k and we're trying to find the kth smallest in A.

- Our first step, as in the quick sort algorithm, is to choose a pivot P. How exactly do we do that? Well, that's exactly our going to be our main task. We'll get back to that later.
- After we choose a pivot P, we partition A into three sub-arrays based on this pivot, the smaller elements, equal elements and bigger elements. Now we're going to recursively search in one of these three lists based on k and the size of this list.
- Now, if this small list is to say of size 4, and k is at most four, then we know that the kth smallest is in the small list. And in general, if k is at most the size of this list, then we can recursively search for the kth smallest in the small list.
  So we run this algorithm recursively on this small list and search for the kth smallest and that's the output of our algorithm.
- In the middle case, k is bigger than the size of the small list but it's not big enough that it resides in the big list. So therefore, we know that the kth smallest is in the middle list and therefore the kth smallest is exactly P. So we can just output P, no recursion needed in this case.
- Now in the final case, we know that the kth smallest resides in the big list. So we're going to recursively search in the big list but we're not going to search for the kth smallest, we're going to have to shift it, so we know that we're discarding these many elements, the size of these two lists. So we're going to look for the k - |A<p| - |A=p| - So, instead of searching for the kth smallest in this large list, we're going to look for the

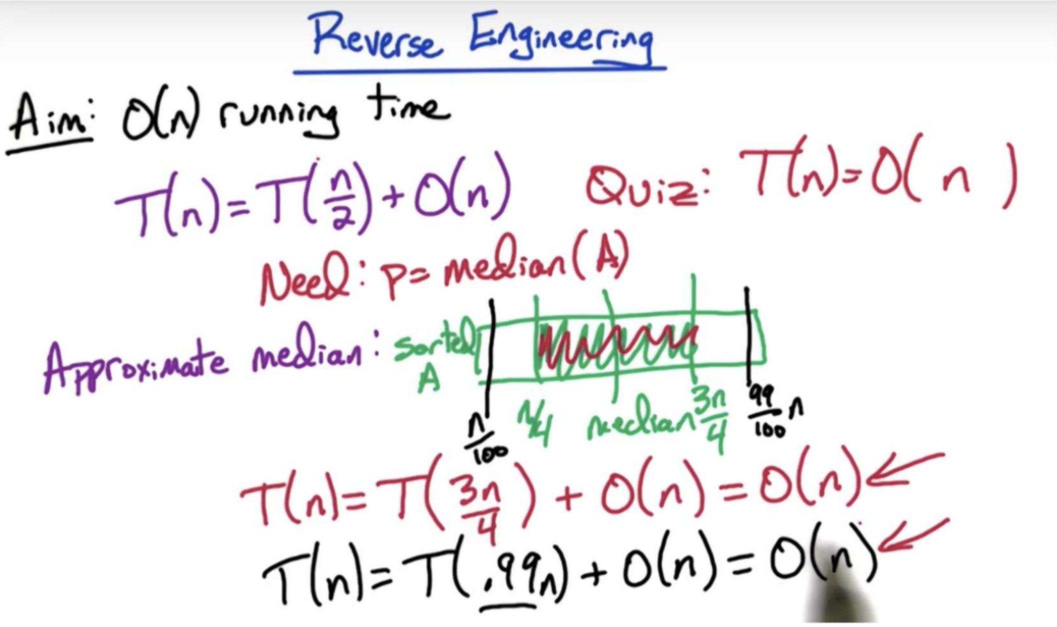
k minus the size of the small list, minus the size of the equal list. These (A<p and A=p) are the elements that we're discarding.

Now, this is the basic algorithm. But the key part is, how do we choose a pivot and what constitutes a good pivot? What does it mean to be good? Let's first look at what a good pivot means. What's a pivot which would lead to an O(n) algorithm?

(Slide 5) DC2: Linear-Time Median Simple Recurrence Quiz

(Slide 6) High-level idea

***



Reverse Engineering

Aim: O(n) running time

$T(n)=T(\frac{n}{2})+O(n)$ Quiz: $T(n)=O(n)$

Need: p= median(A)

Approximate median:

$T(n)=T(\frac{3n}{4})+O(n)=O(n)$

$T(n)=T(.99n)+O(n)=O(n)$

Now we're aiming for an O(n) running time and we're trying to use Divide and Conquer. So let's look at the recurrences which yield an O(n) solution and then this will give us some idea of the algorithm basic approach.

Take a look at this recurrence - What does this recurrence solve to? T(n) = T(n/2) + O(n). This recurrence solves to O(n).

How can we achieve a running time like this from the basic approach that we align on the previous slide? We have one subproblem which is of size at most n/2. In our previous approach, we either recurse on A< p or A > p. In order to achieve that the subproblems of size at most n/2, we need that the pivot p is the median. If the pivot is the median of the unsorted list, then we know that A< p is at most n/2 and a bigger n p is of size at most n/2 and therefore the running time of our algorithm will be exactly this which solves to O(n).

Now the problem is that the whole problem that we were trying to solve originally was to try to find the median of this list. So if you give us a solution then I can run O(n) time and I can find the solution again. That's the punch line so far.

But what if I don't actually have the median exactly? What if I have an approximate, a reasonable guess of the median? So suppose p is an approximate median, it's fairly close to the median but it's not exactly the median.

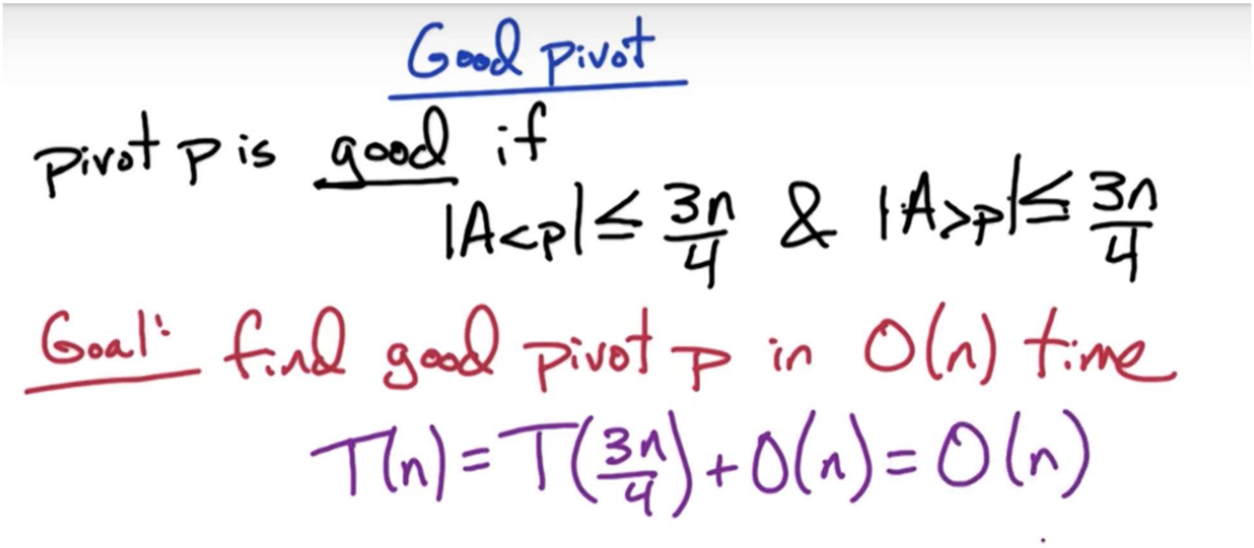
Let's just do a thought process. Let me look at the sorted array A. So I'm not actually sorting A but I'm just thinking about it … this array A as being sorted … and the median little sign right here in the middle. And here I have the smallest element, here I have the largest element. Let me look at n/4th smallest and let me look at the 3n/4th smallest. Suppose instead of giving you the median, I just give you something which is guaranteed to lie within this band. So it's at least the n/4th smallest and it's at most 3n/4th smallest. So it's not lying in either of these extremes, it's lying in this middle band. Suppose I can find a pivot which satisfies this. So it's going to lie in this band.

What does that imply about the running time of my algorithm? How large are these sub problems going to be in the worst case? When the worst case maybe it lies right here, it's a 3n/4th smallest and then A<p is going to be size 3n/4. Similarly, if it's n/4th smallest then A>p is going to be of size 3n/4. And in general, I know that the size of the sub problem is going to be at most 3n/4. So my recurrence is going to satisfy this relation.

Now the question is, what does this recurrence solve to? Turns out, it's still solves to O(n) and in fact, I can relax it even more. I don't even need three quarters here. I can look at n/100th smallest here and then .99n smallest over here. I just have to chop off a constant fraction on both sides. So what is my recurrence going to satisfy in this case where i lies in this bigger bandwidth? So my pivot is at least the n/100th smallest and is at most 99/100th smallest. In this case, the size of my subproblem is the most .99n. So my recurrence satisfies this relation. What does this solve to? This also solves the O(n). The key is, I need a constant here which is strictly less than one. So I'm always chopping off a constant fraction of the nodes.

Now we're going to define a good pivot as a pivot which lies in this middle band between n/4th and 3n/4th smallest. That's going to give us this recurrence. But it's going to be important for our algorithm actually to remember this recurrence. We have some extra slack, so we're going to aim for a pivot which satisfies this relation but they're always going to be some extra slack because I'm allowed to have any constant less than one here and we have to utilize that extra slack in order to find the pivot which is a good pivot. So in the end, our running time is going to satisfy a relation similar to this, bottom recurrence. But our definition of what is a good pivot is lying in this middle band between n/4 and 3n/4.

(Slide 7) Goal: Good Pivot
***



Good pivot

pivot p is good if

$$|A_{<p}| \leq \frac{3n}{4} \quad \& \quad |A_{>p}| \leq \frac{3n}{4}$$

Goal: find good pivot p in O(n) time

$$T(n) = T\left(\frac{3n}{4}\right) + O(n) = O(n)$$

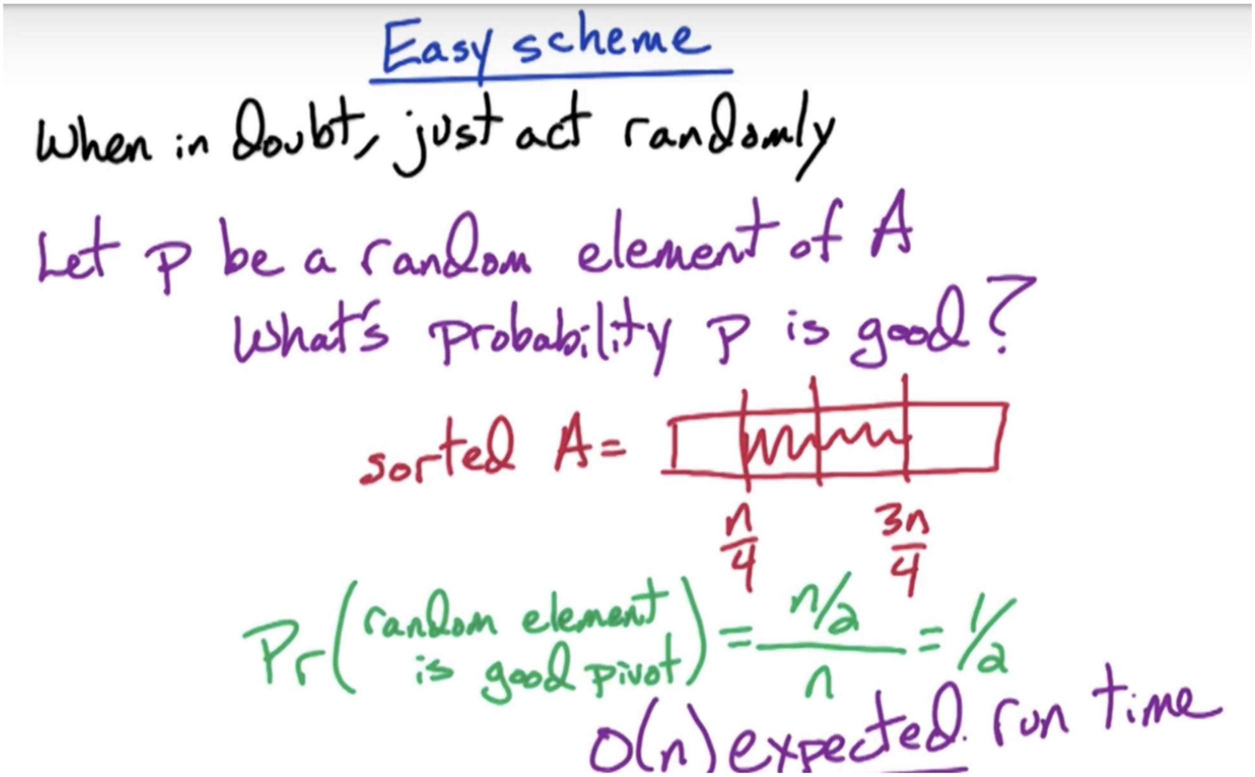
We're going to say a particular pivot p is good if this pivot p is at least the n/4th smallest, and is at most 3n/4th smallest.

What does that imply? That implies that the number of elements which are strictly less than this pivot is at most n/4 and the number of elements strictly bigger than this pivot is at most 3n/4.   So, if this is satisfied, then we say the pivot p is good.

Our main task is to figure out how to find this good pivot p in O(n) runtime. If we can do that, then we're going to get a recurrence such as this. We're going to have a subproblem of size at most 3n/4, because of this relation. And it's going to take us O(n) to partition the array and to find the good pivot. And then, we know that this recurrence solves to O(n), so we'll be done.

Question is, how can we find a good pivot in O(n) runtime?

(Slide 8) Random Pivot
***



Easy scheme

When in Doubt, just act randomly

Let p be a random element of A

What's probability p is good?

sorted A =

$$\Pr\left(\begin{matrix}\text{random element}\\ \text{is good pivot}\end{matrix}\right) = \frac{n/2}{n} = 1/2$$

$O(n)$ expected run time

What's an easy scheme to find a good pivot? Well, if I have no idea what to do, what should I do? I might as well act randomly. So in our case, what does that mean? That means let p be a random element of A - choose a random element of A and set that to be the pivot p. Now, what's the probability that p is a good pivot?

Let's look at our thought experiment from before. So let's look at the sorted array A. We're not actually sorting A. We're just looking at the sorted version of A for the purposes of analyzing the probability that p is good. We have the median element. We have the n/4th smallest, and we have the 3n/4th smallest. What are our good pivots?

Everybody in here is a good pivot. How many good pivots are there? There is exactly n/2 good pivots. So, what's the probability a random element is a good pivot? I can order this however I want. The fact is, exactly n/2 of these elements are good pivots and there's exactly n choices. So the probability a random element is a good pivot is the number of choices which lead to a good pivot divided by the total number of choices. The number of good pivots is n/2 and the total number of choices is n.   Simplifying this, we get one half, exactly half the elements are good pivots. So, no matter how you order this array A, we've got a probability exactly a half of finding a good pivot.

Now, given a proposed element as a pivot, how can I check whether it's a good pivot or not. Well, I can just bend O(n) and I can break partition A into those elements smaller, equal, or bigger than p. And if I keep track of their sizes as I go along, then I can easily check in O(n) whether this proposed pivot p is good or not.

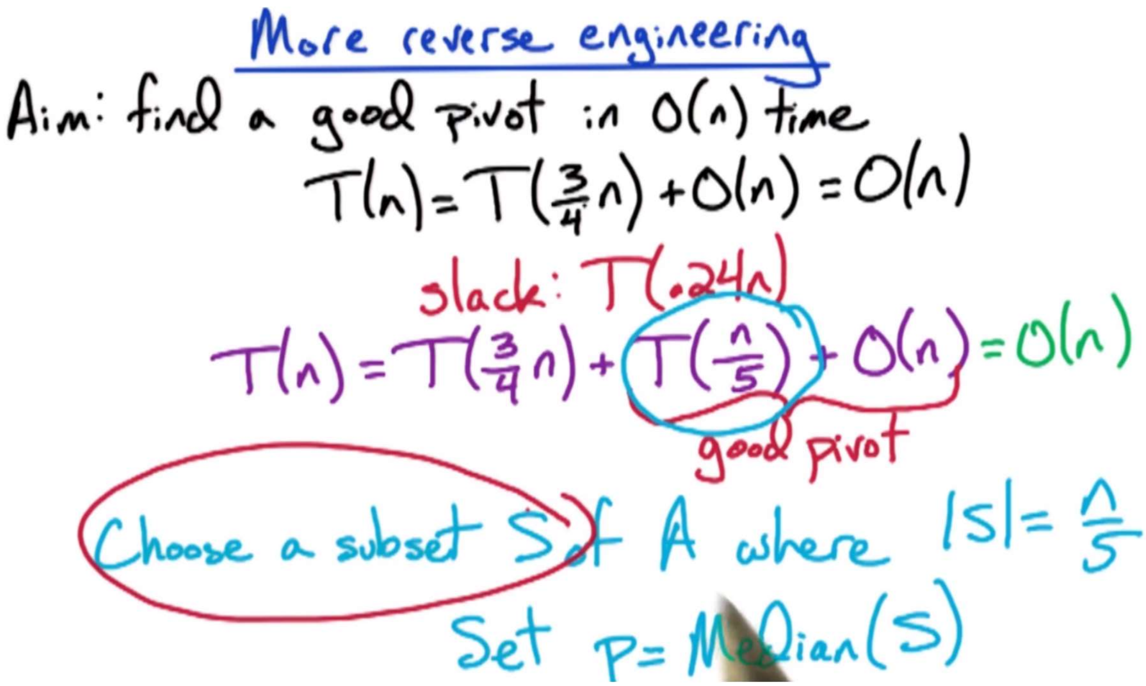
What happens if it's a bad pivot? What should I do in that case? Well, I can rerun this experiment again. So, I choose a new random element of A and then I check whether it's a good pivot or not. If it's still a bad pivot, then I run the experiment again and I keep going until I find a good pivot. Once I find a good pivot, I use it.

In expectation, how many times am I going to have to repeat this experiment until I find a good pivot? This is like flipping a coin. If it ends up with tails, then I'll say that's a good pivot. If it ends up with heads, that's a bad pivot. I got probability exactly a half of finding a good pivot. So I got probably exactly a half of ending in tails. I keep flipping the coin until I get a tail. Once I get one tail, that's a good pivot and I start my experiment. How many times am I going to have to flip the coin until I get A tails? In expectation, twice. So it's going to take me O(n) Expected time to find a good pivot. So the Expected runtime of the whole algorithm is going to be O(n). So this is a reasonable algorithm.

But all I have guaranteed is that the expected runtime is O(n). I want an algorithm whose guaranteed worst case runtime is O(n). So, how can we guarantee to find a good pivot in O(n) time? That's the task we're going to try to tackle now.

(Slide 9) Recursive Pivot

***



More reverse engineering

Aim: find a good pivot in O(n) time

$$T(n) = T(\tfrac{3}{4}n) + O(n) = O(n)$$

slack: $T(.24n)$

$$T(n) = T(\tfrac{3}{4}n) + \underbrace{T(\tfrac{n}{5}) + O(n)}_{\text{good pivot}} = O(n)$$

Choose a subset S of A where $|S| = \frac{n}{5}$

Set $p = $ Median(S)

Now our aim is to try to find a good pivot in O(n) and we're going to do this in worst case, O(n). What we just saw is an algorithm which finds it in O(n) Expected time.

Now if we can successfully find a good pivot in O(n) runtime, then, the running time of our algorithm will satisfy the following relation. T(n) = T(3/4 n) + O(n) … ¾ n because it is a good pivot - the sub-problem will be of size at most three-fourths the original size, plus O(n) to find the good pivot and O(n) to partition A into the three sets.

Now, this recurrence solves to O(n). T(n) = T(3/4 n) + O(n) = O(n). So, the overall running time of our algorithm will be O(n).

Now, we have a bit of slack in this recurrence. In particular, recall that instead of having 0.75 n over here - we could have had 0.99 n. We just need some constant less than one. What we're going to do is we are going to use this extra slack as extra time to help us find a good pivot.

So, now, we are going to spend, instead of, just O(n) runtime to find a good pivot, we're going to spend O(n) + T(0.2 n). So, we are going to design an algorithm with the following running time. T(n) is at most T(3/4 n)…this is for the sub-problem; Once we find a good pivot, + T(n/5) + O(n). This T(n/5) + O(n) is going to be the time it's going to take us to find a good pivot.

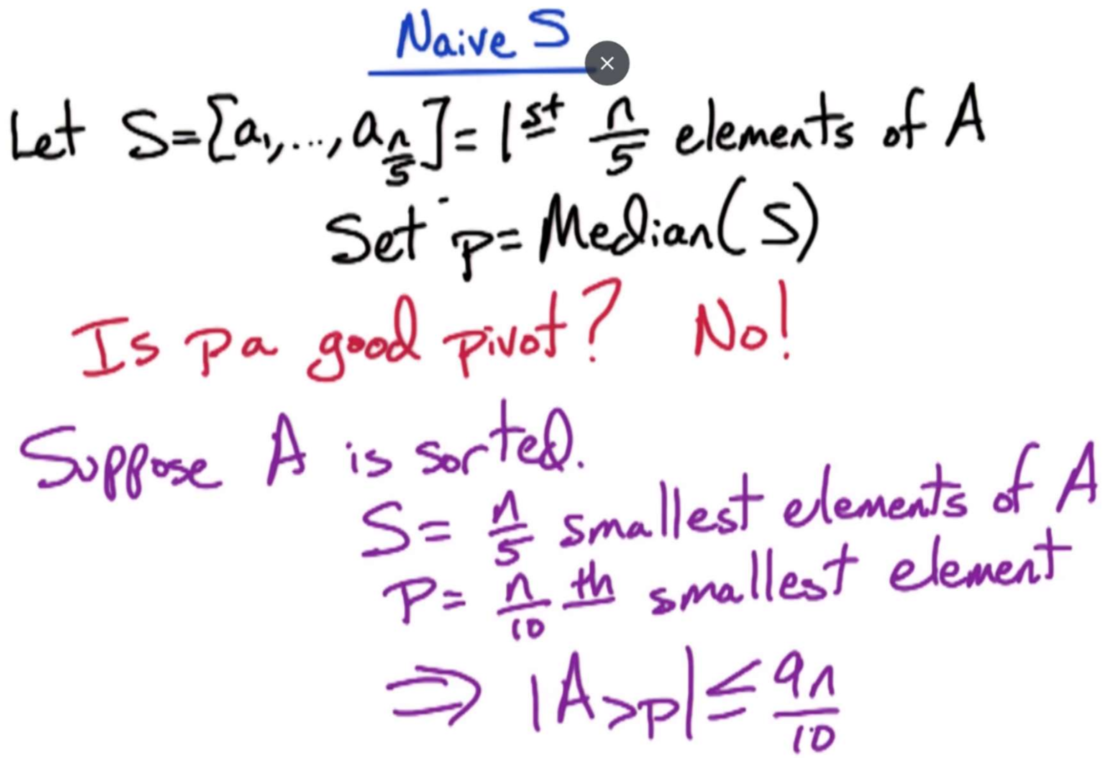
Now the key fact is that ¾ + 1/5 = 0.95 < 1. It's strictly less than one. So, this recurrence solves to

O(n). So, if we can find a good pivot in this amount of time - T(n/5) + O(n) - then we successfully have an O(n) runtime algorithm to find the median.

Now how exactly are we going to utilize this T(n/5) - How are we going to utilize it? Well, we're going to choose a subset of A which is of size n/5 and then we're going to recursively run our median algorithm on this subset S. And we're going to set p, the pivot to be the median of this subset S. The time it's going to take us to find the median of this subset S is going to be T(n/5) since the subset S is of size n/5.

The question is how do we choose this subset S? We need to choose this subset S so that it is a good representative sample of this entire array A. Let's take a look first at a naive choice of the set S and then we'll see how that fails and it will give us some intuition about choosing a better choice for this subset S.

(Slide 10) Representative Sample
***



Naive S

Let $S=[a_1,\dots,a_{\frac{n}{5}}]=1^{st}$ $\frac{n}{5}$ elements of A

Set $p=\text{Median}(S)$

Is p a good pivot? No!

Suppose A is sorted.

$S=\frac{n}{5}$ smallest elements of A

$P=\frac{n}{10}$ th smallest element

$\Rightarrow |A_{>p}|\le\frac{9n}{10}$

What's a simple idea for choosing this subset S, which is of size n/5? Well, perhaps the simplest idea is let the subset S just be the first n/5 elements of A. Then we're going to set a pivot to be the median of this subset S. How does this pivot perform? Is it a good pivot? Is this pivot p a good pivot? No. Why not? Well, that would be too easy. Let's see why it fails. Let's see the scenario where this pivot p is not a good pivot.

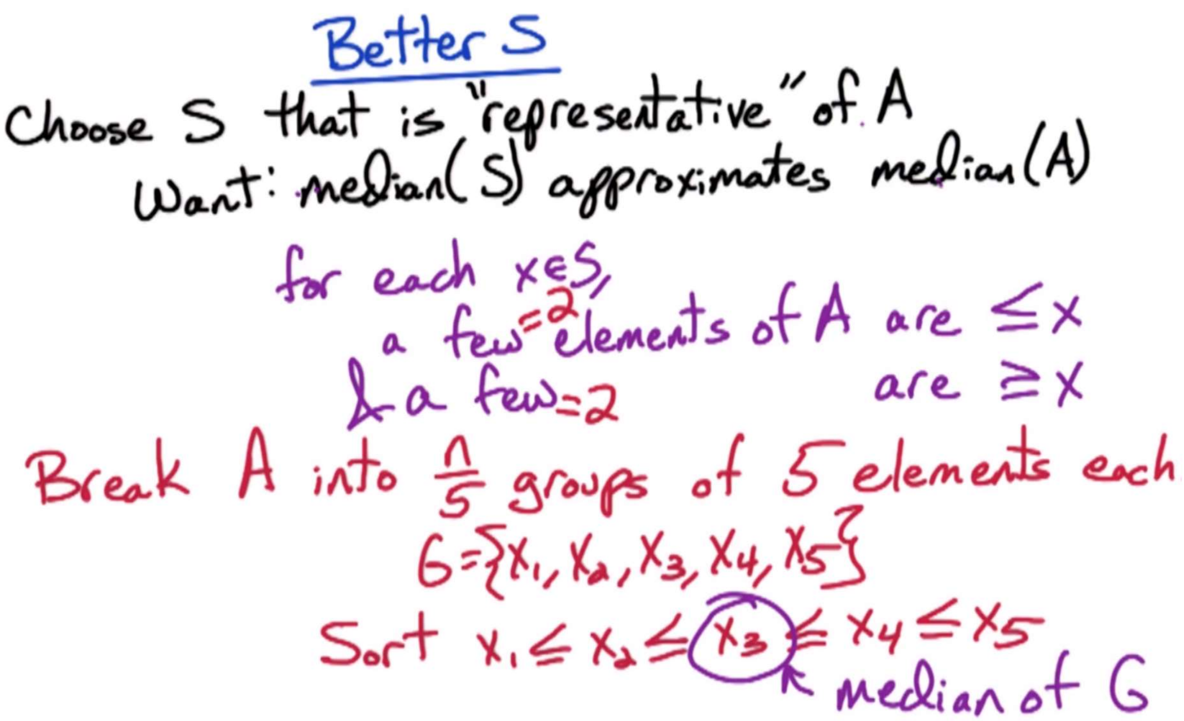
Well, suppose A was sorted. Well in this case, actually, it's easy to find the median of A. So, there's no reason we're running the whole algorithm. But suppose A is sorted and then, actually, this is going to imply that this pivot P is a bad pivot. So if A is sorted, then what is S? S is going to be the first n/5 elements of A. So it's the smallest elements of A. So S contains all the n/5th smallest elements of A. What is the median of these n/5th smallest elements of A? The median of the n/5th smallest elements is the n/10th smallest element. So, this pivot is the n/10th smallest element of this list A.

Well if this is the n/10th smallest element, then when we partition A into the small, equal and big sets, the big set - all we can guarantee is that it's at most 9/10ths n. And then if you go back to the running time of our algorithm, then we're going to have T(n) is at most T(9/10 n) instead of T(¾ n).   So this pivot P is not a good pivot, because this large set is going to be too large in the worst case.

So is there a better choice for this set S? Well, there's some hope because we chose this set S right now without looking at A at all. So, can we look at A and do a little bit of computation and choose a better representative set S?

(Slide 11) Recursive Rep. Sample

***



Better S

Choose S that is "representative" of A

Want: median(S) approximates median(A)

for each $x \in S$,

a few=2 elements of A are $\leq x$

& a few=2 are $\geq x$

Break A into $\frac{n}{5}$ groups of 5 elements each

$G = \{x_1, x_2, x_3, x_4, x_5\}$

Sort $x_1 \leq x_2 \leq \boxed{x_3} \leq x_4 \leq x_5$

median of G

Our goal is to choose a subset S which is a representative of the set A. What exactly do we mean by representative of the set A? We want that the median of this subset S is also a reasonable approximation to the median of the set A. Now the median of A has half the elements smaller than it and half the elements bigger than it.

Now in order for the median of S to approximate this median, what we want is that each element of S is somehow has a similar flavor. So we want that each element of S has the following property: We want that there a few elements of A that we can guarantee are at most x and a few elements of A are at least x so that these elements of A, which are represented by this element X.

What exactly do we mean by few? Well, let's say it's two. We have at least two elements of A, which are at most x and at least two elements, which are at least x and then if we combine, add in x itself, then we've got five elements here that we're considering. We want to look at sets of five elements and we want x to represent that set of five elements. So what we're going to do is we're going to break A into n/5 groups of 5 elements each.

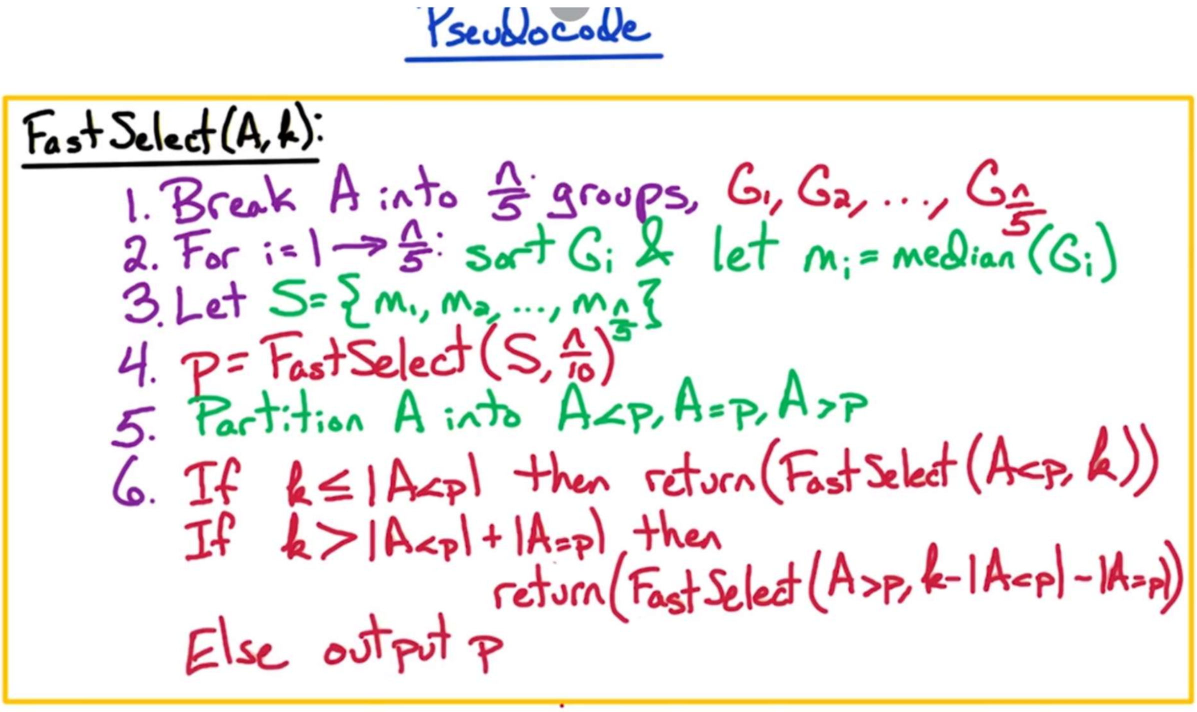
Let's assume n is a power of five, so that this can be done cleanly. Now how are we going to get this subset S from this partition of A? Well, we're going to choose one representative from each group. Each group is going to choose or we're going to choose one representative one element of this group, which is going to represent the group in the following sense: It's going to represent the group in the sense that at least two of the elements of this group are at most the chosen element, and at least two of the elements are at least the chosen element.

Let's take a look at a particular group to see how we do this. Let's look at a group G consisting of the elements G = {x1, x2, x3, x4, x5}. Now who do we want to choose from this group to represent this group in the sense that we talked about just before? Let's look at this group sorted. Let's sort the group and relabel it so that x1 is the most, x2 is a most, x3 and so on. Now who are we're going to choose to represent it in this sense? Well, we're going to choose the middle element. The median of this group. x3 has the desired property. At least three elements of G are at most x3, including x3 itself, and at least three elements including x3 again are at least x3. If we take the median of each of these groups, then that gives n/5 elements and each of those chosen elements in the subset S, it represents a distinct group of five elements so that each have this desired property and this subset S is going to be a good representative sample of this entire array A.

And that's it. Now we have the whole idea for finding a good pivot. We take this array A, we break it into n/5 groups of 5 elements each. For each group, what we do is we sort it. How long does it take us to sort? Well, notice this group is of five elements. So sorting it takes O(1) runtime. It doesn't matter what algorithm we use. If we take an exponential time algorithm, exponential in five is still O(1). So it takes us O(1) time to sort this group and then we can take the median element of this group and that's going to be the representative sample for this group and we take the one median to one representative sample from each group. That gives us n/5 samples and that gives us our subset S and then we take the median that subset S, recursively we find it and that gives us the pivot that we use for A. And then we're going to prove that that pivot that we find, which is the median of this subset S, is a good pivot.

Let's go ahead and detail the pseudocode for this algorithm and then we'll go back and look at the claims that we're making about this pivot … being a good pivot … and look at the running time of our algorithm.

(Slide 12) Median Pseudocode
***



Pseudocode

```
FastSelect(A,k):
    1. Break A into $\frac{n}{5}$ groups, $G_1, G_2, \ldots, G_{\frac{n}{5}}$
    2. For $i=1 \to \frac{n}{5}$: sort $G_i$ & let $m_i = median(G_i)$
    3. Let $S = \{m_1, m_2, \ldots, m_{\frac{n}{5}}\}$
    4. $p = FastSelect(S, \frac{n}{10})$
    5. Partition A into $A_{<p}, A_{=p}, A_{>p}$
    6. If $k \le |A_{<p}|$ then return(FastSelect($A_{<p}$, k))
       If $k > |A_{<p}| + |A_{=p}|$ then
                return(FastSelect($A_{>p}$, $k - |A_{<p}| - |A_{=p}|$))
       Else output p
```

Now, let's detail the pseudocode for a linear time median algorithm. The input to the algorithm is unsorted array A, of size n, and an integer k where the integer k lies between 1 and n. And the output of the algorithm is the kth smallest element of the array A.

1. The first thing we need to do is find a good pivot. To do that we break A into n/5 groups of five elements each. Now to be precise, we should say the ceiling of n/5 groups, because n might not be a multiple of 5. But let's ignore floors and ceilings in this pseudocode.

   Let's denote these n/5 groups as G_1, G_2 up to G_n/5. Now, how exactly do we break A into these n/5 groups. Well, we can do it in any arbitrary way we like. The easiest way to do it, is to take the first five elements of A and put those into group G_1, take the next five elements of A and put those in group G_2 and so on.

   Now we have chosen one representative from each of these groups.

2. Here's a **for loop** to go over the n/5 groups. For the ith group, group G_i, we want to find the median. To do that we first sort this group. There's only five elements so we can take any algorithm we like to sort it. And then we take the median of these five elements. And let's let mi denote the median of group G_i.

Now we want to look at these n/5 medians which we found in step two.

3. Let's let capital S denote this set of n/5 medians.

4. Next, we want to find the median of this set S. This will be our pivot p. How do we find this pivot p? How do we find the median of this set? Well, we recursively call this same algorithm FastSelect, on this subset S. Now S has n/5 elements, we want to find its median, therefore we look for k=n/10. The n/10th smallest element of S, is the median of this set S. And we store that in element p.

   Now we use p as our pivot.

5. We partition the original set A into three sets. Those elements less than pivot p, those equal to the pivot p, and those bigger than the pivot p. This requires just one scan over the set A.

6. Now we can use the quickselect approach from before. Based on the sizes of these three sets, we either recursively search in small set, the big set, or we simply output p. In particular, if the size of the small set is at least as large as k, then we know that the k-th smallest lies in this small set.

   So, we recursively run this same fast select algorithm on the small set, looking for the k-th smallest.

   Now if k is big enough, in particular, if k is bigger than the size of a small set plus the equal set, then we know the kth smallest lies in this big set. So, we recursively run FastSelect on the big set with this k scale …we set k = k minus the size of a small set and minus the size of the equal set … the part that we discarded.

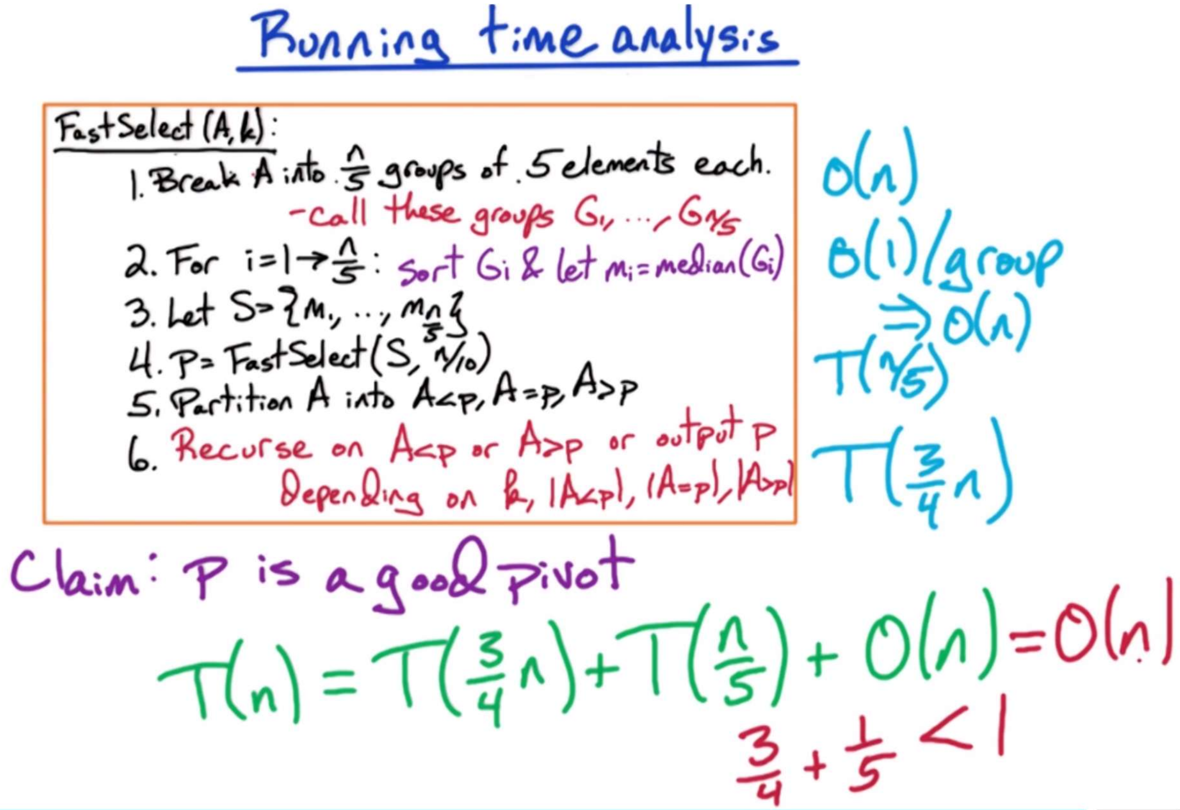
   Finally, if neither of these two cases held, then we know that the kth smallest lies in the equal set, and therefore we simply output P.

   Now these three cases are simply the same as we detailed before for the quickselect algorithm.

This completes the pseudocode for our algorithm.

Now we can analyze this running time assuming that this pivot P that we found is a good pivot. And then we'll go back and prove that this pivot P is in fact guaranteed to be a good pivot.

(Slide 13) Median: Running Time
***



Running time analysis

```
FastSelect(A,k):
    1. Break A into $\frac{n}{5}$ groups of 5 elements each.
        -call these groups $G_1, \dots, G_{n/5}$
    2. For $i=1 \to \frac{n}{5}$: sort $G_i$ & let $m_i = \text{median}(G_i)$
    3. Let $S = \{m_1, \dots, m_{\frac{n}{5}}\}$
    4. $p = $ FastSelect$(S, n/10)$
    5. Partition A into $A_{<p}, A_{=p}, A_{>p}$
    6. Recurse on $A_{<p}$ or $A_{>p}$ or output p
        Depending on k, $|A_{<p}|, |A_{=p}|, |A_{>p}|$
```

$O(n)$

$O(1)$/group $\Rightarrow O(n)$

$T(n/5)$

$T(\frac{3}{4}n)$

Claim: p is a good pivot

$$T(n) = T\left(\frac{3}{4}n\right) + T\left(\frac{n}{5}\right) + O(n) = O(n)$$

$$\frac{3}{4} + \frac{1}{5} < 1$$

We're going to prove that this P that we chose is in fact a good pivot. Now let's assume this fact for now and then let's look at the running time of this algorithm. Let's look at the running time step-by-step.

1. The first step, breaking A into these n/5 groups. How long does that take? Well, that just takes one swipe through the array A. So, that takes O(n).
2. Now, we have to sort each group. How long does it take to sort one of these groups? Well, it's only five elements. So, even if we use a super slow algorithm, let's say 5!, we write down every permutation of these five elements and choose the one which is sorted, that's going to take us 5! runtime which is still O(1). So, it takes us O(1) runtime per group and there's O(n) groups. So, the total time for this step is O(n).
3. [skipped] Let S = {m1,…,m_n/5}
4. Now, we're going to run this same algorithm recursively on this subset S, which is of size n/5. So how long does that take? That takes us T(n/5) since this subset of size n/5.
5. Now we partition A into these three subsets, that takes one swipe through the array A, that also takes O(n) runtime.

6. And then finally, we recurse on one of these smaller subproblems. How large are these subproblems? Well, since P is guaranteed to be good pivot, we know that these subproblems are size, at most, ¾ n.

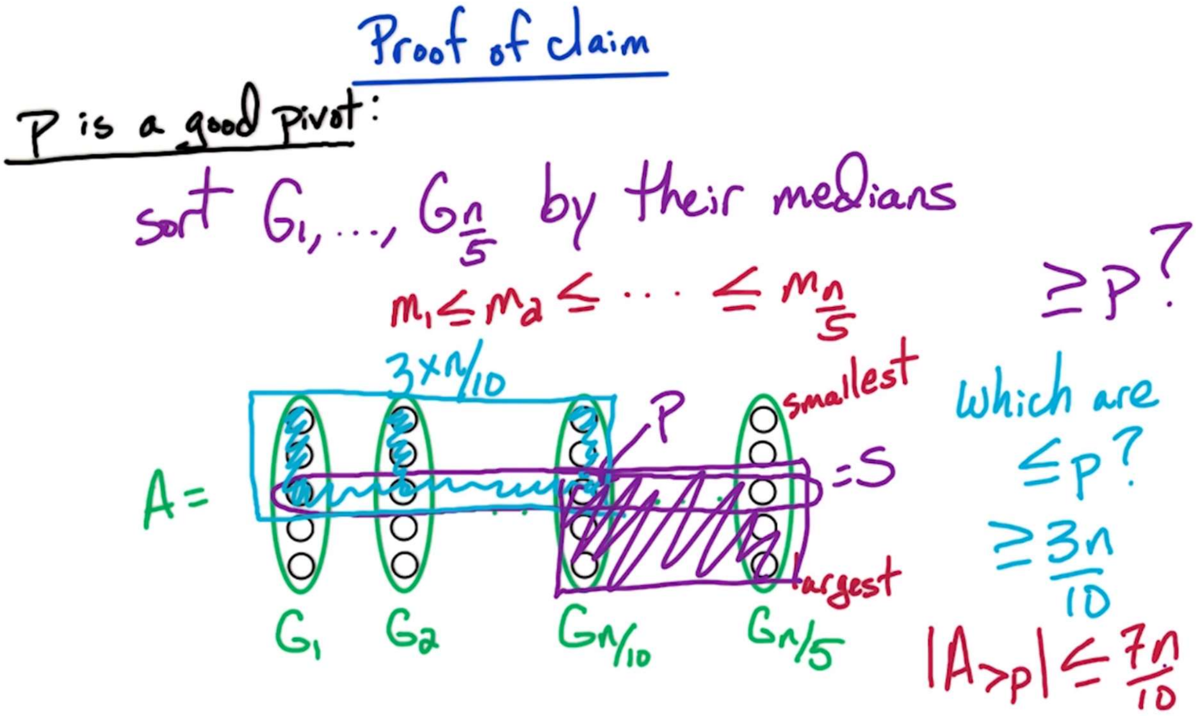
So, our running time satisfies the following recurrence: $T(n) = T(3/4\ n) + T(n/5) + O(n) = O(n)$

We have three-fourths n, we have n/5 for finding this median of this subset S, and then plus we have O(n) for several of these steps. The key is that $¾ + 1/5 = 0.95$, which is strictly < 1. So, this recurrence solves to O(n) and we have an O(n) runtime algorithm.

Now it remains to prove this claim, that p is a good pivot. Once we prove that p is a good pivot, we're all done with our algorithm.

(Slide 14) Linear-Time Median Correctness

***



Now, we are going to prove this claim that this p that we chose in our algorithm is a good pivot.

How do we find this p? Well, we took these groups G_1 through G_n/5, each with five elements each. And we took the median of each of these groups and then we took the median of those medians. So we took these n/5 medians of these groups, and then we recursively ran our algorithm to find the median of that subset of size n/5 and that was our p.

Now what I want to do is I want to sort these groups and I want to sort them by their medians. So this has a smallest median, this has the largest median. I'm just relabelling the labels on the groups. Now this is just in the proof. The algorithm is staying the same just for the purposes of the proof as do the thought experiment where we relabel these groups.

So we've relabeled these groups so that the first group has the smallest median and the last group has the largest median.

Let's look at this set A pictorially, and let's look at it by groups. Here's the first group that consists of five elements. Here's the second group G_2. It consists of five elements and so on. And here's the last group G_n/5, which consists of five elements.

Now, it'll be important for us to consider the middle group, the group G_n/10. Now, let's sort each of these groups by smallest to largest. This will be the smallest element of this group and this will be the largest, and this will be the median. This will be the smallest, largest, median, and so on. In every group, the middle element is the median. This is just for the picture purposes.

Now what does our algorithm do? It takes the median of each of these groups. Who is the median? It's the middle element right now. Our subset S is going to consist of these medians, these n/5 medians from each of the groups and our pivot is the median of this subset S of medians. It's the median of median is our pivot p. Who is the pivot p in this picture? It's the middle element, which is exactly the median of this middle group. This is p.

Now, I want to prove that is a good pivot so I want to first look at which elements are guaranteed to be at most p. Who do I know is the most p? Well, look at this subset S, p is the median of this subset so I know that everybody earlier in this subset … these guys up to and including p itself … are guaranteed to be at most p. So that's n/10 elements that are guaranteed to be at most p.

Now, take this first element. Who's guaranteed to be at most it in its group, G_1? Well, I assume this thing was sorted for the picture purposes. There's two elements, which are guaranteed to be smaller than it. Similarly, in the second group there's two other elements which are guaranteed to be at most this median of this group. In every group, there's two elements, which are guaranteed to be at most this median element.

So what's my conclusion? My conclusion is that all of these elements in this box are guaranteed to be at most the pivot p because their medians are at most this pivot p and these elements are at most this median.

What's the area of this box? How many elements are in here? Well, there's n/10 groups and in each group I got three elements. So there's at least three n/10 elements which are guaranteed to be at most p.

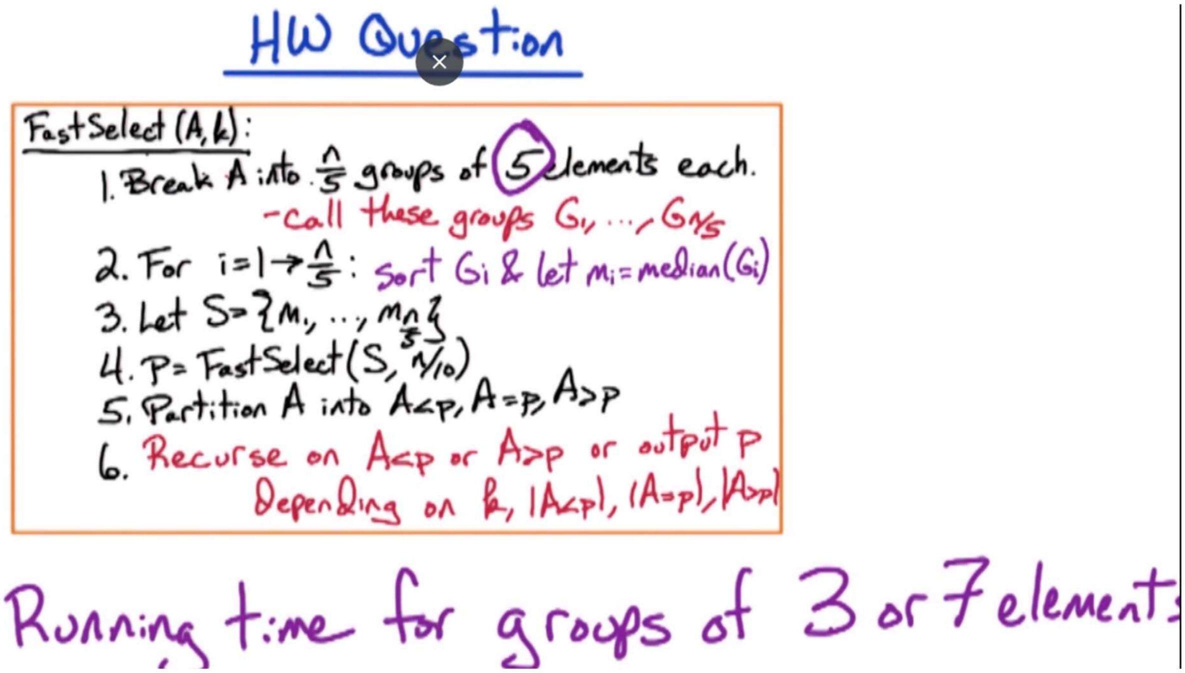
Now look at the partition of A into these subsets: smaller, equal, and bigger than p, and look at the subset consisting of those elements strictly bigger than the pivot p. So this excludes all of those elements which are at most p. So how large is a subset? Well, I've excluded at least 3n/10. So therefore, the size of this is at most 7n/10.

Recall what was my definition of a good pivot? I needed to guarantee that this set is at most 3/4n, and I've shown that it's strictly smaller than at most 0.7n. Now, let's look at the other side.

Which elements are guaranteed to be at least p? Well, now I'm going to look at the other corner.

All of these elements are guaranteed to be at least p. How many elements are here? Well, the same count, 3n/10, are guaranteed to be here. So, the number of elements that are least p is at least 3n/10 and therefore, the number of elements which are strictly less than P is most 7n/10, similarly. That proves that p is a good pivot that completes the proof of the claim.

(Slide 15) HW: Groups of 3? 7?
***



HW Question

```
FastSelect(A,k):
    1. Break A into $\frac{n}{5}$ groups of 5 elements each.
        -call these groups $G_1, \ldots, G_{n/5}$
    2. For $i=1 \to \frac{n}{5}$: sort $G_i$ & let $m_i = \text{median}(G_i)$
    3. Let $S = \{m_1, \ldots, m_{\frac{n}{5}}\}$
    4. $p =$ FastSelect$(S, n/10)$
    5. Partition A into $A_{<p}, A_{=p}, A_{>p}$
    6. Recurse on $A_{<p}$ or $A_{>p}$ or output p
        Depending on k, $|A_{<p}|, |A_{=p}|, |A_{>p}|$
```

Running time for groups of 3 or 7 elements

Now a natural question is, “why do we break A into these groups of five elements each”? Why did we choose five? What would have happened if we would have broken into n/3 groups of size 3 elements each, or n/7 groups of 7 elements each?

This is a nice homework question to make sure you understand the analysis of this algorithm. So consider groups of size 3 each and 7 each, and look at the analysis of the algorithm.

What is the recurrence in these cases where we break into groups of three or seven? Write down the recurrence and see whether that recurrence solves to O(n) or not, and that'll show you why we consider groups of size five.