

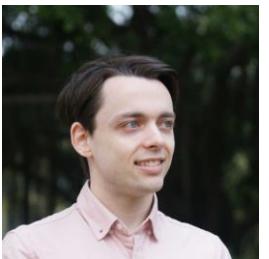
Algorithm Design and Analysis

Introduction, Algorithm Analysis, and Selection

Logistics

Who is 15-451/15-651?

Instructors



Daniel

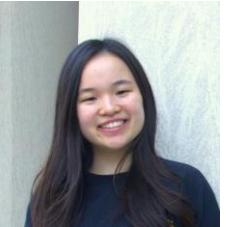


David

Teaching Assistants



Lauren



Emily



Efe



Honghao



Julia



Tanisha

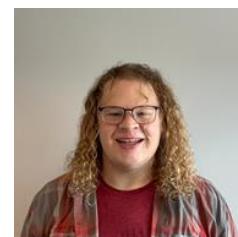
Tutors



Emily



Arturo



Will



Kevin



Thomas

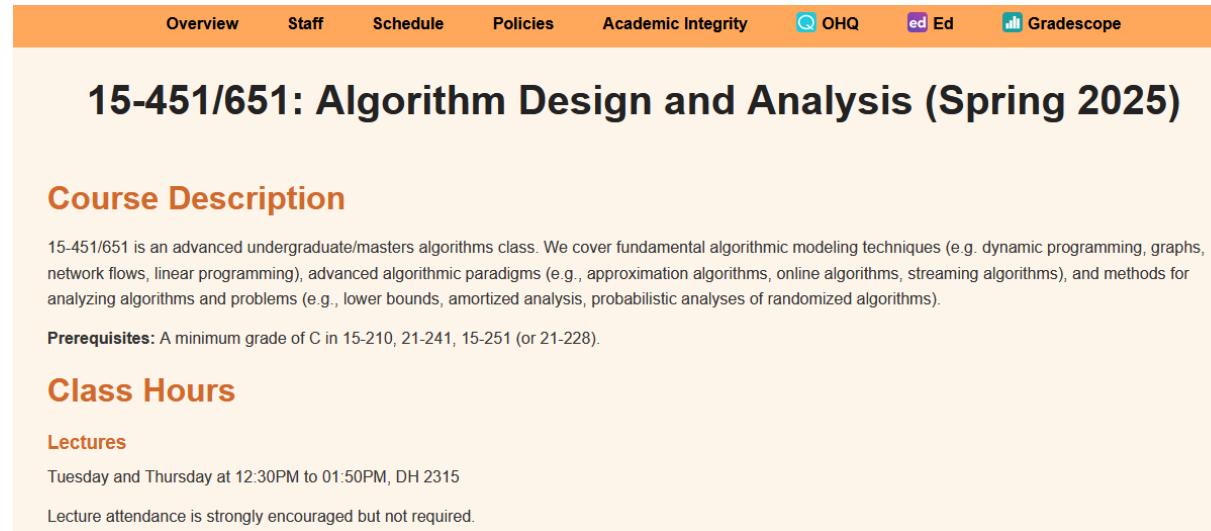


Harrison



Asher

Course website



The screenshot shows the homepage of the course website for 15-451/651: Algorithm Design and Analysis (Spring 2025). The top navigation bar includes links for Overview, Staff, Schedule, Policies, Academic Integrity, OHQ, Ed, and Gradescope. The main title is prominently displayed. Below the title, there is a section for the Course Description, which provides an overview of the class content, including fundamental algorithmic modeling techniques, advanced paradigms, and analysis methods. It also specifies prerequisites (a minimum grade of C in 15-210, 21-241, 15-251, or 21-228). The Class Hours section details the lecture schedule (Tuesday and Thursday at 12:30PM to 01:50PM, DH 2315) and notes that attendance is strongly encouraged but not required.

15-451/651: Algorithm Design and Analysis (Spring 2025)

Course Description

15-451/651 is an advanced undergraduate/masters algorithms class. We cover fundamental algorithmic modeling techniques (e.g. dynamic programming, graphs, network flows, linear programming), advanced algorithmic paradigms (e.g., approximation algorithms, online algorithms, streaming algorithms), and methods for analyzing algorithms and problems (e.g., lower bounds, amortized analysis, probabilistic analyses of randomized algorithms).

Prerequisites: A minimum grade of C in 15-210, 21-241, 15-251 (or 21-228).

Class Hours

Lectures

Tuesday and Thursday at 12:30PM to 01:50PM, DH 2315

Lecture attendance is strongly encouraged but not required.

<https://www.cs.cmu.edu/~15451-s25/>

- Contains course calendar with office hour schedule
- Contains all lecture notes, homework handouts, policies, etc.
- Links to other important platforms (OHQ, Ed, Gradescope)

Homework

- **6 Written Homework:** Each has 2-3 problems. Solutions must be typeset, not handwritten! Submitted to Gradescope.
- **3 Oral Homework:** Collaborate in groups of three and present your solutions to a TA
- **4 Programming Problems:** Submitted via Gradescope.
 - Officially recommended/supported languages are **C++** and **Python**
 - We will accept C, Java, OCaml, Rust when possible (not guaranteed for every assignment, and TA/instructor help will be limited.)

Recitation

- Please review the lecture notes and read the problems beforehand
- Only 50-minutes long so please show up on time!
- 5% of your grade from attendance
- *Two styles of recitation!*
 - **Review-style (R):** More review and goes at a slower pace
 - **Problem-heavy (P):** Faster pace and cover more problems

Midterm exams

- Evening midterms!
- Put these in your calendar right now!

Midterm one (Week 5)

Tuesday February 11th

7:00pm – 9:30pm

Midterm two (Week 10)

Tuesday March 25th

7:00pm – 9:30pm

Tutors

- One-on-one tutoring is available
- ***Highly recommended*** if you struggled with any of the prerequisite courses (15-210, 15-251, 15-122, concepts)
- Open to anyone, space permitting

Academic Integrity

- Healthy collaboration within the rules set out by the assignments is great for learning, but cheating is not.

Definition (Cheating): Includes, but is not limited to:

- Reading/copying answers/code from online sources, books, etc.
- Reading/copying answers of other people, current or former students or otherwise
- Reading/copying answers obtained from generative AI tools (e.g., ChatGPT, CoPilot)
- Accessing problems or solutions from past semesters (e.g., from a friend who took the course in a previous semester, or from a website, etc.)

Detection: We are extremely accurate at catching cheating.

- In Spring 2024, we caught **50-60 cases of cheating** (out of approx. 180 students)
- In Fall 2024, we caught **30-40 cases of cheating** (out of approx. 130 students)

AIV Alternatives

- 451 is a hard course. We know that a lot of AIVs happen out of desperation, not a premeditated plan to cheat.
- If you are struggling with the course, please **sign up for tutoring**, attend **instructor office hours**, and reach out for additional help
- If you are behind and low on time, **request an extension** (see the [extension request form](#) on the course website)
- Try to start your homework early, but if you are stuck at the last minute, we don't close after business hours. **If you reach out for homework help at the last minute, we will almost always try to be available to help.**

Now onto the class!

Formal analysis of algorithms

- We want **provable guarantees** about the properties of algorithms
 - E.g., **prove** that it runs in a certain amount of **time**
 - E.g., **prove** that it outputs the correct **answer**
- **Important question:** How exactly do we measure time?
 - **Answer:** It depends :)
 - Lots more discussion about this in the coming lectures
- We need a ***model of computation!***
 - Specifies exactly what operations are permitted
 - How much each operation costs (sometimes called the *cost model*)

Today's model

The Comparison Model

- Input to the algorithm consists of an array of n items in some order
- The algorithm may perform comparisons (is $a_i < a_j$?) at a **cost of 1**
- Copying/moving items is *free*
- The items are of an arbitrary type. **We are not allowed to assume a type**
 - E.g., the items **can not** be assumed to be numbers
 - This means we can not add, multiply, XOR the items
 - We also can not use hashing, or use elements as array indices, etc.

Quicksort: A journey of algorithm design and analysis

- As seen in 15-122 and 15-210 (and possibly elsewhere!)
- One of the most well-known algorithms in all of computer science

```
function quicksort( $a[0 \dots n - 1]$  : list) {  
    select a pivot element  $p = a_0$   
    let LESS = [ $a_j$  such that  $a_j < p$ ]  
    let GREATER = [ $a_j$  such that  $a_j > p$ ]  
    return quicksort(LESS) + [p] + quicksort(GREATER)  
}
```

Question: What is the *complexity* of Quicksort?

Which measure of complexity?

Definition (Worst-case complexity): The worst-case complexity of an algorithm is the *largest cost* it can incur over *any possible input* (usually as a function of input size n)

Theorem (15-122): The worst-case cost of QuickSort on an input of length n is $\Theta(n^2)$

Note: By default, if not specified, we will pretty much always consider worst-case complexity.

Which measure of complexity?

Definition (Average-case complexity): The average-case complexity of an algorithm is the *average of the costs* of the algorithm over *every possible input*.

Note: Mathematically, this is equivalent to the *expected value of the cost* of the algorithm over an *input chosen uniformly randomly*.

Theorem (15-210): The average cost of QuickSort on an input of length n is $\Theta(n \log n)$

Note: We will rarely, if ever, consider average-case complexity in 15-451/651.

Making it better

- The average-case performance of QuickSort is great
- But its only reliable if the input is random! An evil adversary can always feed our code a worst-case input and ruin our day :(
- Most real-life data **is not random**. Hoping that data is random is **not** a good way to design your algorithms.

Important idea: Instead of hoping that the input is random...
put the randomness *into the algorithm!*

Making it better: *Randomized Quicksort*

```
function random_quicksort( $a[0 \dots n - 1]$  : list) {  
    select a random pivot element  $p = a_i$   
    let LESS = [ $a_j$  such that  $a_j < p$ ]  
    let GREATER = [ $a_j$  such that  $a_j > p$ ]  
    return random_quicksort(LESS) + [p] + random_quicksort(GREATER)  
}
```

Analyzing randomized algorithms

Theorem (15-210): The expected number of comparisons performed by randomized Quicksort on any input of size n is $\Theta(n \log n)$

IMPORTANT NOTES:

Note: When analyzing randomized algorithms, we are usually interested in the *expected value over the random choices* to process a **worst-case user input**

- we are **not** assuming that our random-number generator gives us the worst possible random numbers,
- we are **not** analyzing the algorithm for a randomly chosen input (that's average-case complexity!)

The Quicksort journey so far

Its fast in practice!



Worst-case cost is $\Theta(n^2)$



Average-case cost is $\Theta(n \log n)$



Randomized Quicksort costs
 $\Theta(n \log n)$ in expectation

*What if we could efficiently
find the median element and
use that as the pivot?*

Deterministic Quicksort in
worst-case $\Theta(n \log n)$ cost??

New Problem: Median finding

New problem: Median / k^{th} smallest

Problem (Median) Given a range of distinct elements a_1, a_2, \dots, a_n , output the median.

Definition (Median): The median is the $\left\lfloor \frac{n-1}{2} \right\rfloor^{\text{th}}$ smallest element

- More generally, we can try to solve the “ k^{th} smallest” problem. Given a range of distinct elements and an integer k , we want to find the element such that there are exactly k smaller elements
- k is zero-indexed, so the minimum element is the 0^{th} smallest element

Algorithm design strategy

Algorithm design idea: Start with a simple but inefficient algorithm, then optimize and remove unnecessary steps.

Simple algorithm (k^{th} smallest): Sort the array and output element k

- **Redundancy:** We are finding the k^{th} smallest for **every** k

Take inspiration from Quicksort?

```
function quicksort( $a[0 \dots n - 1]$ ) {  
    select a pivot element  $p = a_i$   
    let LESS = [ $a_j$  such that  $a_j < p$ ]  
    let GREATER = [ $a_j$  such that  $a_j > p$ ]  
    return quicksort(LESS) + [p] + quicksort(GREATER)  
}
```

Question: If we only want the k^{th} number, what is wasteful here?

return quicksort(LESS) + [p] + quicksort(GREATER)

The answer is either in here 

 *Or the answer is in here*

The result: Randomized Quickselect

```
function quickselect( $a[0 \dots n - 1]$ ,  $k$ ) {  
    select a random pivot element  $p = a_i$  for a random  $i$   
    let LESS = [ $a_j$  such that  $a_j < p$ ]  
    let GREATER = [ $a_j$  such that  $a_j > p$ ]  
    if  $k < |LESS|$  then return quickselect( LESS,  $k$  )  
    else if  $k > |LESS|$  then return quickselect( GREATER,  $k - |LESS| - 1$  )  
    else return  $p$   
}
```

Now the analysis

Theorem: The *expected* number of comparisons performed by Quickselect on an input of size n is at most $8n$

Warning: The proof is subtle because it uses probability. We must be careful to not make false assumptions about how probability and randomness work...

Let $T(n) =$ the *expected* number of comparisons performed by Quickselect on a **worst-case input** of size n

$$T(n) = n - 1 + \mathbb{E}(T(x))$$

$x =$ recursive array size (random variable)

First attempt: Almost-correct analysis

Note: This proof is nearly, but not quite correct. It does, however, provide some useful insight that gets us closer to a correct proof.

Question: What is a (good) upper bound on the expected size of the recursive subproblem?

$$\mathbb{E}(x) \leq \frac{3n}{4}$$

So, we might try the recurrence...

$$T(n) \leq n - 1 + T\left(\frac{3n}{4}\right)$$

$$\mathbb{E}(T(x))$$

A better proof

Question: Let's be more precise. How often is the recursive subproblem size at most $3n/4$?



Half !

So, a better recurrence relation is

$$T(n) \leq n-1 + \frac{1}{2} T\left(\frac{3}{4}n\right) + \frac{1}{2} T(n)$$

$$T(n) \leq 2(n-1) + T\left(\frac{3}{4}n\right)$$

Validating the recurrence relation

$$T(n) \leq 2(n - 1) + T\left(\frac{3n}{4}\right)$$

**Exercise: Solve the recurrence or verify through substitution
that the solution satisfies $T(n) \leq 8n$**

Summary of randomized Quickselect

```
function quickselect(a[0 ... n - 1], k) {
    select a random pivot element p = ai for a random i
    let LESS = [aj such that aj < p]
    let GREATER = [aj such that aj > p]

    if |LESS| ≥ k then return quickselect(LESS, k)
    else if |LESS| = k then return p
    else return quickselect(RIGHT, k - |LESS| - 1)
}
```

- Runs in $O(n)$ **expected** time in the **comparison model**.
- More tightly, uses at most $8n$ comparisons in expectation.
- As an exercise, the analysis can be improved to $4n$ comparisons.

Have we achieved our goal?

- Use Quickselect to select the pivot for Quicksort
- Guaranteed best-case recursion for Quicksort
- Problem?
 - Randomized Quickselect is still... randomized.
 - So, Quicksort would still be $O(n \log n)$ randomized, not deterministic

We want a deterministic algorithm!!

- Where was the randomness in Randomized QuickSelect? How can we get rid of it?
- What if we could deterministically find the optimal pivot? What would that be? The median! Oh...

What we need: In $O(n)$ comparisons, we need to find a “good” pivot. A good pivot would leave us with cn elements in the recursive call, for some fraction $c < 1$, e.g., $\frac{3n}{4}$ elements is good.

Picking a good pivot

- Picking the median as the pivot is too much to ask for, so we want some kind of “approximate median”

Idea (doesn't quite work, but very close): Pick the median of a smaller subset of the input (faster to find) then hope that it is a good approximation to the true median.

Question: What if we find the median of half of the elements? How good of a pivot is this element?

$$\frac{31}{4}$$

Median of half

If we pivot on the median of half of the elements, the number of comparisons will be

$$T(n) \leq n - 1 + \underbrace{T\left(\frac{3n}{4}\right)}_{\text{left}} + \underbrace{T\left(\frac{n}{2}\right)}_{\text{right}}$$

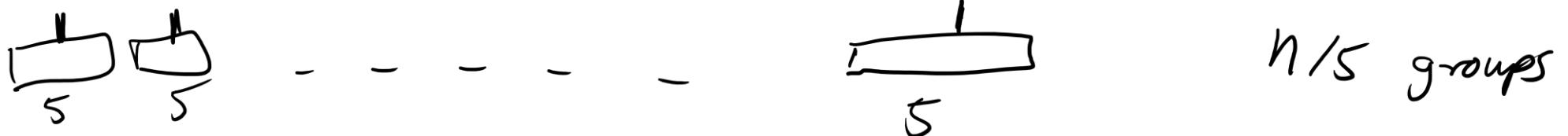
Exercise: Show that picking any constant-fraction sized subset (e.g., a quarter, one tenth) and taking the median doesn't work.

We need to go deeper!

Note: This idea is extremely subtle. It took four Turing Award winners to figure it out. We don't expect that you would produce this algorithm on your own.

- Finding the median of a smaller set **almost** worked, but it was just a bit too much work since the “approximate median” wasn’t good enough.

Huge idea (median of medians): Find the medians of several small subsets of the input, then find the **median of those medians**.

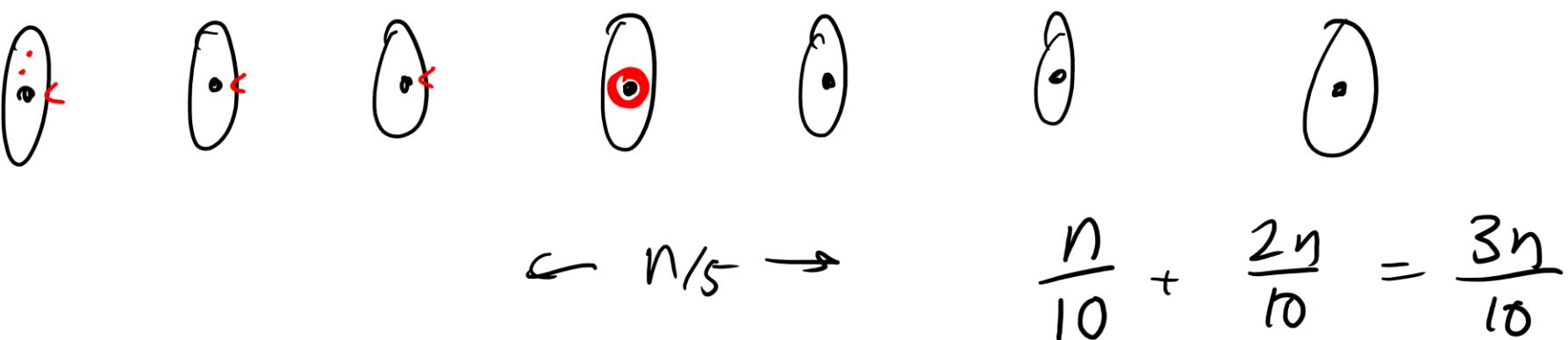


Median of medians algorithm

```
function DeterministicSelect( $a[0 \dots n - 1]$ ,  $k$ ) {  
    group the array into  $n/5$  groups of size 5, find the median of each group  
    recursively find the median of these medians, call it  $p$   
  
    // Below is the same as Randomized Quickselect  
    let LESS = [ $a_j$  such that  $a_j < p$ ]  
    let GREATER = [ $a_j$  such that  $a_j > p$ ]  
    if  $|LESS| \geq k$  then return DeterministicSelect(LESS,  $k$ )  
    else if  $|LESS| = k$  then return  $p$   
    else return DeterministicSelect(GREATER,  $k - |LESS| - 1$ )  
}
```

How good is the median of medians?

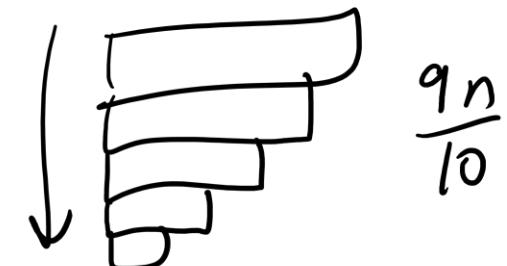
Theorem: The median of medians is larger than at least $3/10^{\text{ths}}$ of the input, and smaller than at least $3/10^{\text{ths}}$ of the input

$$\begin{array}{c} \uparrow \\ 5 \\ \downarrow \end{array} \quad \begin{array}{ccccccc} \textcircled{1} & \textcircled{2} & \textcircled{3} & \textcircled{4} & \textcircled{5} & \textcircled{6} & \textcircled{7} \end{array} \quad \leftarrow \frac{n}{5} \rightarrow \quad \frac{n}{10} + \frac{2n}{10} = \frac{3n}{10}$$


Analysis of DeterministicSelect

Theorem: The number of comparisons performed by DeterministicSelect on an input of size n is $O(n)$

1. Find the median of $n/5$ groups of size 5 $O(n)$
2. Recursively find the median of medians $T(\frac{n}{5})$
3. Split the input into LESS and GREATER $n - 1$
4. Recurse on the appropriate piece $T(\frac{7n}{10})$



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

Solving the recurrence

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

**Exercise: Solve the recurrence using your preferred technique
(e.g., unrolling, stack of bricks)**

So, the total running time is...

$$T(n) \leq cn(1 + (9/10) + (9/10)^2 + (9/10)^3 + \dots)$$

Applying the formula for a geometric series, we get $T(n) \leq 10cn = O(n)$

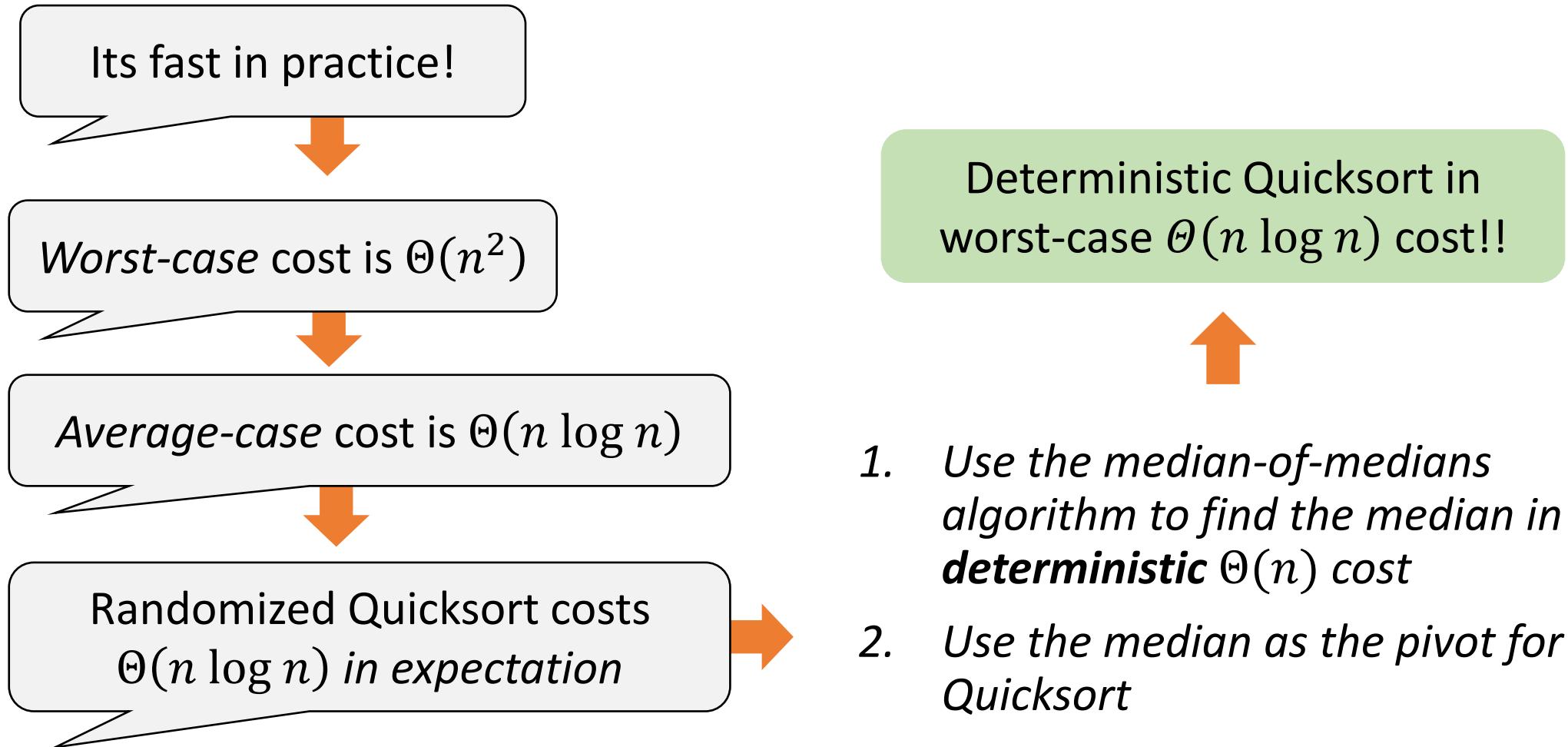
Summary of DeterministicSelect

```
function DeterministicSelect( $a[0 \dots n - 1]$ ,  $k$ ) {
    group the array into  $n/5$  groups of size 5,
        find the median of each group
    recursively find the median of these medians, call it  $p$ 

    // Below is the same as Randomized Quickselect
    let LESS = [ $a_j$  such that  $a_j < p$ ]
    let GREATER = [ $a_j$  such that  $a_j > p$ ]
    if  $|LESS| \geq k$  then return DeterministicSelect(LESS,  $k$ )
    else if  $|LESS| = k$  then return  $p$ 
    else return DeterministicSelect(GREATER,  $k - |LESS| - 1$ )
}
```

- The **median of medians** is the key ingredient for getting a deterministic algorithm
- To analyze the recurrence, we used the “stack of bricks” method.
- We could also prove it by induction, but this requires us to know the runtime already

The Quicksort journey



Take-home messages for today

- There's more to Quicksort than you think!
- Recursion is powerful, randomization is powerful.
- Analyzing *randomized recursive algorithms* is tricky. Be careful with expected values!!
- Analyzing runtime via *recurrence relations* is very useful.