Divide and Conquer: More Examples

Admin

- Welcome back!

- Assignment 4 extended to Saturday

- Remember that if you’re watching at home you can switch between the board/me and the slides with a button in the top-right corner of the zoom

Quick Sort Analysis

- Partition takes $O(n)$ time
- Size of the subproblems depends pivot; let r be the rank of the pivot, then:
- $T(n) = T(r-1) + T(n-r) + O(n),\ \ T(1) = 1$
- Let us analyze some cases for r
    - **Best case:** r is the median: $r = \lfloor n/2 \rfloor$ (we will learn how to compute the median in $O(n)$ time)
    - **Worst case:** $r = 1$ or $r = n$
    - **In between:** say $n/10 \leq r \leq 9n/10$
- Note in the worst-case analysis, we only consider the worst case for r. We are looking at the difference cases, just to get a sense for it.

Quick Sort: Cases

- Suppose $r = n/2$ (pivot is the median element), then
  - $T(n) = 2T(n/2) + O(n)$, $T(1) = 1$
  - We have already solved this recurrence
  - $T(n) = O(n \log n)$
- Suppose $r = 1$ or $r = n - 1$, then
  - $T(n) = T(n-1) + T(1) + 1$
  - What running time would this recurrence lead to?
  - $T(n) = \Theta(n^2)$ (notice: this is tight!)

Quick Sort: Cases

- Suppose $r = n/10$ (that is, you get a one-tenth, nine-tenths split
- $T(n) = T(n/10) + T(9n/10) + O(n)$
- Let's look at the recursion tree for this recurrence

- We get $T(n) = O(n \log n)$, in fact, we get $\Theta(n \log n)$

Challenge Recurrence

- Solve the following recurrence:

$$T(n) = \sqrt{n}T(\sqrt{n}) + n$$

- **Hint.** Try some change of variables

Counting Inversions

- Way to compare two different rankings
- Or a way to measure how far an array is from sorted
- Let $a_1, a_2, \ldots, a_n$ be an ordering of n numbers
- We say two indices $i < j$ form an **inversion** if $a_i > a_j$

- Example: How many inversions in $2,4,1,3,5$?
    - $2,1$ is an inversion
    - $4,1$ and $4,3$ is an inversion
    - 3 inversions total

Counting Inversions

- Way to compare two different rankings
- Or a way to measure how far an array is from sorted
- Let $a_1, a_2, \ldots, a_n$ be an ordering of n numbers
- We say two indices $i < j$ form an **inversion** if $a_i > a_j$
- Counting all inversions in a naive way:
    - Comparing every pair is $\Theta(n^2)$
- **Can we do better by divide and conquer?**

Divide and Conquer

Tools we need:

- Split the instance into multiple parts
- Way to combine solution for each part into a solution for the entire instance

Counting Inversions: Divide & Conquer

- **Divide:** break array into two halves A and B
- **Conquer:** recursively count number of inversions in both
- **Combine:** count number of inversions of the type (a, b) where $a \in A, b \in B$ and return total

- How do combine in $O(n)$ time?
- Problem: there are $n/2$ elements in A and $n/2$ elements in B, so there may be $n^2/4$ inversions we didn't count recursively
- **Idea:** easy if A and B are sorted!

Sort and Recurse

- We will simultaneously sort the array while counting inversions

- Key observation: sorting A and B does not change the number of inversions crossing the midpoint

Counting Inversions: Divide & Conquer

- Counting inversions: (a, b) where $a \in A, b \in B$ when A, B are sorted
- Scan both from left to right
- Compare a_i and b_j

count inversions (a, b) with a ∈ A and b ∈ B

3	7	10	a_i	18

2	11	b_j	20	23
5	2			

merge to form sorted list C

2	3	7	10	11					

Counting Inversions: Divide & Conquer

- Counting inversions: (a, b) where $a \in A, b \in B$ when A, B are sorted
- Scan both from left to right
- Compare a_i and b_j
- If $a_i < b_j$,
    - a_i is not inverted wrt all remaining elements in B
- If $a_i > b_j$
    - b_j is inverted with respect to every element left in A
- Append smaller element to sorted list C

Counting Inversions: Divide & Conquer

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). ⟵ $T(n / 2)$

$(r_B,\ B) \leftarrow$ SORT-AND-COUNT(B). ⟵ $T(n / 2)$

$(r_{AB},\ L) \leftarrow$ MERGE-AND-COUNT(A, B). ⟵ $\Theta(n)$

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

Combine Step

Counting Inversions: Analysis

- Same as merge sort
- $O(n)$ time to merge and count (non-recursive)
- Two subproblems of half the size

- $T(n) = 2T(n/2) + cn$
- $T(n) = O(n \log n)$

Other Kinds of Recurrences

So far we saw divide and conquer algorithms, where we split the problem in more than one subproblem.

Question. Can you think of some examples (that you have likely seen before) where we split the problem into **one** smaller subproblem?

D&C: One Smaller Subproblem

- Binary search
  - $T(n) = T(n/2) + 1$
- Binary search tree
  - $T(n) = T(n/2) + 1$

D&C: One Smaller Subproblem

- Fast exponentiation (you may not have seen this)
    - Compute a^n, how many multiplications?
    - Naive way: $a \cdot a \cdot \ldots \cdot a$ (n times)
    - Faster way: $a^n = (a^{n/2})^2$ (suppose n is even)
    - $T(n) = T(n/2) + 1$
    - What does this solve to?
    - Think at home: What if n is odd?

General Recursion Trees

- Consider a divide and conquer algorithm that
    - spends $O(f(n))$ time on non-recursive work and makes r recursive calls, each on a problem of size n/c
- Up to constant factors (which we hide in $O()$), the running time of the algorithm is given by what **recurrence**?
    - $T(n) = rT(n/c) + f(n)$
- Because we care about asymptotic bounds, we can assume base case is a small constant, say $T(n) = 1$

General Recursion Trees

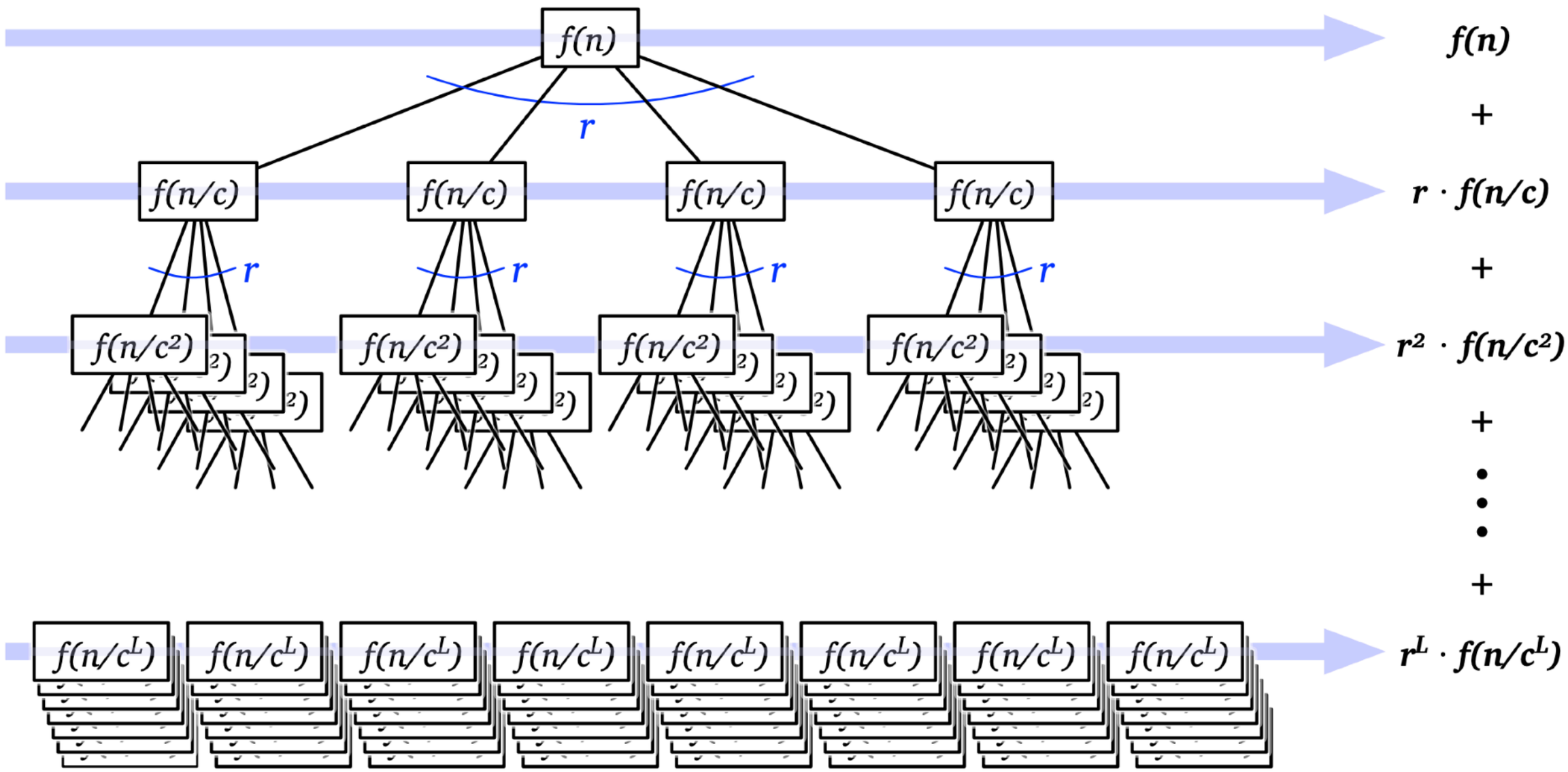


Figure 1.9. A recursion tree for the recurrence $T(n) = r\,T(n/c) + f(n)$

General Recursion Trees

- Running time $T(n)$ of a recursive algorithm is the sum of all the values (sum of work at all nodes at each level) in the recursion tree
- For each i, the ith level of tree has exactly r^i nodes
- Each node at level i, has cost $f(n/c^i)$
- Thus, $T(n) = \sum_{i=0}^{L} r^i \cdot f(n/c^i)$
- Here $L = \log_c n$ is the depth of the tree
- The number of leaves in the tree is $r^L = n^{\log_c r}$ (why?)
- Cost at leaves: $O(n^{\log_c r} f(1))$

Common Cases

Don't forget: $\sum_{i=0}^{L} a^i = \frac{a^{L+1} - 1}{a - 1}$

$$T(n) = \sum_{i=0}^{L} r^i \cdot f(n/c^i)$$

- **Decreasing series.** If the series decays exponentially (every term is a constant factor smaller than previous), cost at root dominates:
$$T(n) = O(f(n))$$
- **Equal.** If all terms in the series are equal:
$$T(n) = O(f(n) \cdot L) = O(f(n) \log n)$$
- **Increasing series.** If the series grows exponentially (every term is constant factor larger), then the cost at leaves dominates:
$$T(n) = O(n^{\log_c r})$$

Master Theorem (optional)

Set of rules to solve some common recurrences automatically

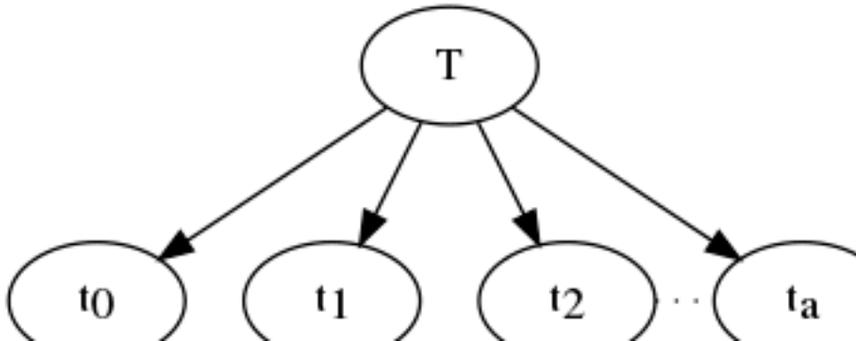
(Master Theorem) Let $a \geq 1$, $b > 1$ and $f(n) \geq 0$. Let $T(n)$ be defined by the recurrence $T(n) = aT(n/b) + f(n)$ and $T(1) = O(1)$.

Then $T(n)$ can be bounded asymptotically as follows.

- If $f(n) = n^{\log_b a - \epsilon}$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$
- If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$
- If $f(n) = \Omega(n^{\log_b a + \epsilon})$, for some constant $\epsilon > 0$, and if $af(n/b) \leq c_0 f(n)$ for some constant $c_0 < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$

Master Theorem

- It exists; it can make things easier. You don't need to know it
- OK to use in this class, but I don't encourage (nor discourage) it
  - Recursion trees promote a better understanding of the recurrence—and they can be simpler
- Master Theorem only applies to some recurrences (generalizations do exist)



Selection: Problem Statement

Given an array $A[1,\ldots,n]$ of size n, find the kth smallest element for any $1 \leq k \leq n$ (a.k.a. the element of **rank** k)

- Special cases: min $k = 1$, max $k = n$:
    - Linear time, $O(n)$
- What about **median** $k = \lfloor n + 1 \rfloor / 2$?
    - Sorting: $O(n \log n)$ compares

Question. Can we do it in $O(n)$ compares?

- **Surprisingly yes.**
- We'll find the element of rank k in $O(n)$ time for any k
- Selection is easier than sorting.

Selection: Problem Statement

Example. Take this array of size 10:

$$A = 12|2|4|5|3|1|10|7|9|8$$

Suppose we want to find 4th smallest element

- First, take any pivot p from $A[1,\ldots n]$
- If p is the 4th smallest element, return it
- Else, we partition A around p and recurse

Selection Algorithm: Idea

Select (A, k):

If $|A| = 1$: return $A[1]$

Else:

- Choose a pivot $p \leftarrow A[1, \ldots, n]$; let r be the rank of p
- $r, A_{<p}, A_{>p} \leftarrow$ Partition$((A, p)$
- If $k == r$, return p
- Else:
    - If $k < r$: Select $(A_{<p}, k)$
    - Else: Select $(A_{>p}, k - r)$

Selection: Problem Statement

Example. Take this array of size 10:

$$A = 12|2|4|5|3|1|10|7|9|8$$

Suppose we want to find 4th smallest element

- Choose pivot 8
- What is its rank?
  - Rank 7
- So let's find all of the smaller elements of A:
  - $A' = 2|4|5|3|1|7$
- Want to find the element of rank 4 in this new array

Selection: Problem Statement

Example. Take this array of size 10:

$$A = 12|2|4|5|3|1|10|7|9|8$$

Suppose we want to find 4th smallest element

- Choose as pivot 3
- What is its rank?
  - Rank 3
- So let's find all of the **larger** elements of A:
  - $A' = 12|4|5|3|10|7|9|8$
- Want to find the element of rank $4 - 3 = 1$ in this new array

When is this method good?

- If we guess the pivot right! (but we can't always do that)
- If we partition the array pretty evenly (the pivot is close to the middle)
  - Let's say our pivot is not in the first or last $3/10$ths of the array
  - What is our recurrence?
  - $T(n) \leq T(7n/10) + O(n)$
  - $T(n) = O(n)$

Our high-level goal

- Find a pivot that's in the "middle" of the array
- But the array is unsorted? How do we do that?
- Want to *always* be successful

Finding an Approximate Median

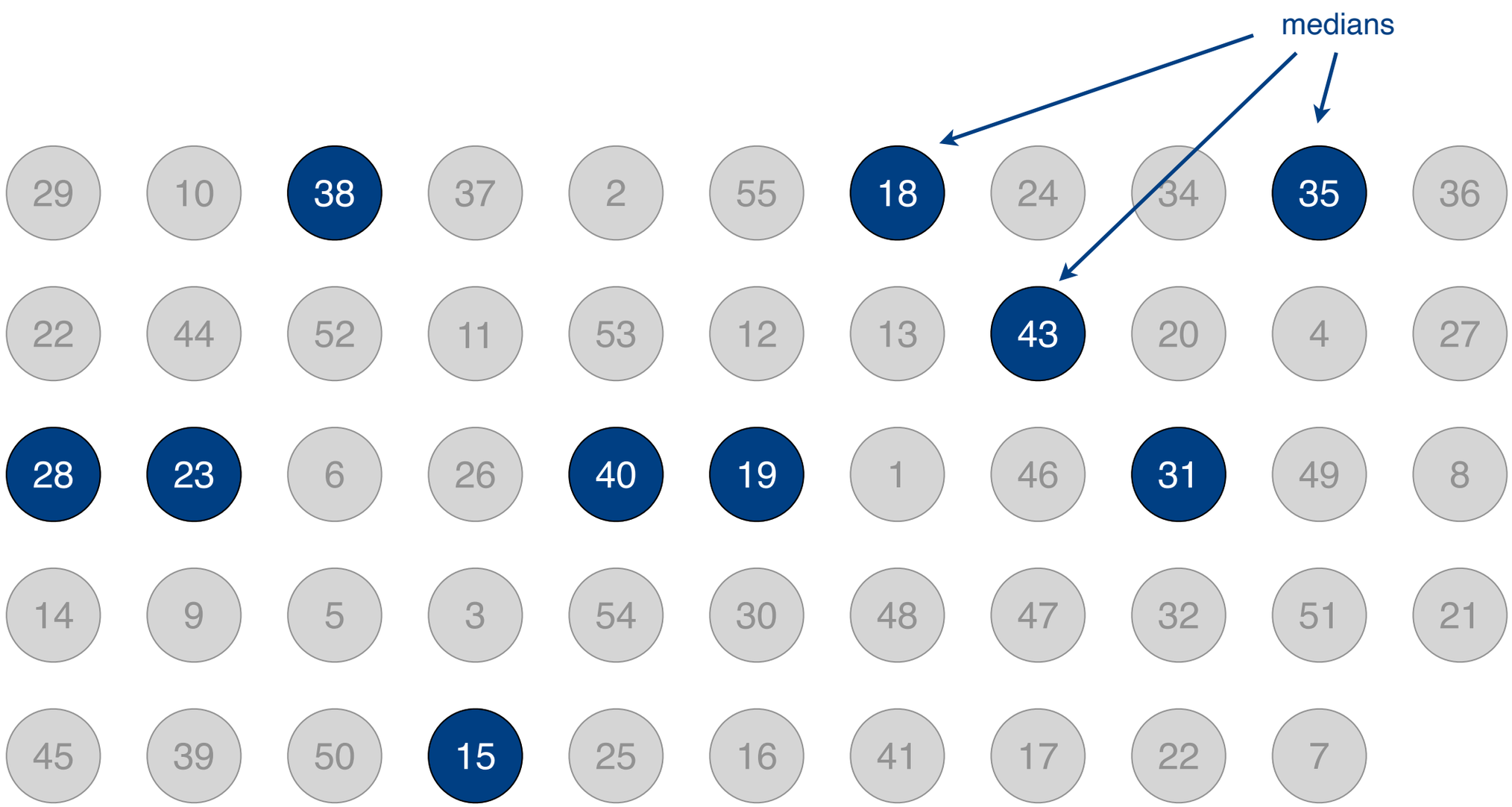
- Divide the array of size n into $\lceil n/5 \rceil$ groups of 5 elements (ignore leftovers)
- Find median of each group

29	10	38	37	2	55	18	24	34	35	36
22	44	52	11	53	12	13	43	20	4	27
28	23	6	26	40	19	1	46	31	49	8
14	9	5	3	54	30	48	47	32	51	21
45	39	50	15	25	16	41	17	22	7	

n = 54

Finding an Approximate Median

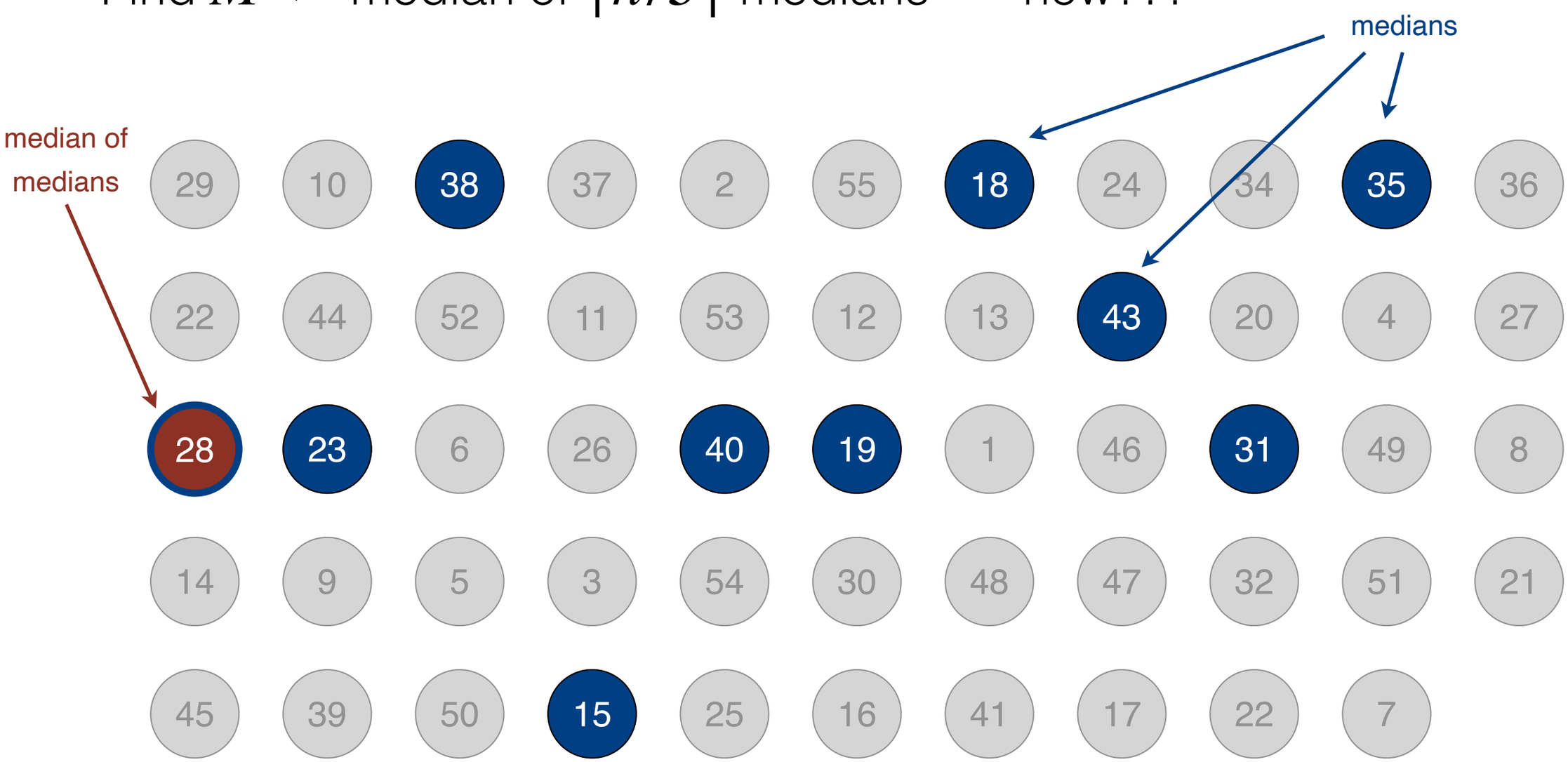
- Divide the array of size n into $\lceil n/5 \rceil$ groups of 5 elements (ignore leftovers)
- Find median of each group



n = 54

Finding an Approximate Median

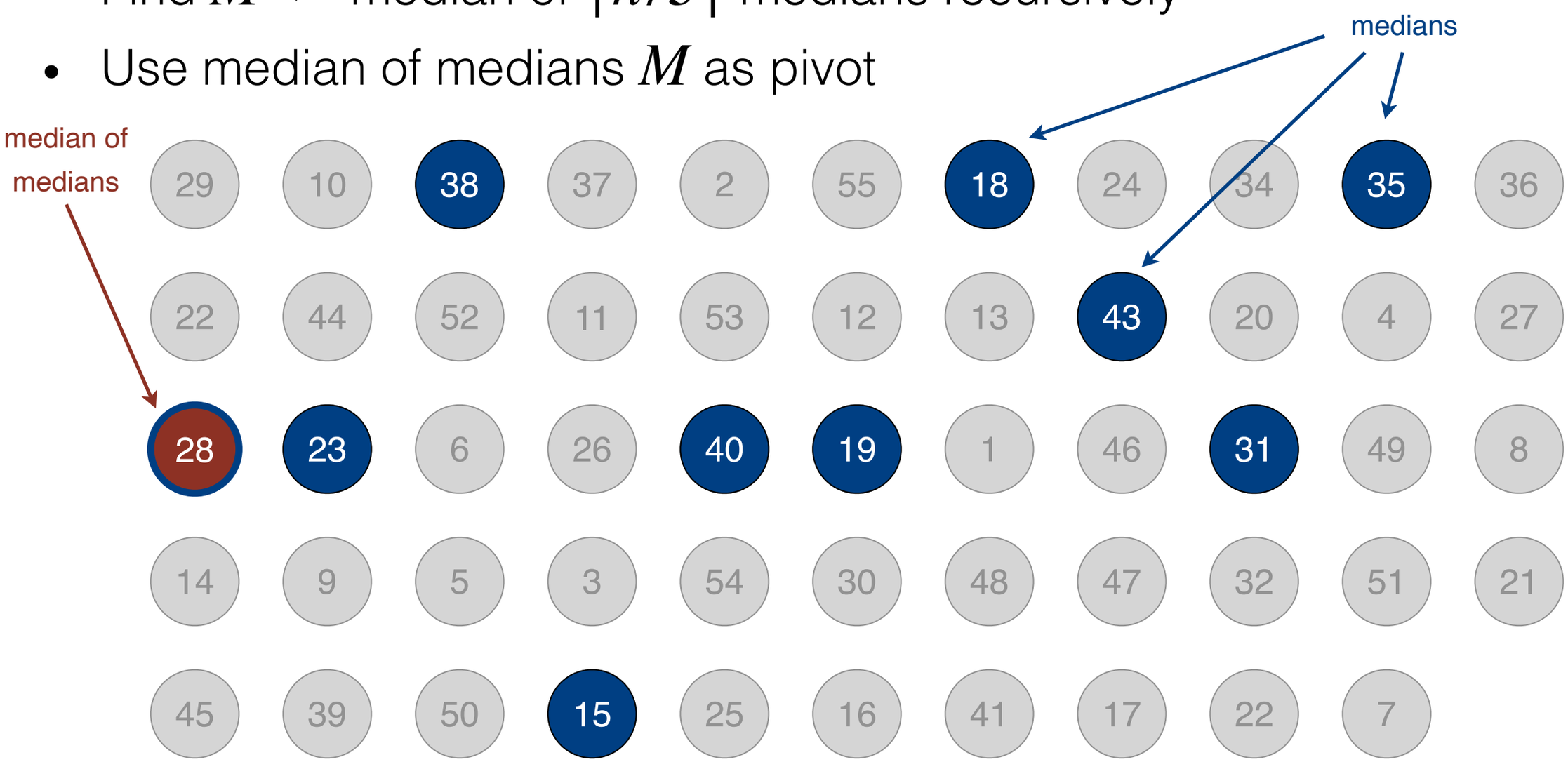
- Divide the array of size n into $\lceil n/5 \rceil$ groups of 5 elements (ignore leftovers)
- Find median of each group
- Find $M \leftarrow$ median of $\lceil n/5 \rceil$ medians —- how???



n = 54

Finding an Approximate Median

- Divide the array of size n into $\lceil n/5 \rceil$ groups of 5 elements (ignore leftovers)
- Find median of each group
- Find $M \leftarrow$ median of $\lceil n/5 \rceil$ medians recursively
- Use median of medians M as pivot



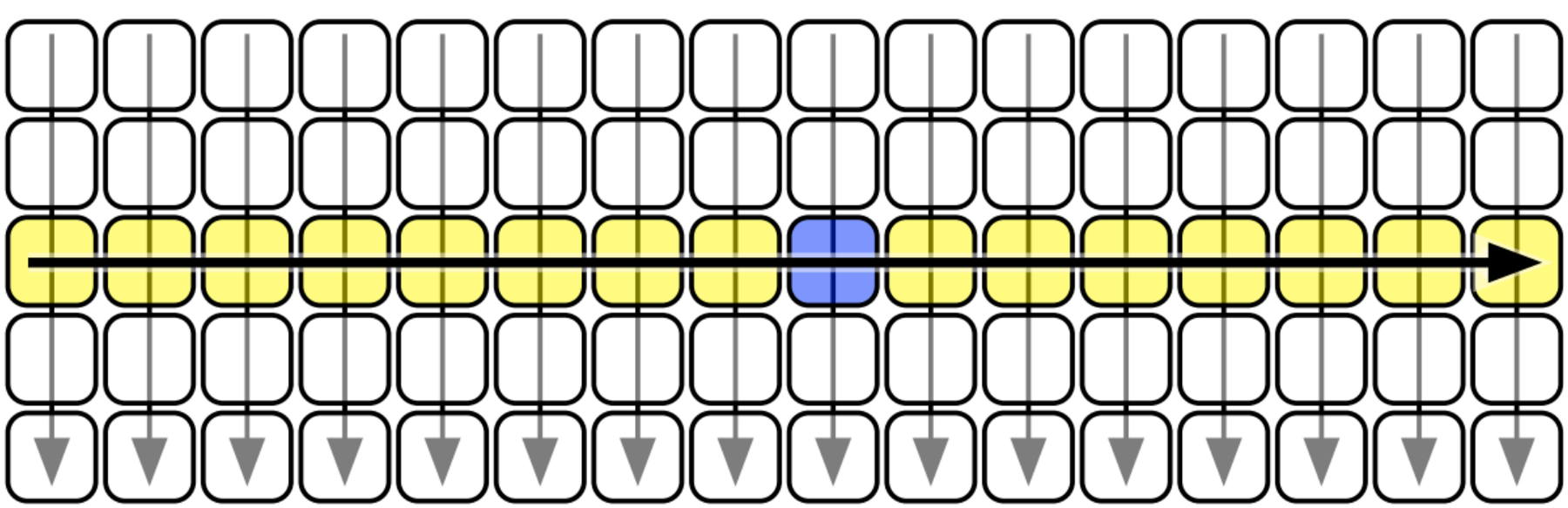
n = 54

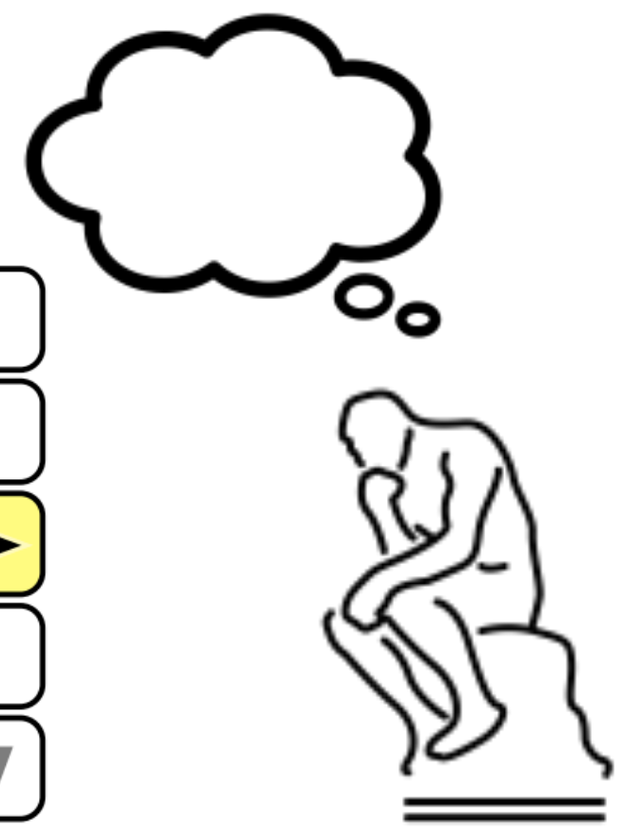
What did we gain?

- How can I show that the median of medians is “close to the center” of the array?
- What elements can I say, for sure, are $\leq$ the median of medians?
  - The smaller half of the medians
  - $n/10$ elements
- Any other elements?
  - Another 2 elements in each median’s list

Visualizing MoM

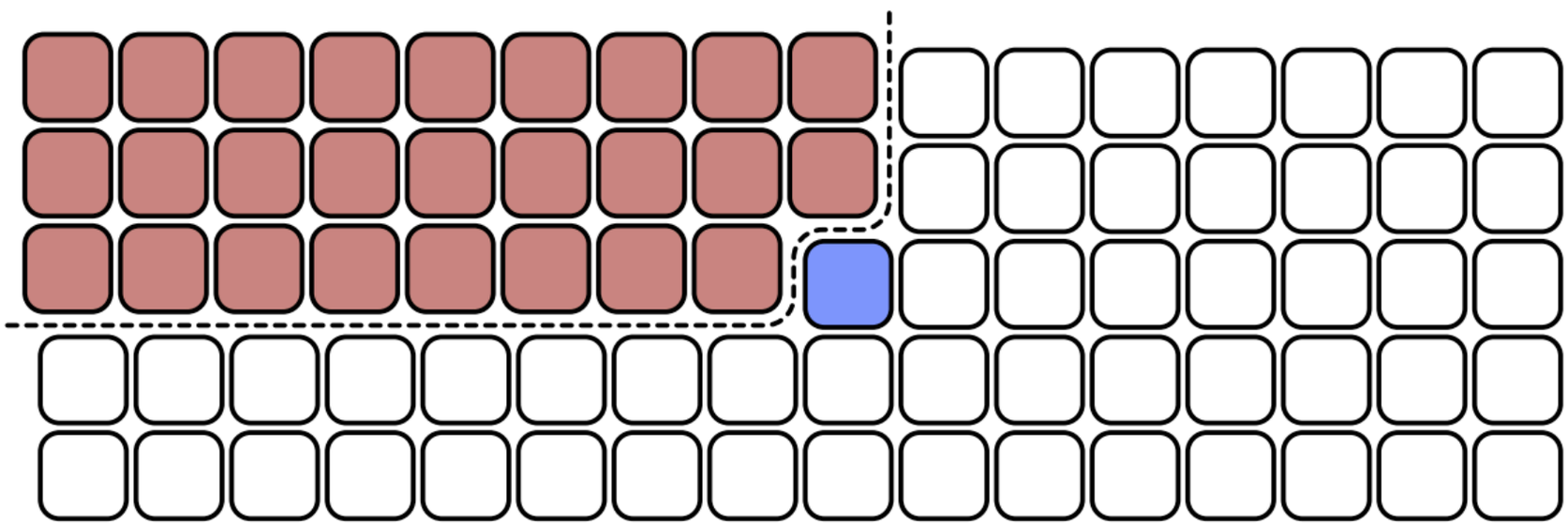
- In the $5 \times n/5$ grid, each column represents five consecutive elements
- Imagine each column is sorted top down
- Imagine the columns as a whole are sorted left-right
    - We don't actually do this!
- MoM is the element closest to center of grid





Visualizing MoM

- Red cells (at least $3n/10$) in size are smaller than M



How Good is the MoM?

Claim. Median of medians M is a good pivot, that is, at least $3/10$th of the elements are $\geq M$ and at least $3/10$th of the elements are $\leq M$.

Proof.

- Let $g = \lceil n/5 \rceil$ be the size of each group.
- M is the median of g medians
  - So $M \geq g/2$ of the group medians
  - Each median is greater than 2 elements in its group
  - Thus $M \geq 3g/2 = 3n/10$ elements
- Symmetrically, $M \leq 3n/10$ elements. ∎

How to Use the MoM?

- There are $3n/10$ elements smaller than the MoM
- By the same argument: $3n/10$ elements larger than the MoM
- So we can throw out $3n/10$ elements, adjust the value of k we are looking for, and recurse!
- Don't forget: we *also* recursed to find the MoM!

Recall: Selection

Select (A, k):

If $|A| = 1$: return $A[1]$

Else:

- Choose a pivot $p \leftarrow A[1, \ldots, n]$; let r be the rank of p
- $r, A_{<p}, A_{>p} \leftarrow$ Partition$((A, p)$
- If $k == r$, return p
- Else:
    - If $k < r$: Select $(A_{<p}, k)$
    - Else: Select $(A_{>p}, k - r)$

Linear time Selection

Select (A, k):

If $|A| = 1$: return $A[1]$; else:

- Group elements into subarrays of size 5; find median in each
- Choose a pivot p as the median of these medians
- $r, A_{<p}, A_{>p} \leftarrow$ Partition$((A, p)$
- If $k == r$, return p
- Else:
  - If $k < r$: Select $(A_{<p}, k)$
  - Else: Select $(A_{>p}, k - r)$

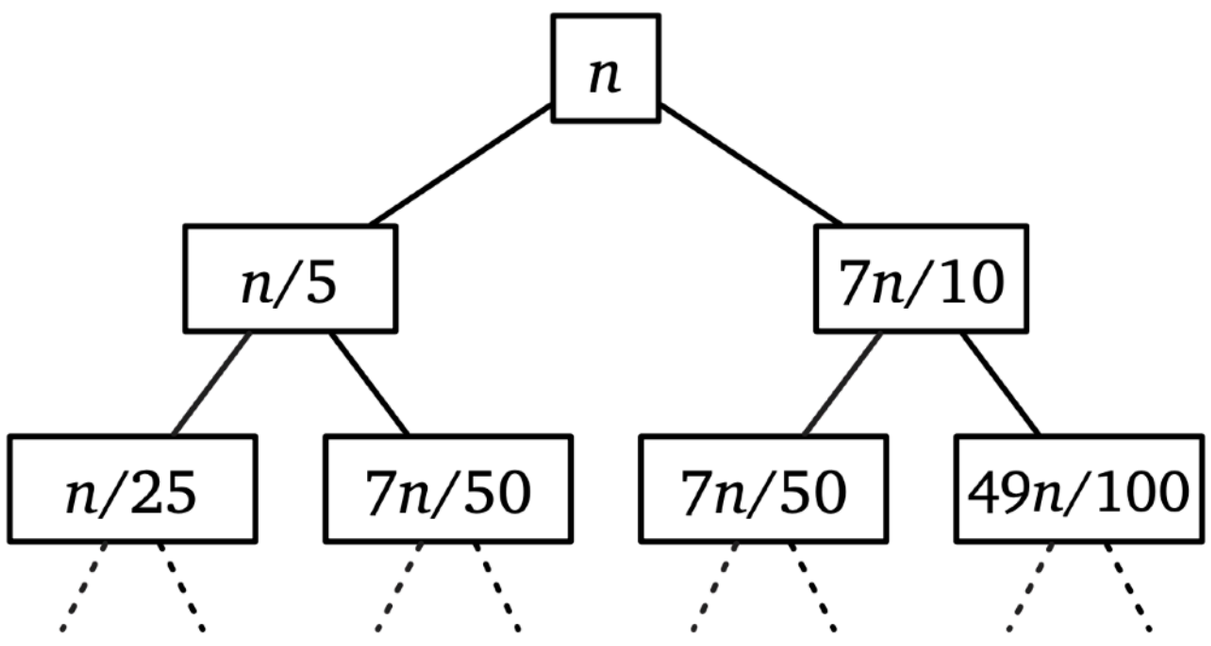
$T(n/5) + O(n)$

Larger subproblem has size $\leq 7n/10$

Overall: $T(n) = T(n/5) + T(7n/10) + O(n)$

Selection Recurrence

- Okay, so we have a good pivot
- We are still doing two recursive calls
    - $T(n) \leq T(n/5) + T(7n/10) + O(n)$
- Key: total work at each level still goes down!
- Decaying series gives us : $T(n) = O(n)$



Why the Magic Number 5?

- What was so special about 5 in our algorithm?
- It is the smallest odd number that works!
    - (Even numbers are problematic for medians)
- Let us analyze the recurrence with groups of size 3
    - $T(n) \leq T(n/3) + T(2n/3) + O(n)$
    - Work is equal at each level of the tree!
    - $T(n) = \Theta(n \log n)$

Theory vs Practice

- $O(n)$-time selection by [Blum–Floyd–Pratt–Rivest–Tarjan 1973]
    - Does $\leq 5.4305n$ compares
- Upper bound:
    - [Dor–Zwick 1995] $\leq 2.95n$ compares
- Lower bound:
    - [Dor–Zwick 1999] $\geq (2 + 2^{-80})n$ compares.
- Constants are still too large for practice
- Random pivot works well in most cases!
    - We will analyze this when we do randomized algorithms

Recall Challenge Recurrence

- Recall the challenge recurrence

$$T(n) = \sqrt{n}T(\sqrt{n}) + O(n)$$

- Analyzing how quickly the problem size goes down
- $n \to n^{1/2} \to n^{1/4} \to \ldots \to n^{1/2^L}$
- What is L for this to be a small constant?
- $L = \log\log n$ (number of levels)
- How much work at each level? $O(n)$
- $T(n) = \Theta(n \log\log n)$,

Floors and Ceilings

- Why doesn't floors and ceilings matter?
- Suppose $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n)$
- First, for upper bound, we can safely overestimate
  - $T(n) \leq 2T(\lceil n/2 \rceil) + n \leq 2T(n/2 + 1) + n$
- Second, we can define a function $S(n) = T(n + \alpha)$, so that $S(n)$ satisfies $S(n) \leq S(n/2) + O(n)$

$$
\begin{aligned}
S(n) = T(n + \alpha) &\leq 2T(n/2 + \alpha/2 + 1) + n + \alpha \\
&= 2T(n/2 + \alpha - \alpha/2 + 1) + n + \alpha \\
&= 2S(n/2 - \alpha/2 + 1) + n + \alpha \\
&\leq 2S(n/2) + n + 2, \text{ for } \alpha = 2
\end{aligned}
$$

Floors & Ceilings Don't Matter

- Why doesn't floors and ceilings matter?
- Suppose $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n)$
- First, for upper bound, we can safely overestimate
  - $T(n) \leq 2T(\lceil n/2 \rceil) + n \leq 2T(n/2 + 1) + n$
- Second, we can define a function $S(n) = T(n + \alpha)$, so that $S(n)$ satisfies $S(n) \leq S(n/2) + O(n)$
  - Setting $\alpha = 2$ works
- Finally, we know $S(n) = O(n \log n) = T(n + 2)$
- $T(n) = O((n - 2)\log(n - 2)) = O(n \log n)$

Can Assume Powers of 2

- Why doesn't taking powers of 2 matter?
- Running time $T(n)$ is monotonically increasing
- Suppose n is not a power of 2, let $n' = 2^{\ell}$ be such that $n \leq n' \leq 2n$; then
- We can upper bound our asymptotic using n' and lower bound using $n'/2$
- In particular, let $T(n) \leq T(n')$
- And $T(n) \geq T(n'/2)$
- That is, $T(n) = \Theta(T(n'))$

Acknowledgments

- Some of the material in these slides are taken from
  - Kleinberg Tardos Slides by Kevin Wayne (https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/04GreedyAlgorithmsI.pdf)
  - Jeff Erickson's Algorithms Book (http://jeffe.cs.illinois.edu/teaching/algorithms/book/Algorithms-JeffE.pdf)