CSC373

Algorithm Design, Analysis & Complexity

Nisarg Shah

Introduction

- Instructor: Nisarg Shah (me)
  - www.cs.toronto.edu/~nisarg, SF 3312 (only drop by after making an appointment)
  - Email: csc373-2023-09@cs.toronto.edu
  - LEC 0101 and 0201
- TAs: Too many to list
- Who will do what?
  - I'll deliver the lectures and hold office hours
  - TAs will deliver the tutorials and grade your work
  - TAs and I will collectively address remark requests

Course Information

- Course Page www.cs.toronto.edu/~nisarg/teaching/373f23/
- Discussion Board piazza.com/utoronto.ca/fall2023/csc373
- Grading: markus.teach.cs.toronto.edu
  - LaTeX preferred, scans are OK!
- All times will be in the Eastern time zone

Lectures, Tutorials, Office Hours

- See the course web page for times and locations of lectures and tutorials

- Office hours:
  - Monday: 1:30-2:30pm
  - Friday: 1-2pm
  - Location:
    - SF 3312 (my office)
    - In weeks where I expect many students to show up, I'll book a bigger seminar room
    - Occasionally, Friday's office hour may need to shift to Zoom, but I'll announce in advance

Lecture Format

- Delivered by me

- Will start at 10 minutes past the hour
  - 10-minute break after 50 minutes of lecture in the 2-hour slot

- Ask questions by raising your hand

Tutorial Format

- Delivered by the TAs

- Think of them as preparation for assignments/exams
  - Some of the tutorial problems may be easier than assignment/exam questions

- Problem sets & solutions
  - Problem sets will be posted to the course webpage in advance of the tutorial
  - Solutions will be posted to the course webpage after the tutorial

- What to do
  - Please attempt the problems before coming to the tutorials
  - During the tutorials, the TAs will go over the solutions and explain key ideas

Tutorial Format

- Further details
  - There are two tutorial subsections in each section of the course (A,B)
  - You can find the room & time information on the course web page
  - Feel free to attend any tutorial subsection of your choice
    - *Except on two days when the tutorial slots will be used to conduct a midterm*
    - *See the next slide*

Tests

- 2 midterms (20% each, 40% total), one final exam (25%)
  - I'll post practice exams from prior years before each test
- Midterms (check the syllabus for dates):
  - Two slots: Friday 11-13 & Friday 14-16
  - **LEC 0101 writes during 11-13, LEC 0201 writes during 14-16**
  - If you have a conflict with your own slot and want to write the midterm in the other slot (or request an alternate time), you must reach out to me AT LEAST 1 WEEK prior to the midterm and request it

Assignments

- 4 assignments, best 3 out of 4, 10% each (30% total)
- Group work
  - In groups of up to three students
  - Best way to learn is for each member to try each problem
- Questions will be more difficult
  - May need to mull them over for several days; do *not* expect to start and finish the assignment on the same day!
  - May include bonus questions
- Submission (and later remark requests) on MarkUs
  - May need to compress the PDF

Late Days

- 4 total late days across all 4 assignments
  - Managed by MarkUs
  - At most 2 late days can be applied to a single assignment
  - Already covers legitimate reasons such as illness, university activities, etc.
  - Petitions will only be granted for circumstances which cannot be covered by this
- If you are registered with Accessibility Services, send me your letter early
  - If a midterm is on a Friday following Sunday night's assignment deadline, you may only be granted until EOD on Tuesday (without any late days charged) as I'll need to release solutions on Wednesday morning

Embedded EthiCS Module

- Goal
  - Help you learn how to reason about ethical issues, practice conveying your thoughts on such issues
  - In the context of a topic from the course

- During the 2-hour lecture slot on Dec 6 (final lecture)
  - A lightweight survey before and after the module (0.5% each)
  - A lightweight assignment before and after the module (2% each)
  - Discussion-based group activities during the module

Grading Policy

• Best 3/4 homeworks	*	10%	=	30%
• 2 midterms	*	20%	=	40%
• EthiCS Module	*	5%	=	5%
• Final exam	*	25%	=	25%

- NOTE: If you score less than 40% on the final exam, your overall course marks may be reduced below 50

Approximate Due Dates

- Assignment 1: Oct 8
- Assignment 2: Oct 29
- Assignment 3: Nov 19
- Assignment 4: Dec 7
- Midterm 1: Nov 3
- Midterm 2: Nov 24

Textbook

- Primary reference: lecture slides

- Primary textbook
  - [CLRS] Cormen, Leiserson, Rivest, Stein: *Introduction to Algorithms*.

- Supplementary textbooks (optional)
  - [DPV] Dasgupta, Papadimitriou, Vazirani: *Algorithms*.
  - [KT] Kleinberg; Tardos: *Algorithm Design*.
  - [RG] Roughgarden: Algorithms Illuminated.
  - Check the info page of the course website ☺

Other Policies

- Collaboration
  - Free to discuss with classmates or read online material
  - Must write solutions in your own words
    - Easier if you do not take any pictures/notes from discussions
- Citation
  - For each question, must cite the peer (write the name) or the online sources (provide links), if you obtained a significant insight directly pertinent to the question
  - Failing to do this is plagiarism!

Other Policies

- “No Garbage” Policy
  - Borrowed from: Prof. Allan Borodin (citation!)
  - Applies to all (sub)questions in assignments and tests, except for any bonus (sub)questions

1. Partial marks for viable approaches
2. Zero marks if the answer makes no sense
3. 20% marks if you admit to not knowing how to approach the question (“I do not know how to approach this question”)

- 20% > 0% !!

Questions?

Enough with the boring stuff.

What will we study?

Why will we study it?

Muhammad ibn Musa al-Khwarizmi
c. 780 – c. 850

What is this course about?

- Algorithms
  - Ubiquitous in the real world
    - From your smartphone to self-driving cars
    - From graph problems to graphics problems
    - …
  - Important to be able to design and analyze algorithms
  - For some problems, good algorithms are hard to find
    - For some of these problems, we can formally establish complexity results
    - We'll often find that one problem is easy, but its minor variants are suddenly hard

What is this course about?

- Algorithms
  - Algorithms in specialized environments or using advanced techniques
    - Distributed, parallel, streaming, sublinear time, spectral, genetic…
  - Other concerns with algorithms
    - Fairness, ethics, …
  - …mostly beyond the scope of this course

What is this course about?

- Designing fast algorithms
  - Divide and Conquer
  - Greedy
  - Dynamic programming
  - Network flow
  - Linear programming
- Proving that no fast algorithms are likely possible
  - Reductions & NP-completeness
- What to do if no fast algorithms are likely possible
  - Approximation algorithms (if time permits)
  - Randomized algorithms (if time permits)

What is this course about?

- How do we know which paradigm is right for a given problem?
  - A very interesting question!
  - Subject of much ongoing research…
    - Sometimes, you just know it when you see it…
- How do we analyze an algorithm?
  - Proof of correctness
  - Proof of running time
    - We'll try to prove the algorithm is *efficient* in the *worst case*
    - In practice, average case matters just as much (or even more)

What is this course about?

- What does it mean for an algorithm to be efficient in the worst case?
  - Polynomial time
  - It should use at most poly(n) steps on *any* n-bit input
    - $n, n^2, n^{100}, 100n^6 + 237n^2 + 432, \ldots$
  - If the input to an algorithm is a number x, the number of bits of input is $\log x$
    - This is because it takes $\log x$ bits to represent the input x in binary
    - So, the running time should be polynomial in $\log x$, not in x
  - How much is too much?

What is this course about?

Picture-Hanging Puzzles*

Erik D. Demaine[†] Martin L. Demaine[†] Yair N. Minsky[‡] Joseph S. B. Mitchell[§]
Ronald L. Rivest[†] Mihai Pătraşcu[¶]

Theorem 7 *For any $n \geq k \geq 1$, there is a picture hanging on n nails, of length $n^{c'}$ for a constant c', that falls upon the removal of any k of the nails.*

$n^{6{,}100 \log_2 c}$. Using the $c \leq 1{,}078$ upper bound, we obtain an upper bound of $c' \leq 6{,}575{,}800$. Using

So, while this construction is polynomial, it is a rather large polynomial. For small values of n, we can use known small sorting networks to obtain somewhat reasonable constructions.

What is this course about?

Better Balance by Being Biased:
A 0.8776-Approximation for Max Bisection

Per Austrin[*], Siavosh Benabbas[*], and Konstantinos Georgiou[†]

has a lot of flexibility, indicating that further improvements may be possible. We remark that, while polynomial, the running time of the algorithm is somewhat abysmal; loose estimates places it somewhere around $O\left(n^{10^{100}}\right)$; the running time of the algorithm of [RT12] is similar.

What is this course about?

- What if we can’t find an efficient algorithm for a problem?
  - Try to prove that the problem is hard
  - Formally establish complexity results
  - NP-completeness, NP-hardness, …

- We’ll often find that one problem may be easy, but its simple variants may suddenly become hard
  - Minimum spanning tree (MST) vs bounded degree MST
  - 2-colorability vs 3-colorability

I’m not convinced.

Will I really ever need to know how to design abstract algorithms?

At the very least…

This will help you prepare for your technical job interview!

Real Microsoft interview question:

- Given an array a, find indices (i, j) with the largest $j - i$ such that $a[j] > a[i]$
- Greedy? Divide & conquer?

Disclaimer

- The course is theoretical in nature
  - You'll be working with abstract notations, proving correctness of algorithms, analyzing the running time of algorithms, designing new algorithms, and proving complexity results.

- Something for everyone…
  - If you're somewhat scared going into the course
  - If you're already comfortable with the proofs, and want challenging problems

Related/Follow-up Courses

- Direct follow-up
  - CSC473: Advanced Algorithms
  - CSC438: Computability and Logic
  - CSC463: Computational Complexity and Computability
- Algorithms in other contexts
  - CSC304: Algorithmic Game Theory and Mechanism Design (self promotion!)
  - CSC384: Introduction to Artificial Intelligence
  - CSC436: Numerical Algorithms
  - CSC418: Computer Graphics

Divide & Conquer

History?

- Maybe you saw a subset of these algorithms?
  - Mergesort - $O(n \log n)$
  - Karatsuba algorithm for fast multiplication - $O\left(n^{\log_2 3}\right)$ rather than $O(n^2)$
  - Largest subsequence sum in $O(n)$
  - …
- Have you seen some divide & conquer algorithms before?
  - Maybe in CSC236/CSC240 and/or CSC263/CSC265

Divide & Conquer

- General framework
  - Break (a large chunk of) a problem into two smaller subproblems of the same type
  - Solve each subproblem recursively and independently
  - At the end, quickly combine solutions from the two subproblems and/or solve any remaining part of the original problem

- Hard to formally define when a given algorithm is divide-and-conquer…
- Let’s see some examples!

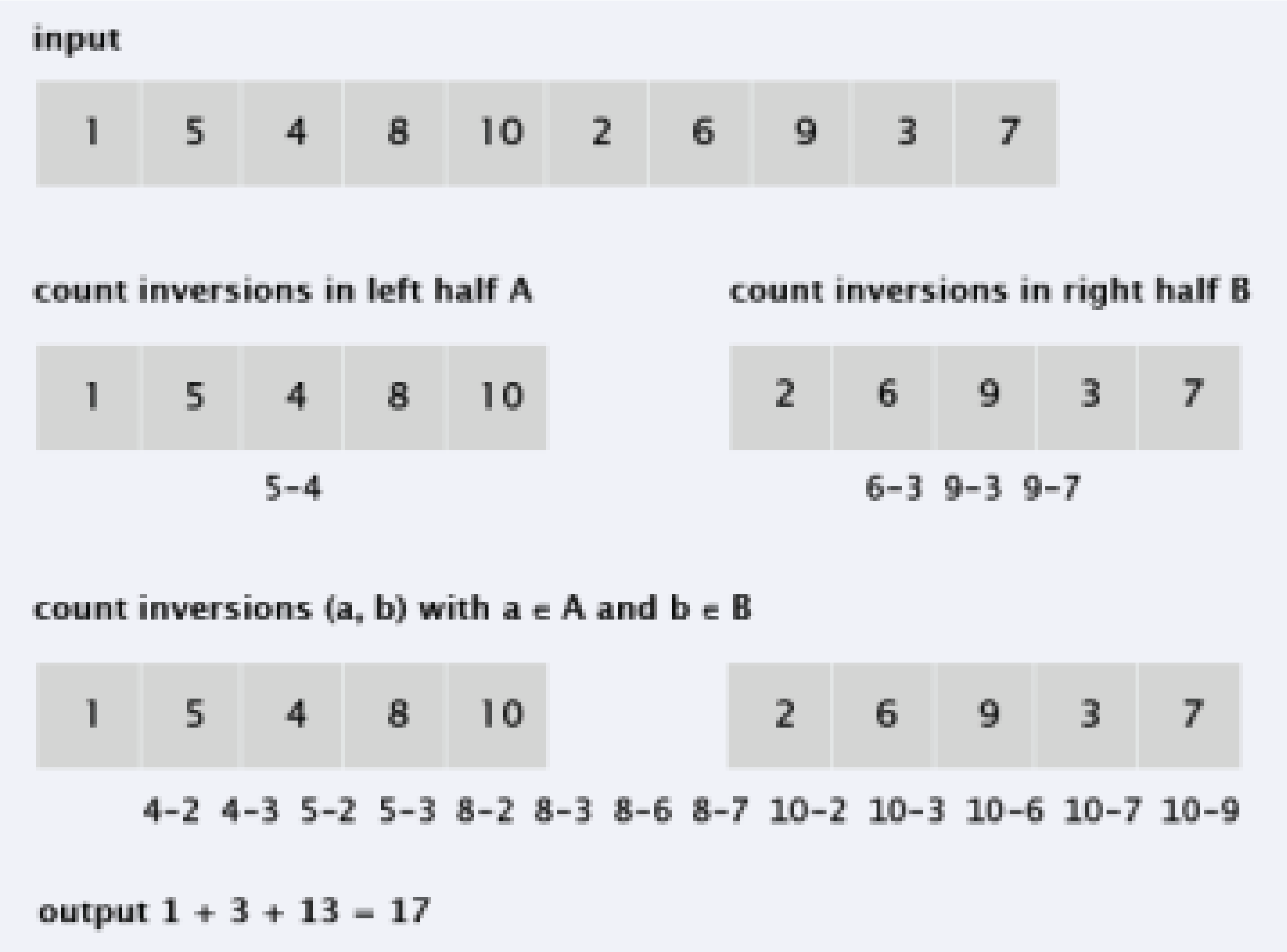
Counting Inversions

- Problem
  - Given an array a of length n, count the number of pairs (i, j) such that $i < j$ but $a[i] > a[j]$
- Applications
  - Voting theory
  - Collaborative filtering
  - Measuring the "sortedness" of an array
  - Sensitivity analysis of Google's ranking function
  - Rank aggregation for meta-searching on the Web
  - Nonparametric statistics (e.g., Kendall's tau distance)

Counting Inversions

- Problem
  - Count (i, j) such that $i < j$ but $a[i] > a[j]$
- Brute force
  - Check all $\Theta(n^2)$ pairs
- Divide & conquer
  - Divide: break array into two equal halves x and y
  - Conquer: count inversions in each half recursively
  - Combine:
    - Solve (we'll see how): count inversions with one entry in x and one in y
    - Merge: add all three counts

Counting Inversions



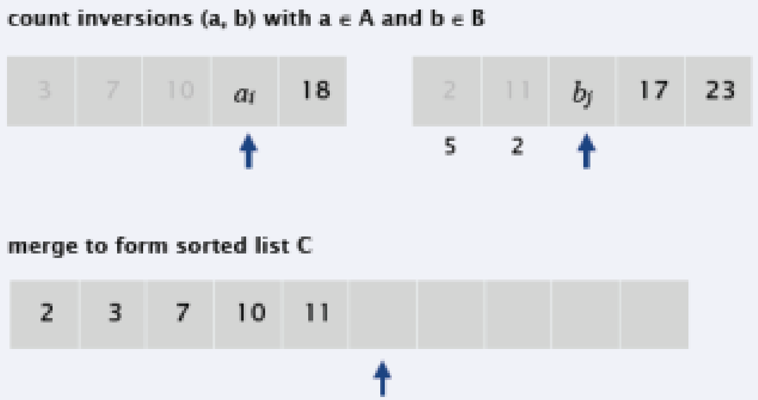
Courtesy: Kevin Wayne

Counting Inversions

Q. How to count inversions (a, b) with $a \in A$ and $b \in B$?
A. Easy if A and B are sorted!

Count inversions (a, b) with $a \in A$ and $b \in B$, assuming A and B are sorted.

- Scan A and B from left to right.
- Compare a_i and b_j.
- If $a_i < b_j$, then a_i is not inverted with any element left in B.
- If $a_i > b_j$, then b_j is inverted with every element left in A.
- Append smaller element to sorted list C.



Courtesy: Kevin Wayne

Counting Inversions

SORT-AND-COUNT (L)

IF list L has one element
  RETURN $(0, L)$.

DIVIDE the list into two halves A and B.
$(r_A , A) \leftarrow$ SORT-AND-COUNT(A).
$(r_B , B) \leftarrow$ SORT-AND-COUNT(B).
$(r_{AB} , L') \leftarrow$ MERGE-AND-COUNT(A, B).

RETURN $(r_A + r_B + r_{AB} , \ L')$.

Courtesy: Kevin Wayne

Counting Inversions

- How do we formally prove correctness?
  - (Strong) Induction on n is usually very helpful
    - Assume that the algorithm correctly solves problems of size strictly smaller than n
    - Thus, the algorithm, when applied recursively on the two halves, correctly sorts them & counts inversions within them
    - Just need to prove correctness of the "Combine" step (and argue the base case)

- Running time analysis
  - Suppose $T(n)$ is the worst-case running time for inputs of size n
  - Our algorithm satisfies $T(n) \leq 2\, T(n/2) + O(n)$
  - Master theorem says this is $T(n) = O(n \log n)$
    - Pictorial proof!

Master Theorem

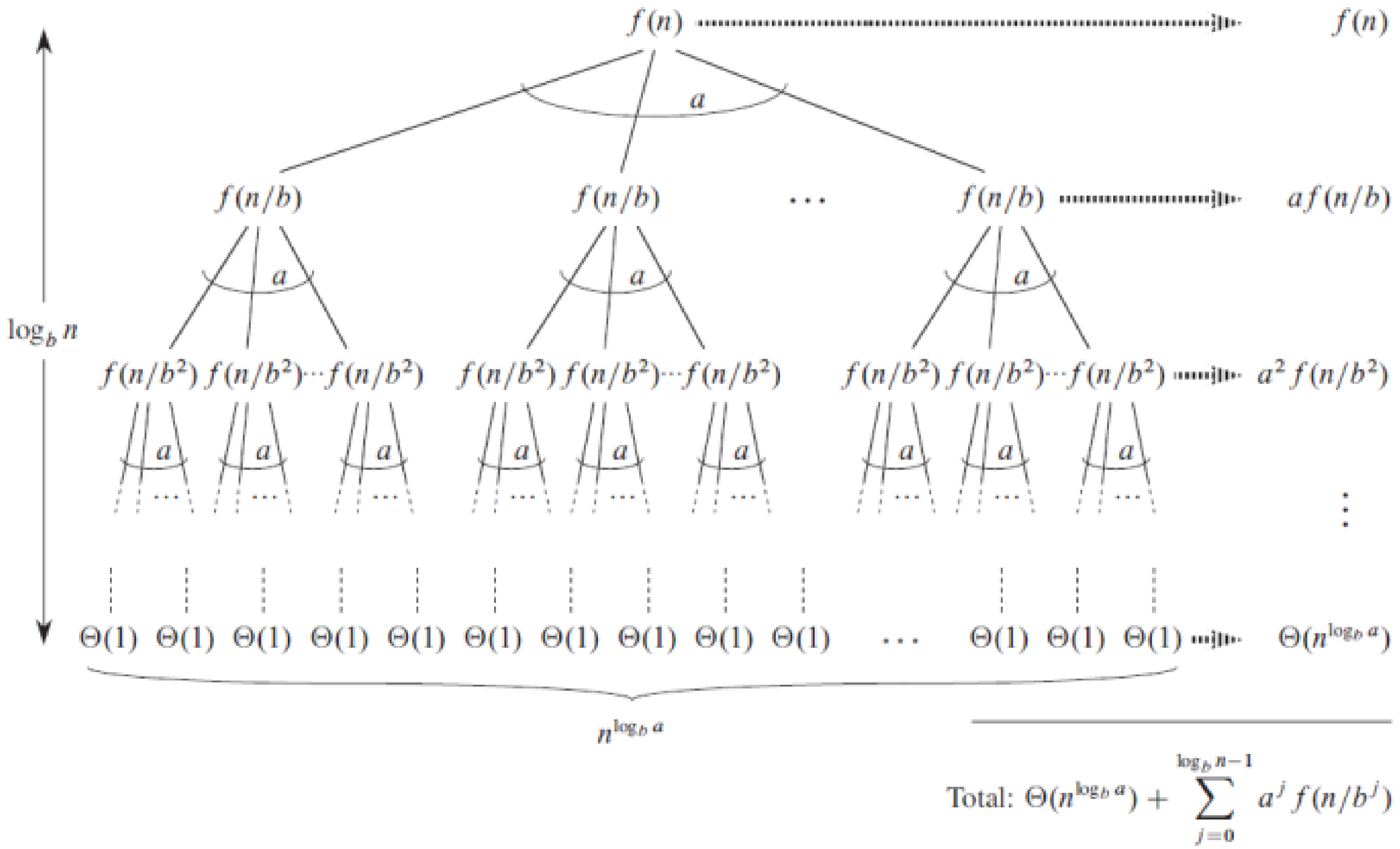
- Here's the master theorem
  - Useful for analyzing divide-and-conquer running time
  - If you haven't already seen it, please spend some time understanding it

  - Theorem: Let $a \geq 1$ and $b > 1$ be constants, $f(n)$ be a function, and $T(n)$ be defined on non-negative integers by the recurrence $T(n) \leq a \cdot T\left(\frac{n}{b}\right) + f(n)$, where n/b can be $\left\lceil \frac{n}{b} \right\rceil$.
    Let $d = \log_b a$. Then:

    - If $f(n) = O\left(n^{d-\epsilon}\right)$ for some constant $\epsilon > 0$, then $T(n) = O\left(n^d\right)$.
    - If $f(n) = O\left(n^d \log^k n\right)$ for some $k \geq 0$, then $T(n) = O\left(n^d \log^{k+1} n\right)$.
    - If $f(n) = O\left(n^{d+\epsilon}\right)$ for some constant $\epsilon > 0$, then $T(n) = O\left(f(n)\right)$.

Master Theorem

Intuition: Compare *f(n)* with $n^{\log_b a}$. The larger determines the recurrence solution.



Closest Pair in $\mathbb{R}^2$

- Problem:
  - Given n points of the form (x_i, y_i) in the plane, find the closest pair of points.

- Applications:
  - Basic primitive in graphics and computer vision
  - Geographic information systems, molecular modeling, air traffic control
  - Special case of nearest neighbor

- Brute force: $\Theta(n^2)$

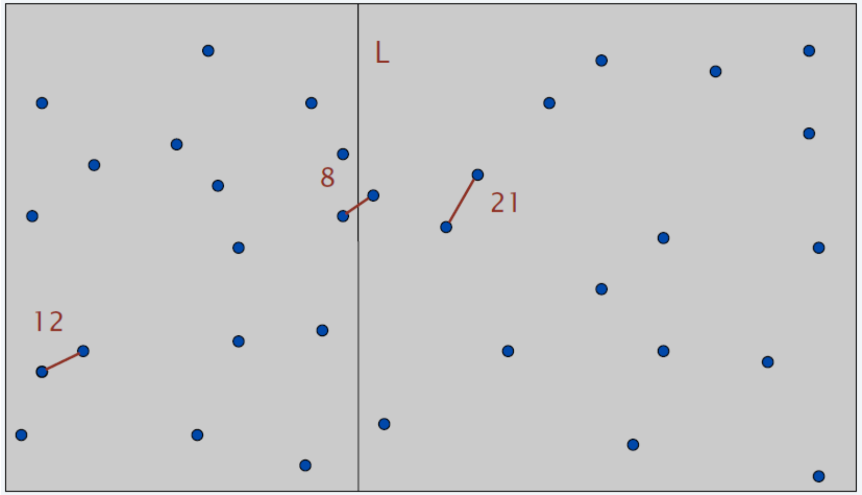
Intuition from 1D?

- In 1D, the problem would be easily $O(n \log n)$
  - Sort and check!
- Sorting attempt in 2D
  - Find closest points by x coordinate
  - Find closest points by y coordinate
  - Doesn't work! (Exercise: come up with a counterexample)
- Non-degeneracy assumption
  - No two points have the same x or y coordinate

Closest Pair in $\mathbb{R}^2$

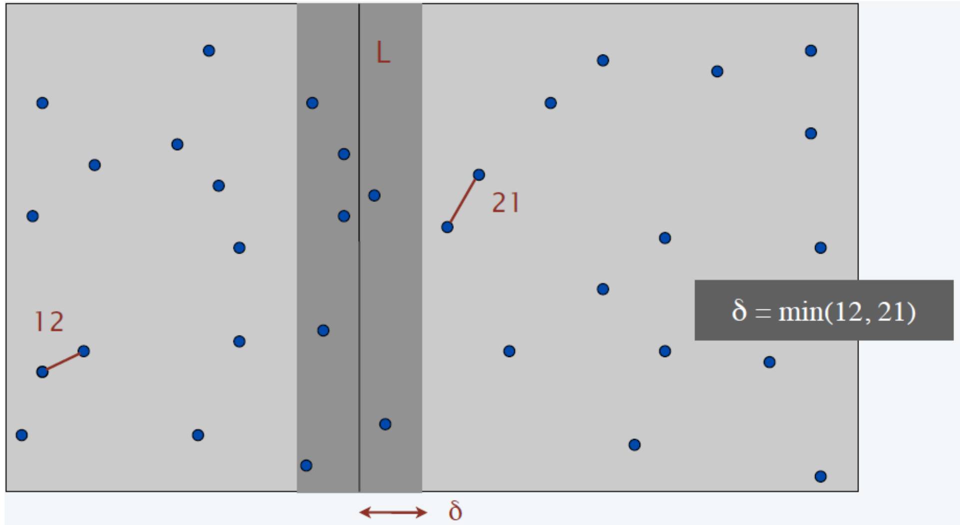
- Let's try divide-and-conquer!
  - Divide: points in equal halves by drawing a vertical line L
  - Conquer: solve each half recursively
  - Combine: find closest pair with one point on each side of L
  - Return the best of 3 solutions

Seems like $\Omega(n^2)$ ☹



Closest Pair in $\mathbb{R}^2$

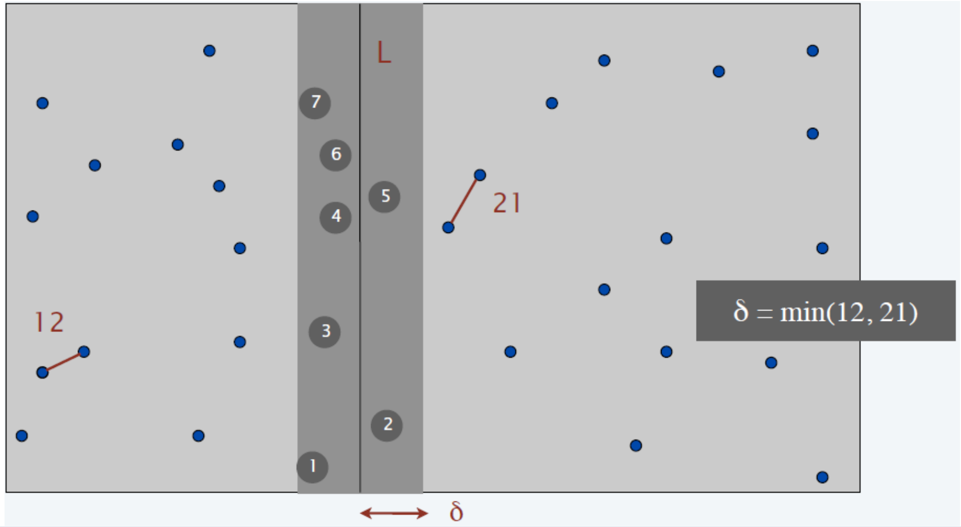
- Combine
  - We can restrict our attention to points within δ of L on each side, where δ = best of the solutions within the two halves



Closest Pair in $\mathbb{R}^2$

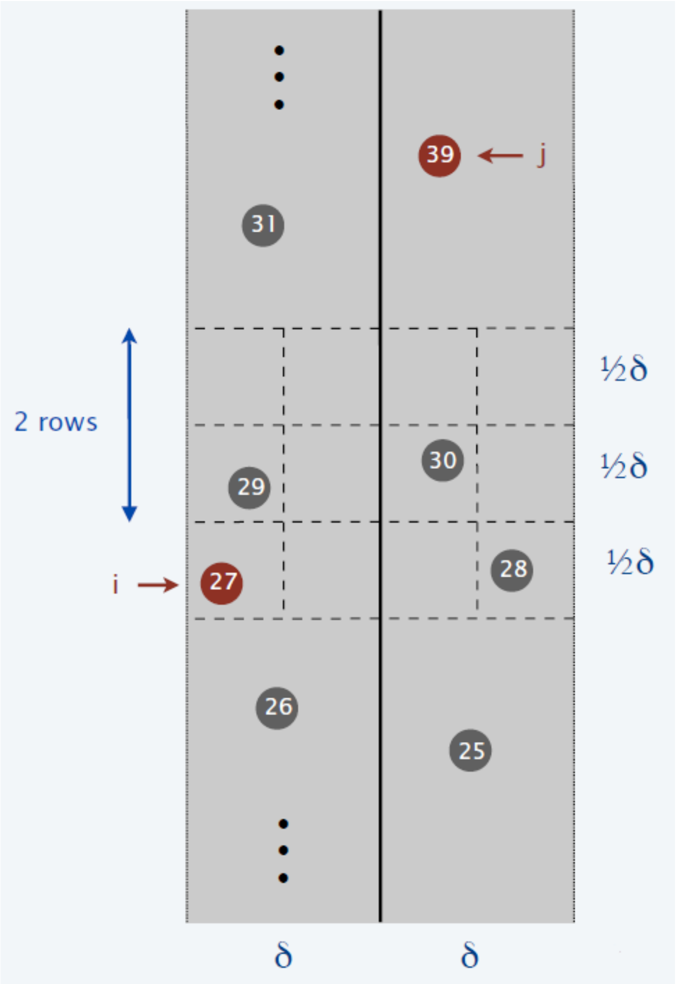
- Combine (let δ = best of solutions in two halves)
  - Only need to look at points within δ of L on each side,
  - Sort points on the strip by y coordinate
  - Only need to check each point with next 11 points in sorted list!

Wait, what? Why 11?



Why 11?

- Claim:
  - If two points are at least 12 positions apart in the sorted list, their distance is at least δ
- Proof:
  - No two points lie in the same $\delta/2 \times \delta/2$ box
  - Two points that are more than two rows apart are at distance at least δ



Running Time Analysis

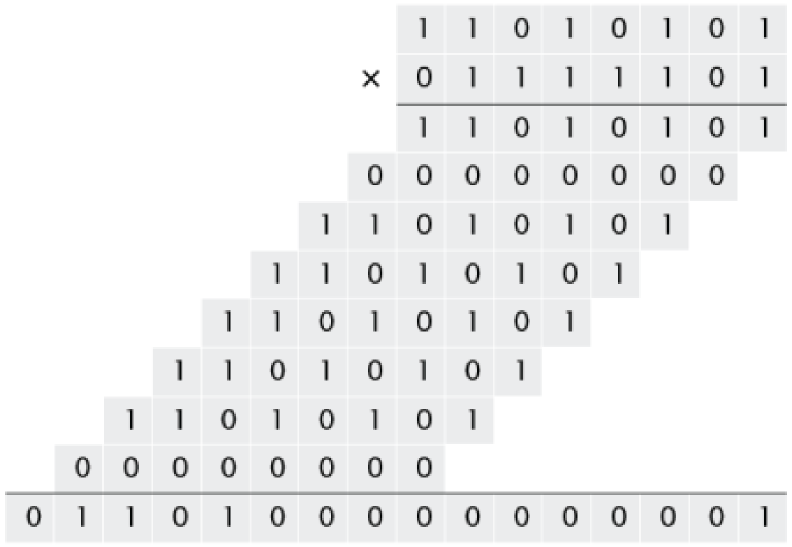
- Running time for the combine operation
  - Finding points on the strip: $O(n)$
  - Sorting points on the strip by their y-coordinate: $O(n \log n)$
  - Testing each point against 11 points: $O(n)$

- Total running time: $T(n) \leq 2T\left(\frac{n}{2}\right) + O(n \log n)$

- By the Master theorem, this yields $T(n) = O(n \log^2 n)$
  - Can be improved to $O(n \log n)$ by doing a single global sort by y-coordinate at the beginning

Recap: Karatsuba's Algorithm

- Fast way to multiply two n digit integers x and y
- Brute force: $O(n^2)$ operations
- Karatsuba's observation:
  - Divide each integer into two parts
    - $x = x_1 * 10^{n/2} + x_2, y = y_1 * 10^{n/2} + y_2$
    - $xy = (x_1 y_1) * 10^n + (x_1 y_2 + x_2 y_1) * 10^{n/2} + (x_2 y_2)$
  - Four $n/2$-digit multiplications can be replaced by three
    - $x_1 y_2 + x_2 y_1 = (x_1 + x_2)(y_1 + y_2) - x_1 y_1 - x_2 y_2$
  - Running time
    - $T(n) \leq 3\,T(n/2) + O(n) \Rightarrow T(n) = O\left(n^{\log_2 3}\right)$



Strassen's Algorithm

- Generalizes Karatsuba's insight to design a fast algorithm for multiplying two $n \times n$ matrices
  - Call n the "size" of the problem

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} * \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

  - Naively, this requires 8 multiplications of size $n/2$
    - $A_{11} * B_{11}, A_{12} * B_{21}, A_{11} * B_{12}, A_{12} * B_{22}, \ldots$
  - Strassen's insight: replace 8 multiplications by 7
    - Running time: $T(n) \leq 7\, T(n/2) + O(n^2) \Rightarrow T(n) = O\left(n^{\log_2 7}\right)$

Strassen's Algorithm

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} * \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

STRASSEN (n, A, B)

IF $(n = 1)$ RETURN $A \times B$.

assume n is a power of 2

Partition A and B into 2-by-2 block matrices.

keep track of indices of submatrices (don't copy matrix entries)

$P_1 \leftarrow$ STRASSEN $(n/2, A_{11}, (B_{12} - B_{22}))$.

$P_2 \leftarrow$ STRASSEN $(n/2, (A_{11} + A_{12}), B_{22})$.

$P_3 \leftarrow$ STRASSEN $(n/2, (A_{21} + A_{22}), B_{11})$.

$P_4 \leftarrow$ STRASSEN $(n/2, A_{22}, (B_{21} - B_{11}))$.

$P_5 \leftarrow$ STRASSEN $(n/2, (A_{11} + A_{22}) \times (B_{11} + B_{22}))$.

$P_6 \leftarrow$ STRASSEN $(n/2, (A_{12} - A_{22}) \times (B_{21} + B_{22}))$.

$P_7 \leftarrow$ STRASSEN $(n/2, (A_{11} - A_{21}) \times (B_{11} + B_{12}))$.

$C_{11} = P_5 + P_4 - P_2 + P_6$.

$C_{12} = P_1 + P_2$.

$C_{21} = P_3 + P_4$.

$C_{22} = P_1 + P_5 - P_3 - P_7$.

RETURN C.

Median & Selection

- Selection:
  - Given array A of n comparable elements, find kth smallest
  - $k = 1$ is min, $k = n$ is max, $k = \lfloor (n+1)/2 \rfloor$ is median
  - $O(n)$ is easy for min/max
- What about k-selection?
  - $O(nk)$ by modifying bubble sort
  - $O(n \log n)$ by sorting
  - $O(n + k \log n)$ using min-heap
  - $O(k + n \log k)$ using max-heap
- Q: What about just $O(n)$?
- A: Yes! Selection is easier than sorting.

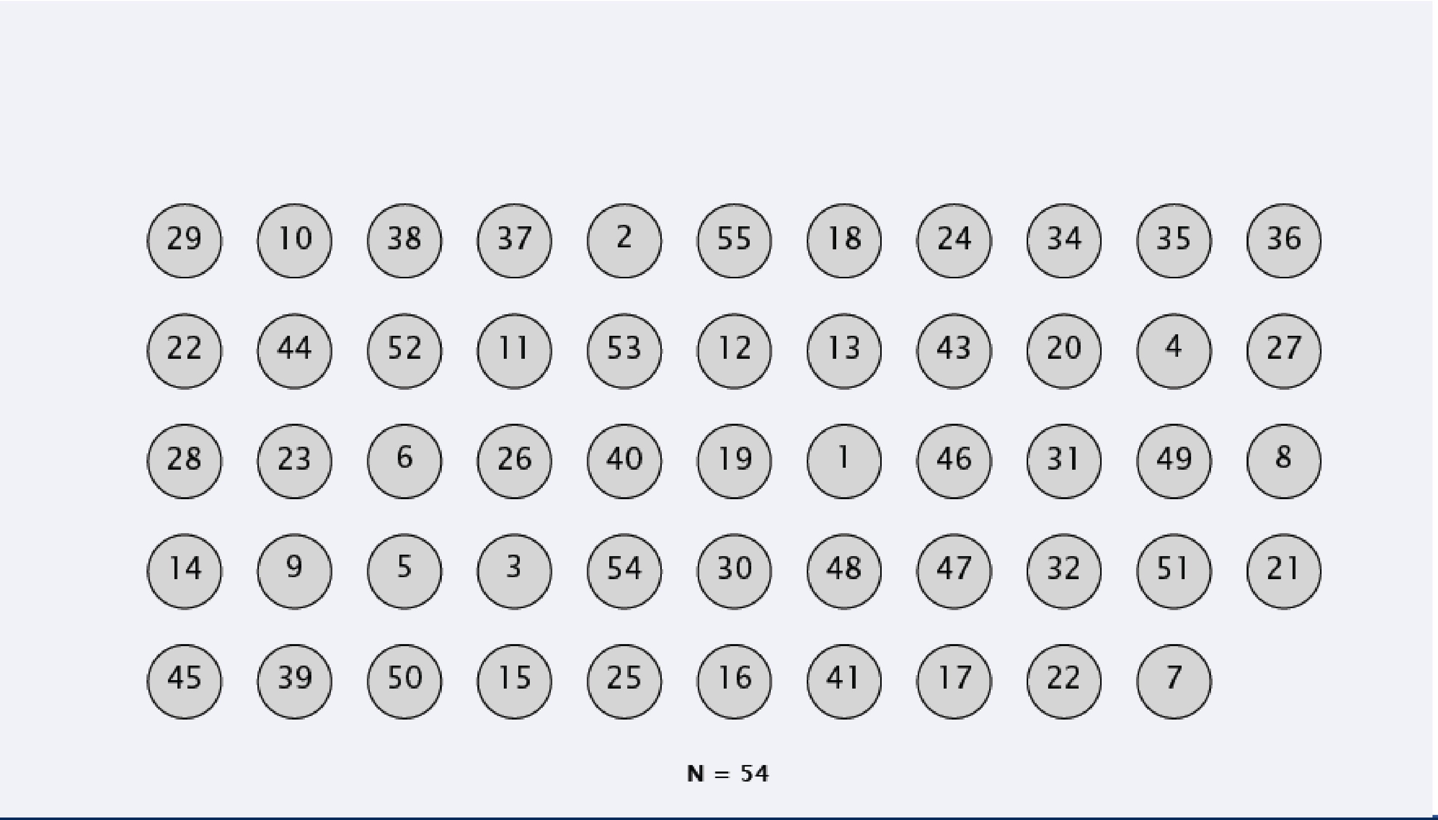
QuickSelect

- Find a pivot p
- Divide A into two sub-arrays
  - A_{less} = elements $\leq p$, A_{more} = elements $> p$
  - If $|A_{less}| \geq k$, return k-th smallest in A_{less}
  - Otherwise, return the $(k - |A_{less}|)$-th smallest element in A_{more}

- Problem?
  - The algorithm is *correct regardless of the choice of the pivot*
  - But the pivot choice crucially affects the running time
    - If pivot is close to min or max, then we get $T(n) \leq T(n-1) + O(n) \Rightarrow T(n) = O(n^2)$
    - We want to reduce $n - 1$ to a fraction of n (e.g., $n/2$, $5n/6$, etc)

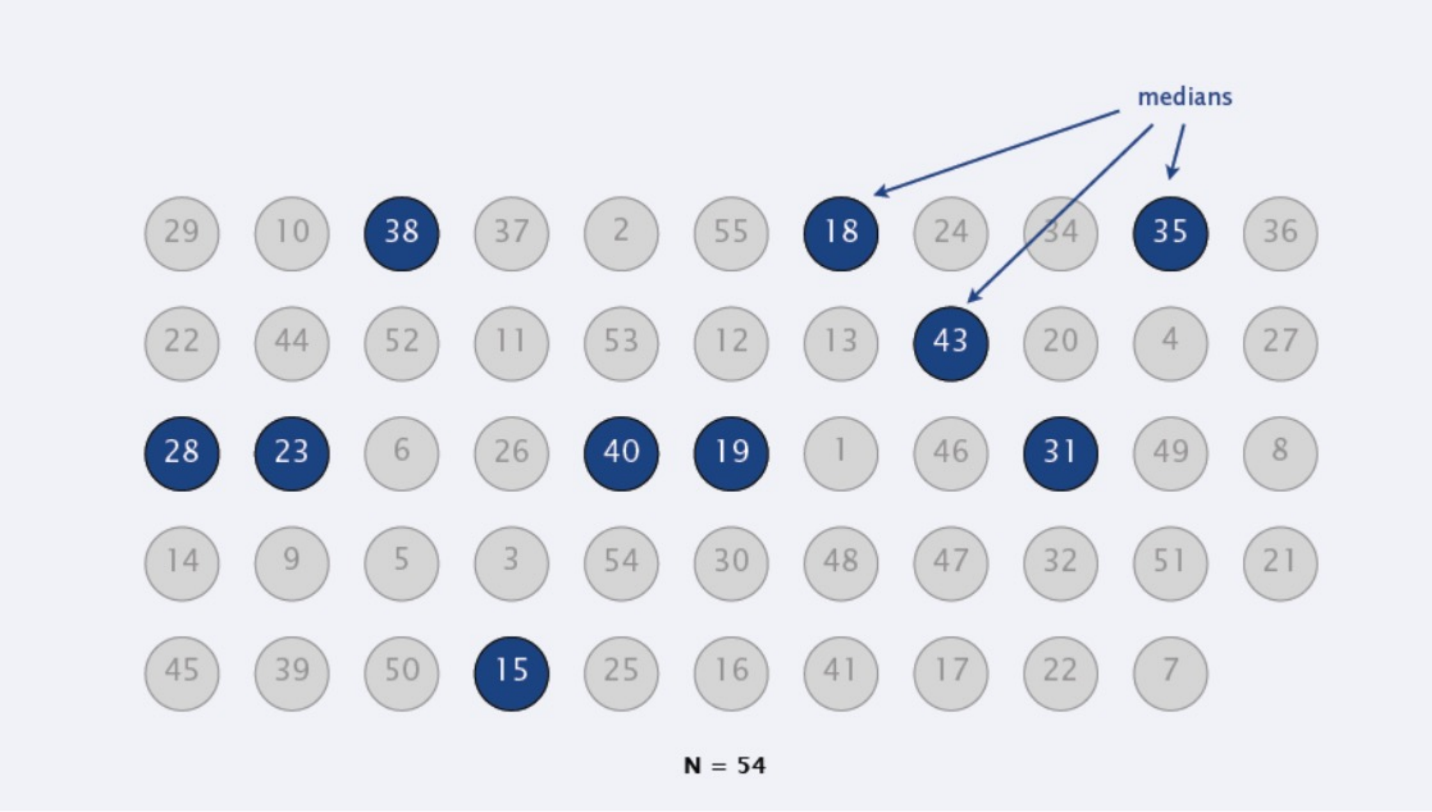
Finding a Good Pivot

- Divide n elements into $^n/_5$ groups of 5 each



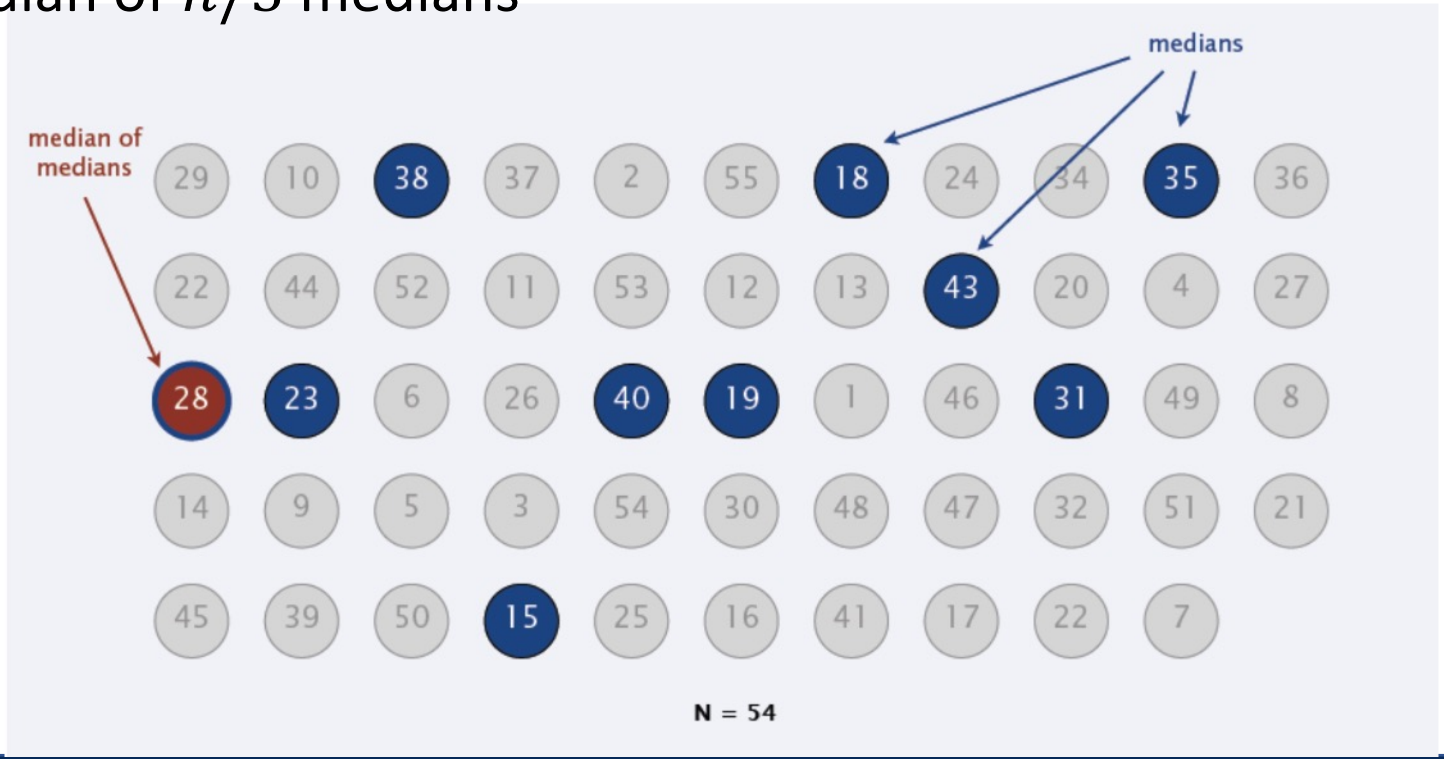
Finding a Good Pivot

- Divide n elements into ${}^{n}/_{5}$ groups of 5 each
- Find the median of each group



Finding a Good Pivot

- Divide n elements into ${}^{n}/_{5}$ groups of 5 each
- Find the median of each group
- Find the median of $n/5$ medians



Finding a Good Pivot

- Divide n elements into ${}^{n}/_{5}$ groups of 5 each
- Find the median of each group
- Find the median of $n/5$ medians
- Use this median of medians (call it p^*) as the pivot in quickselect

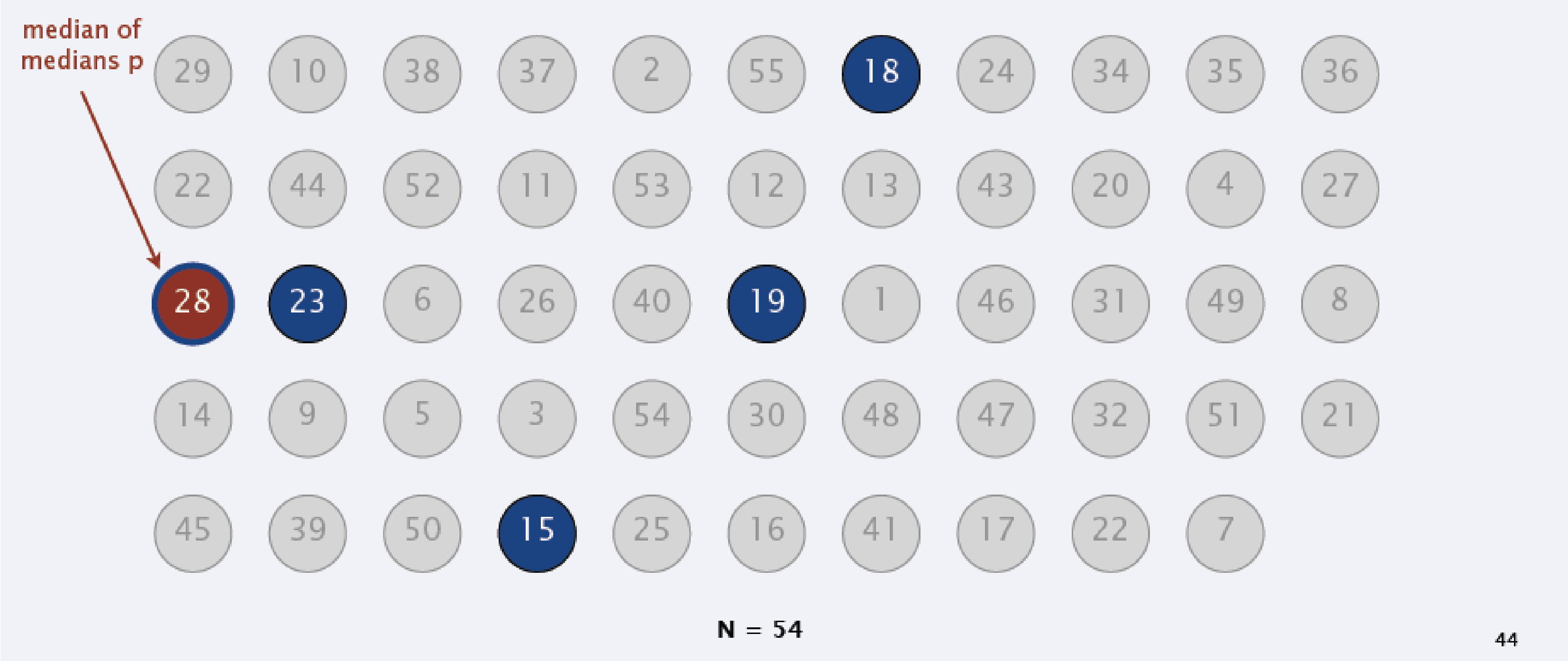
- Q: Why does this work?

Analysis

- Let's find an upper bound on $|A_{more}|$, which contains elements $> p^*$
- How many elements can be $> p^*$?
- $n/10$ out of the $n/5$ medians are $> p^*$
  - Even if all 5 elements in their groups are more than p^*, that's only $5n/10$ in total
- What about the other groups whose medians are $\leq p^*$?

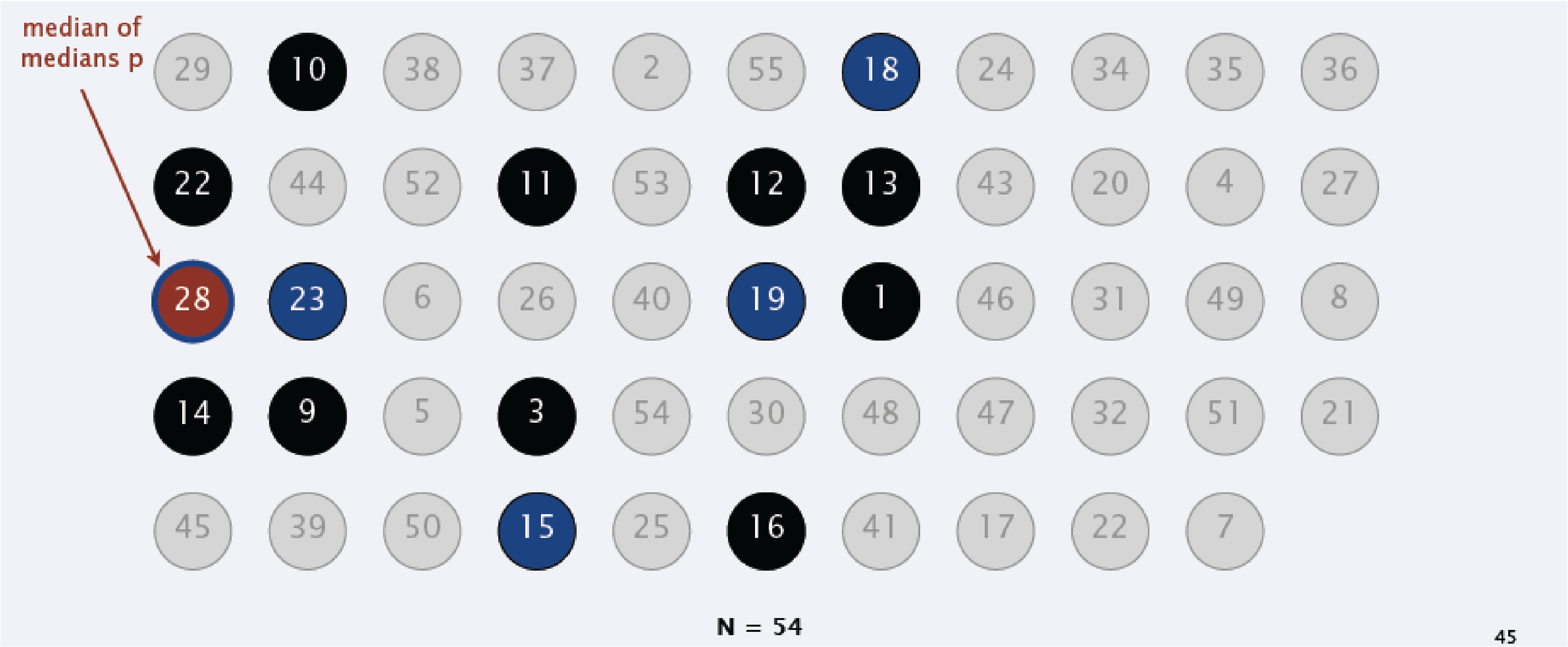
Analysis

- $n/_{10}$ of the $n/_5$ medians are $\leq p^*$



Analysis

- ${}^{n}/_{10}$ of the ${}^{n}/_{5}$ medians are $\leq p^*$
  - For each such group, there are at least 3 elements $\leq p^*$, so only 2 elements can be $> p^*$
  - So, from such groups, there can be at most ${}^{2n}/_{10}$ elements $> p^*$



Analysis

- Thus, $|A_{more}| \leq {}^{5n}/_{10} + {}^{2n}/_{10} = {}^{7n}/_{10}$
  - Similarly, $|A_{less}| \leq {}^{7n}/_{10}$
  - These are rough calculations, but can be made exact by using ceiling/floor and accounting for the last group with possibly less than 5 elements, but that makes no difference asymptotically

- How does this factor into overall algorithm analysis?

Analysis

- Divide n elements into ${}^{n}/_{5}$ groups of 5 each
- Find the median of each group
- Find p^* = median of ${}^{n}/_{5}$ medians
- Create A_{less} and A_{more} according to p^*
- Run selection on *one of* A_{less} or A_{more}

$O(n)$

$T(n/5)$

$O(n)$

$T(7n/10)$

- $T(n) \leq T({}^{n}/_{5}) + T({}^{7n}/_{10}) + O(n)$
- Note: ${}^{n}/_{5} + {}^{7n}/_{10} = {}^{9n}/_{10}$
  - Only a fraction of n, so using a similar analysis to the one in the Master theorem, $T(n) = O(n)$

Residual Notes

- Lower bounds on the worst-case running time
  - Note that we only derived *upper bounds* on the worst-case running time of the form $T(n) = O(n^2)$ or $T(n) = O(n)$
  - If we want to claim that our algorithm does not run *faster* than what is claimed in this upper bound, we have to produce a matching *lower bound*, e.g., $T(n) = \Omega(n^2)$
    - This is typically done by producing a *family of examples*, one for each value of n, such that the algorithm's running time on these examples grows like n^2 as the value of n grows
  - If we want to claim that *no algorithm* can solve the problem faster than, say, $O(n^2)$, that's usually much, much harder (but has been done for several problems)!

Residual Notes

- Best algorithm for a problem?
  - We still don't know best algorithms for multiplying two n-digit integers or two $n \times n$ matrices
    - Integer multiplication
      - 1960 (Karatsuba): $O\left(n^{\log_2 3}\right) \approx O(n^{1.585})$
      - 1971: $O(n \log n \log \log n)$
      - 2007: $O\left(n \log n \ 2^{C \log^* n}\right)$ for some constant C
      - 2014: $O\left(n \log n \ 2^{3 \log^* n}\right)$
      - 2019: $O(n \log n)$ --- breakthrough, conjectured to be asymptotically optimal
    - Matrix multiplication
      - 1969 (Strassen): $O(n^{2.807})$
      - 1990: $O(n^{2.376})$
      - 2013: $O(n^{2.3729})$
      - 2014: $O(n^{2.3728639})$

Residual Notes

- Best algorithm for a problem?
  - Usually, we design an algorithm and then analyze its running time
  - Sometimes we can do the reverse:
    - E.g., if you know you want an $O(n^2 \log n)$ algorithm
    - Master theorem suggests that you can get it by
      $$T(n) = 4\,T\left({}^{n}/_{2}\right) + O(n^2)$$
    - So maybe you want to break your problem into 4 problems of size $n/2$ each, and then do $O(n^2)$ computation to combine

Residual Notes

- Access to input
  - For much of this analysis, we are assuming random access to elements of input
  - So, we're ignoring underlying data structures (e.g., doubly linked list, binary tree, etc.)

- Machine operations
  - We're only counting the number of comparisons or arithmetic operations
  - So, we're ignoring issues like how real numbers are stored in the closest pair problem
  - When we get to P vs NP, representation will matter

Residual Notes

- Size of the problem
  - Can be any reasonable parameter of the problem
  - E.g., for matrix multiplication, we used n as the size
  - But an input consists of two matrices with n^2 entries
  - It doesn't matter whether we call n or n^2 the size of the problem
  - The actual running time of the algorithm won't change