Divide and Conquer: Sorting and Recurrences

Divide & Conquer: Quicksort

- Choose a pivot element from the array
- Partition the array into two parts:
  - LEFT: all elements that are less than or equal to the pivot
  - RIGHT: all elements that are greater than the pivot
- Recursively quicksort the LEFT and RIGHT subarrays

Input:	S	O	R	T	I	N	G	E	X	A	M	P	L
Choose a pivot:	S	O	R	T	I	N	G	E	X	A	M	**P**	L
Partition:	A	G	O	E	I	N	L	M	**P**	T	X	S	R
Recurse Left:	A	E	G	I	L	M	N	O	**P**	T	X	S	R
Recurse Right:	A	E	G	I	L	M	N	O	**P**	R	S	T	X

Divide & Conquer: Quicksort

- **Description.** (Divide and conquer): often the cleanest way to present is **short and clean pseudocode** with high level explanation
- **Correctness proof.** Induction and showing that partition step correctly partitions the array.

```
QuickSort(A[1 .. n]):
    if (n > 1)
        Choose a pivot element A[p]
        r ← Partition(A, p)
        QuickSort(A[1 .. r − 1])    ⟨⟨Recurse!⟩⟩
        QuickSort(A[r + 1 .. n])    ⟨⟨Recurse!⟩⟩
```

Quick Sort Analysis

- How long does partition take? $O(n)$
- Let's write a recurrence relation for quick sort!
- Challenge: the size of the subproblems depends pivot.
  - Idea: let r be the rank of the pivot, where rank is the (lowest) index of the item in the sorted list.
  - Base case:

$$T(1) = 1$$

  - General Case:

Partition that's $\leq$ pivot

Partition that's $>$ pivot

$$T(n) = T(r-1) + T(n-r) + O(n)$$

Quick Sort Analysis

- Let us analyze some cases for r
  - **Best case:**
    - r is the median: $r = \lfloor n/2 \rfloor$
      - (we can show how to compute the median in $O(n)$ time)
  - **Worst case:**
    - $r = 1$ or $r = n$
      - When everything falls on “one side” of the pivot
  - **Something in between:**
    - say $n/10 \leq r \leq 9n/10$

Note in the worst-case analysis, we would only consider the worst case for r.
We will look at the different cases to get a sense and get some practice.

Quick Sort: Cases

- Suppose $r = n/2$ (pivot is the median element), then recurrence is:
  - $T(n) = 2T(n/2) + O(n)$, $T(1) = 1$
    - We have already solved this recurrence!
    - $T(n) = O(n \log n)$

- Suppose $r = 1$ or $r = n - 1$, then the recurrence is:
  - $T(n) = T(n-1) + T(1) + O(n), T(1) = 1$
  - What running time would this recurrence lead to?
    - Let's draw the recurrence tree…
    - $T(n) = \Theta(n^2)$ (notice: this is tight!)

Quick Sort: Cases

- Suppose $r = n/10$ (that is, you get a one-tenth, nine-tenths split)
  - What is the recurrence?
    - $T(n) = T(n/10) + T(9n/10) + O(n)$
  - Let's look at the recursion tree for this recurrence…
- We get $T(n) = O(n \log n)$, in fact, we get $\Theta(n \log n)$
- In general, the following holds (we'll show it later):
  - $T(n) = T(\alpha n) + T(\beta n) + O(n)$
    - If $\alpha + \beta < 1 : T(n) = O(n)$
    - If $\alpha + \beta = 1 : T(n) = O(n \log n)$

Quick Sort: Theory and Practice

- We can find the **median element in** $\Theta(n)$ time
    - Using divide and conquer!
    - But in practice, the constants hidden in the Oh notation for median finding are too large to use for sorting
- Common heuristic
    - Median of three (pick elements from the start, middle and end and take their median)
- If the pivot is chosen **uniformly at random**
    - quick sort runs in time $O(n \log n)$ in expectation and *with high probability*
    - We will prove this in the second half of the class

Recurrences

So far we’ve focused on divide and conquer algorithms, where we split the problem in more than one subproblem.

Question. Can you think of some examples (that you haven’t seen so far) where we split the problem into **one** smaller subproblem?

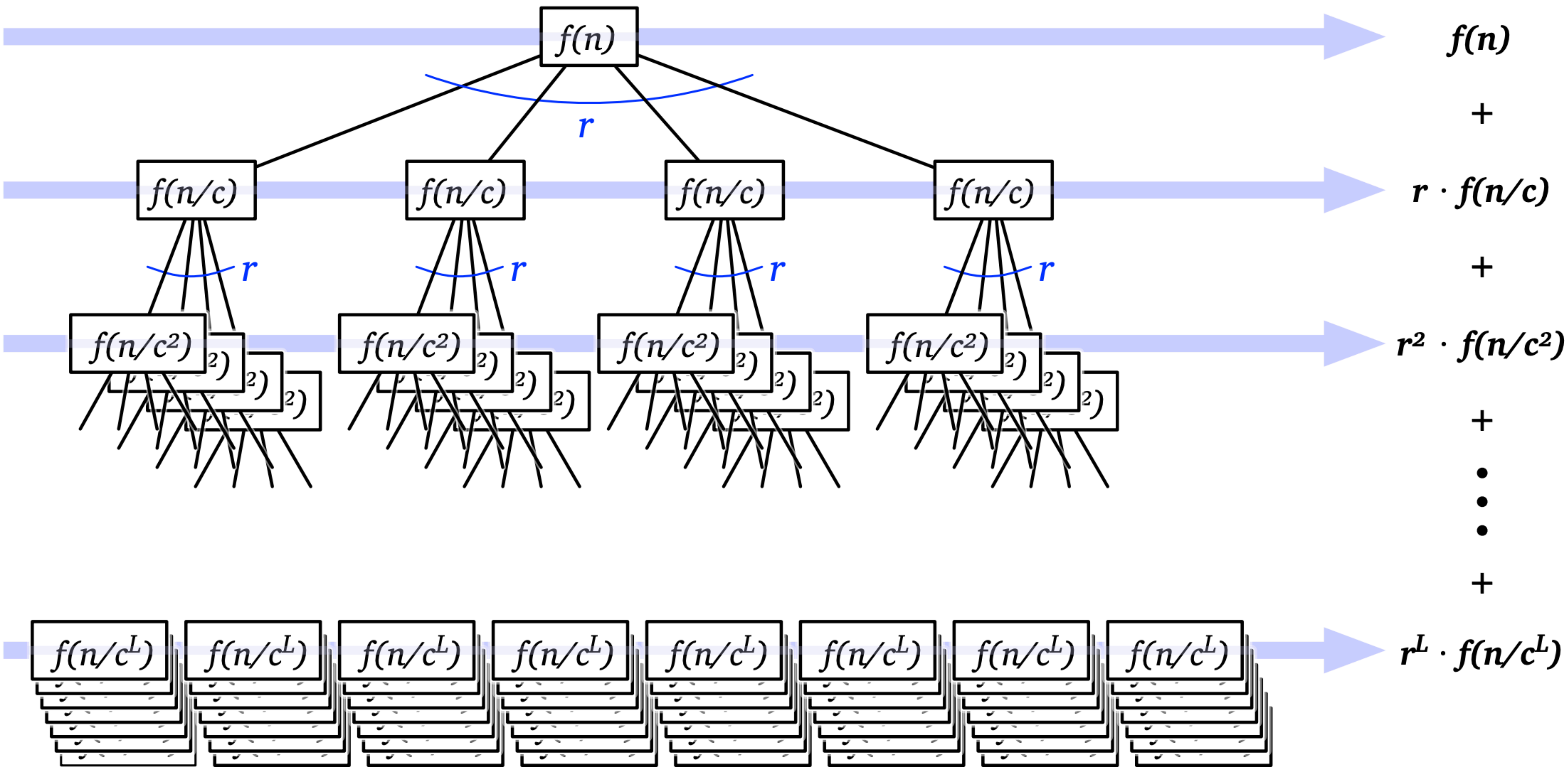
D&C: One Smaller Subproblem

- Binary search in array
  - $T(n) = T(n/2) + 1$
- Search in a binary search tree
  - $T(n) = T(n/2) + 1$
- 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?

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



A recursion tree for the recurrence $T(n) = rT(n/c) + f(n)$

- For each i, the ith level of tree has exactly r^i nodes
- Each node at level i, has cost $f(n/c^i)$

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
- The ith level of the tree has exactly r^i nodes
- And each node at level i, has cost $f(n/c^i)$

Thus, the total recurrence costs: $T(n) = \sum_{i=0}^{L} r^i \cdot f(n/c^i)$

- Here $L = \log_c n$ is the depth of the tree
- Number of leaves in the tree: $r^L = n^{\log_c r}$
- Cost at leaves: $O(n^{\log_c r} f(1))$

$$r^L = r^{\log_c n} = (2^{\log_2 r})^{\log_c n} = (2^{\log_c n})^{\log_2 r} = (2^{\log_2 n})^{\frac{\log_2 r}{\log_2 c}} = n^{\log_c r}$$

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

Selection: Problem Statement

Given an array $A[1,\ldots,n]$ of size n, find the kth smallest element for any $1 \leq k \leq n$

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

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

- **Surprisingly yes.**
- 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:
  - $A' = 12\,|\,4\,|\,5\,|\,10\,|\,7\,|\,9\,|\,8$
- 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

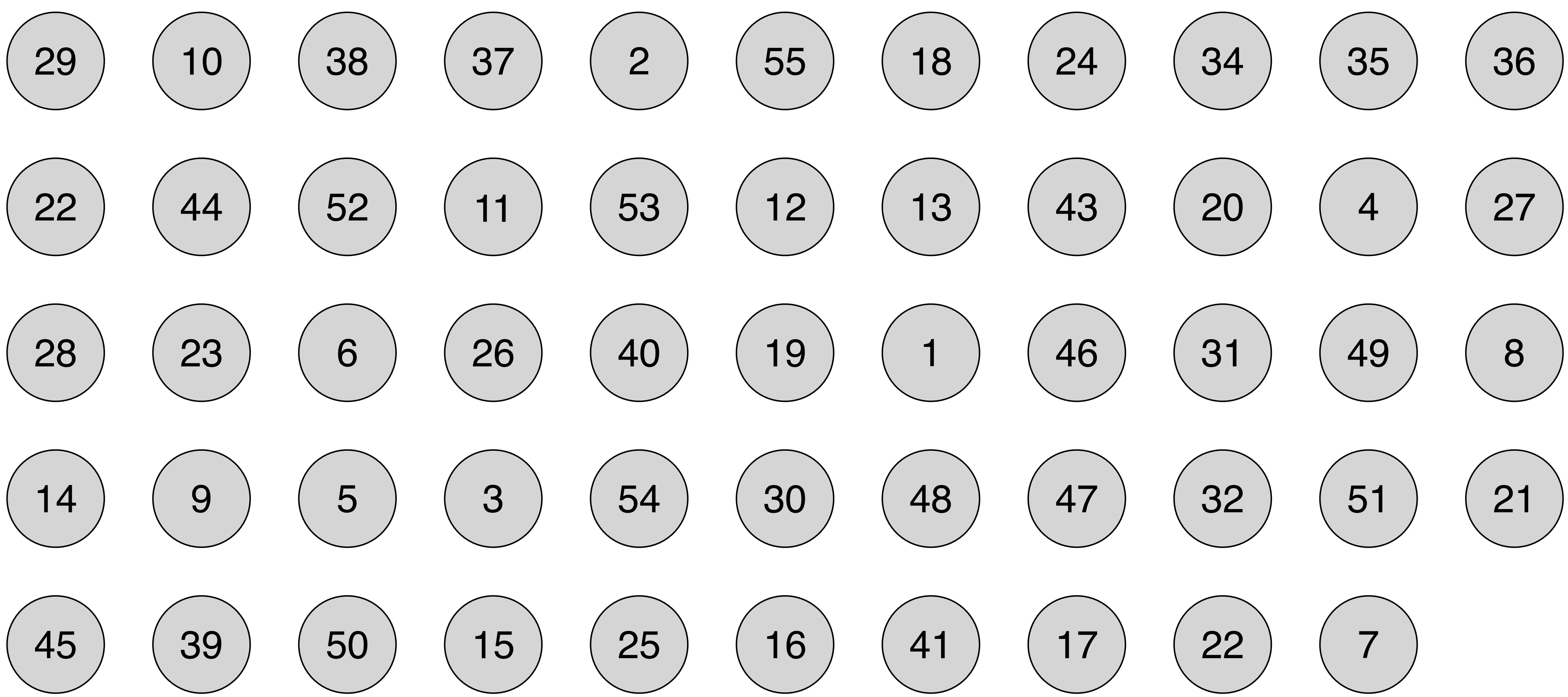
- Find a pivot that's close to the median—has a rank between $3n/10$ and $7n/10$, in time $O(n)$

- But the array is unsorted? How do we do that?

- Want to *always* be successful

Finding an Approximate Median

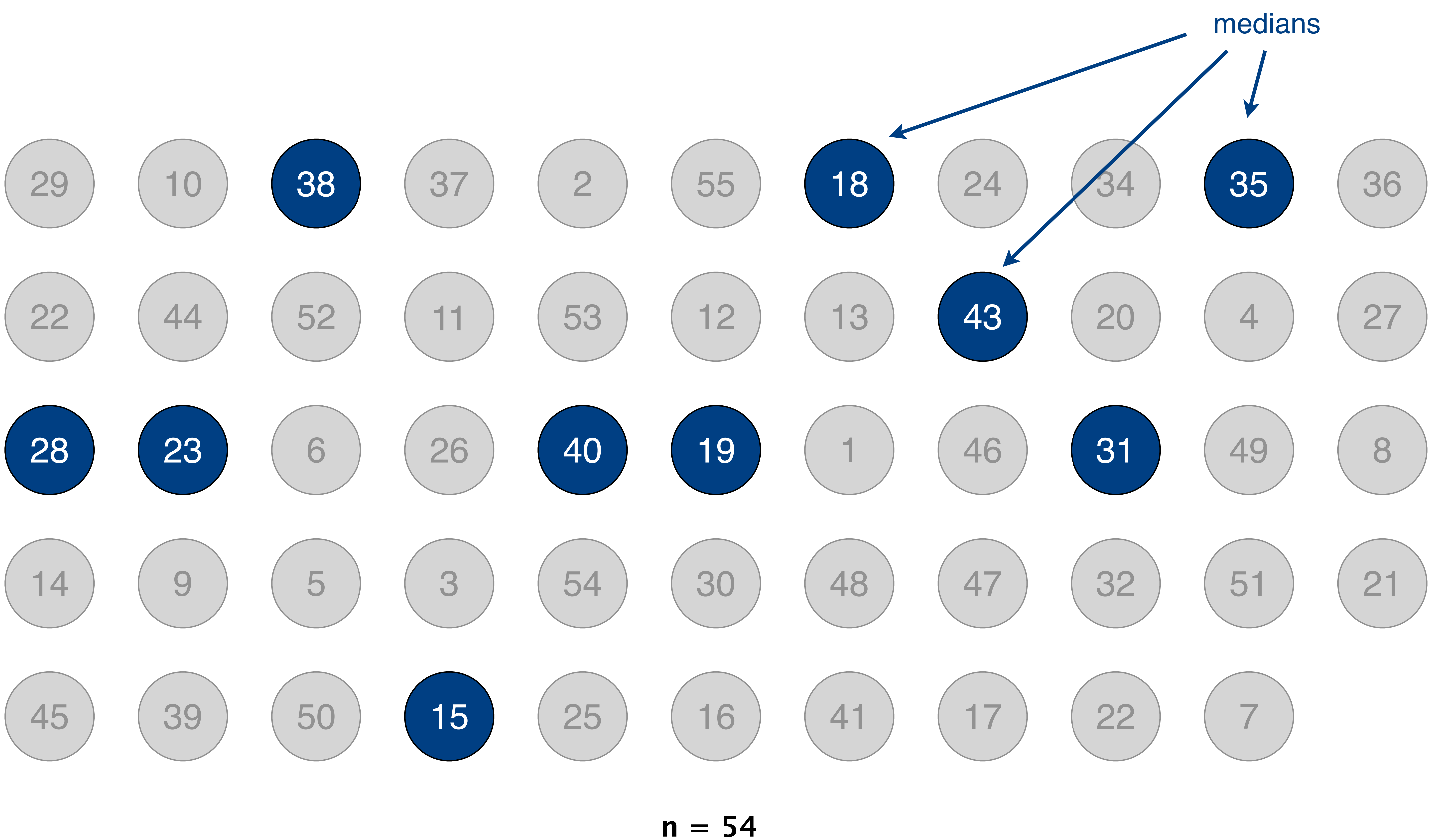
- Divide the array of size n into $\lceil n/5 \rceil$ groups of 5 elements (ignore leftovers)
- Find median of each group



n = 54

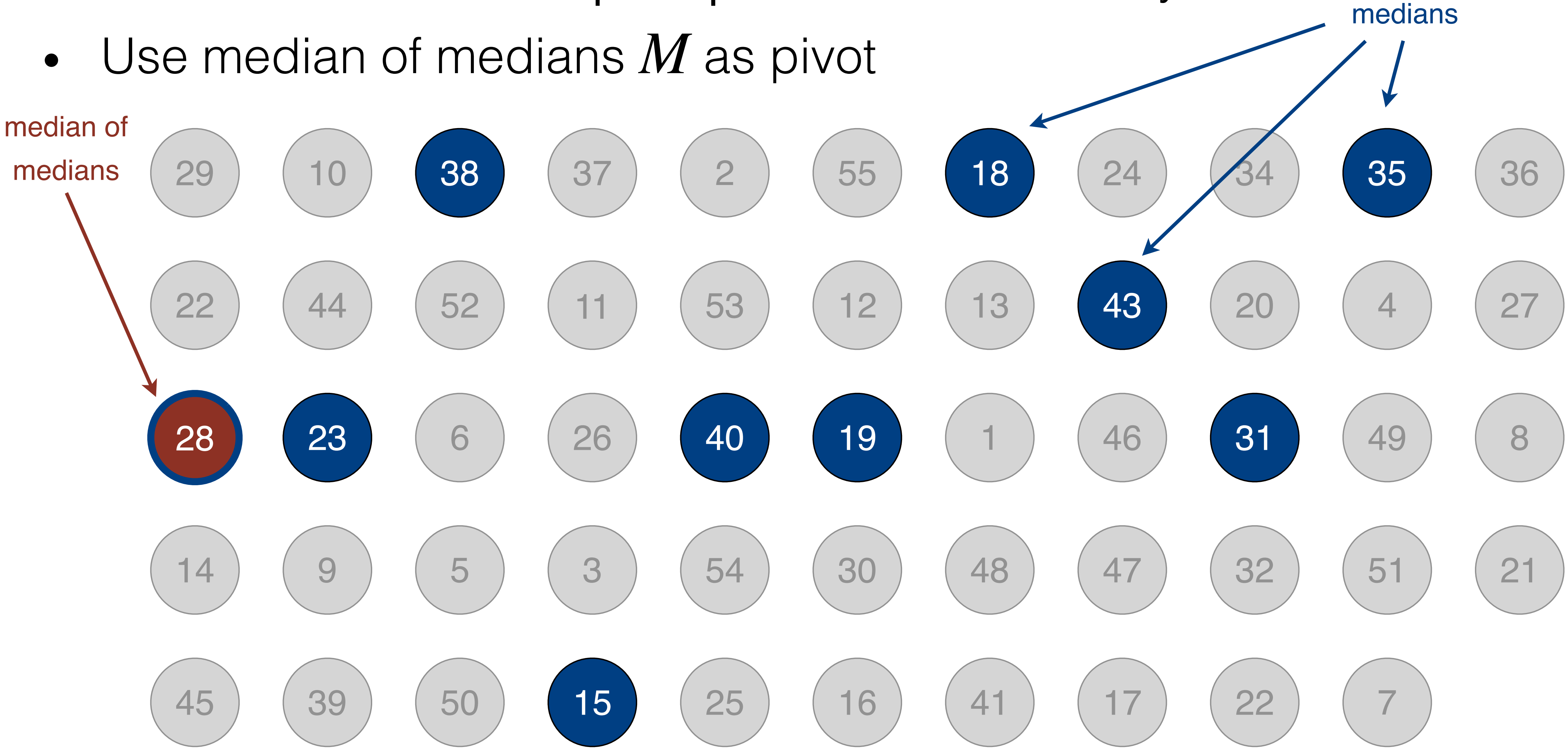
Finding an Approximate Median

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



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



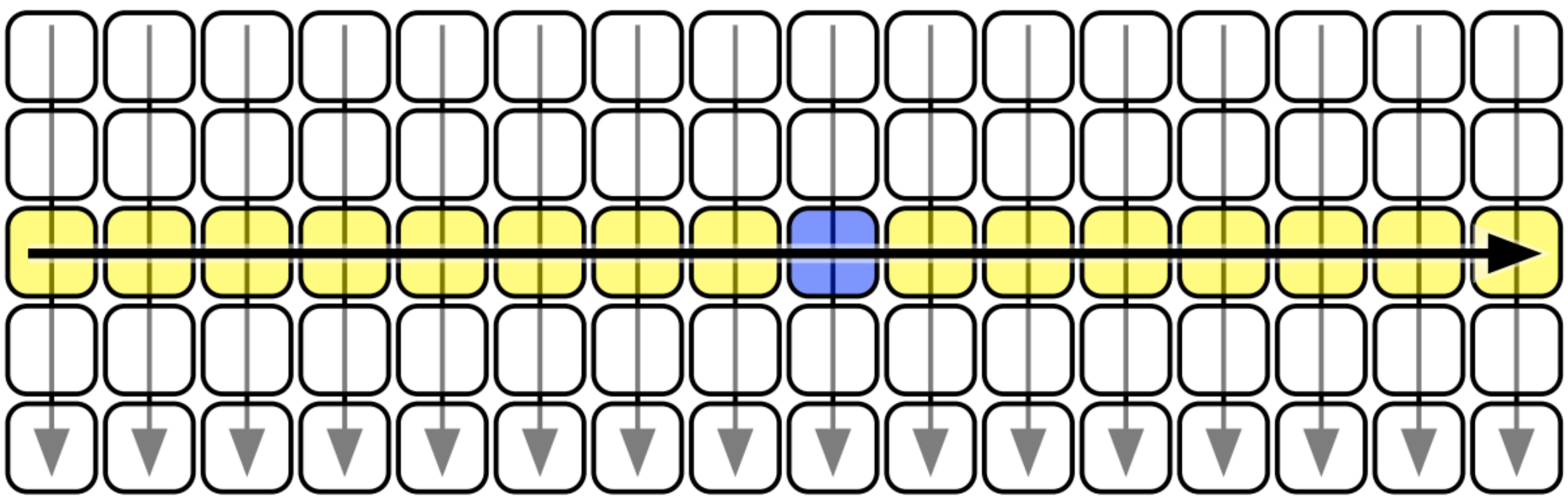
n = 54

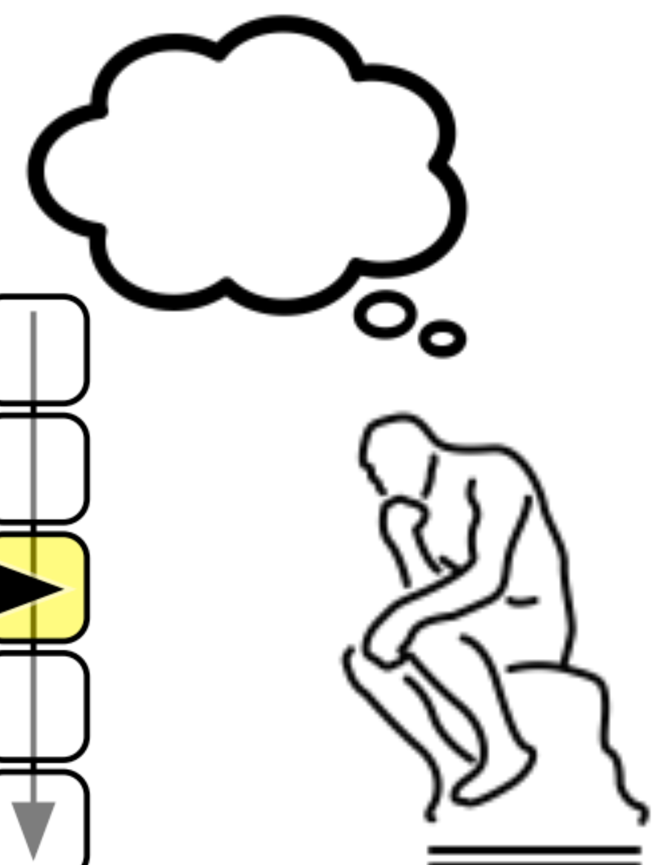
What did we gain?

- 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

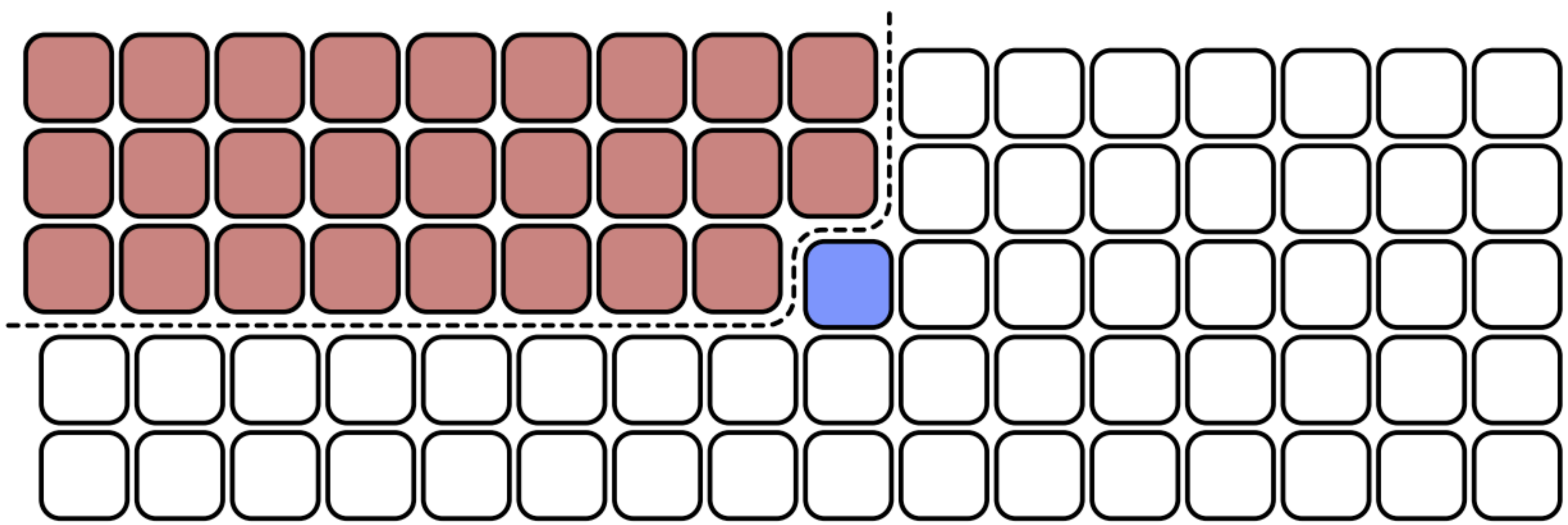
- In the 5 x n/5 grid, each column represents five consecutive elements
- **Imagine** each column is sorted top down
- **Imagine** the columns as a whole are sorted left-right
    - We don't actually do this!
- MoM is the element closest to center of grid





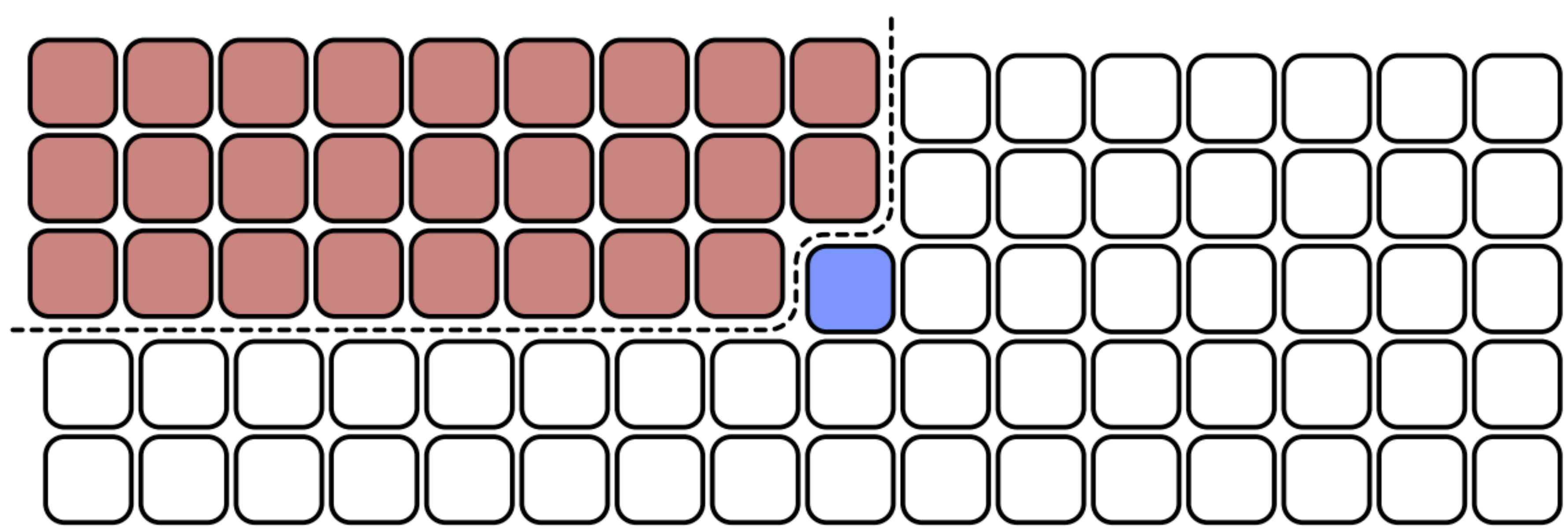
Visualizing MoM

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



Visualizing MoM

- Red cells (at least $3n/10$) in size are smaller than M
- If we are looking for an element larger than M, we can throw these out, before recursing
- Symmetrically, we can throw out $3n/10$ elements larger than M if looking for a smaller element
- Thus, the recursive problem size is at most $7n/10$



How Good is Median of Medians

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. ∎

Median of Medians Subroutine

- MoM(A, n):
  - If $n == 1$: return $A[1]$
  - Else:
    - Divide A into $\lceil n/5 \rceil$ groups
    - Compute median of each group
    - $A' \leftarrow$ group medians
    - Mom(A', $\lceil n/5 \rceil$)

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

Linear time Selection

Select (A, k):

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

- Call median of medians to find a good pivot

  $p \leftarrow$ MoM(A, n); $n = |A|$

- $r, A_{<p}, A_{>p} \leftarrow$ Partition$((A, p)$

- If $k == r$, return p

- Else:

  - If $k < r$: Select $(A_{<p}, k)$

  - Else: Select $(A_{>p}, k - r)$

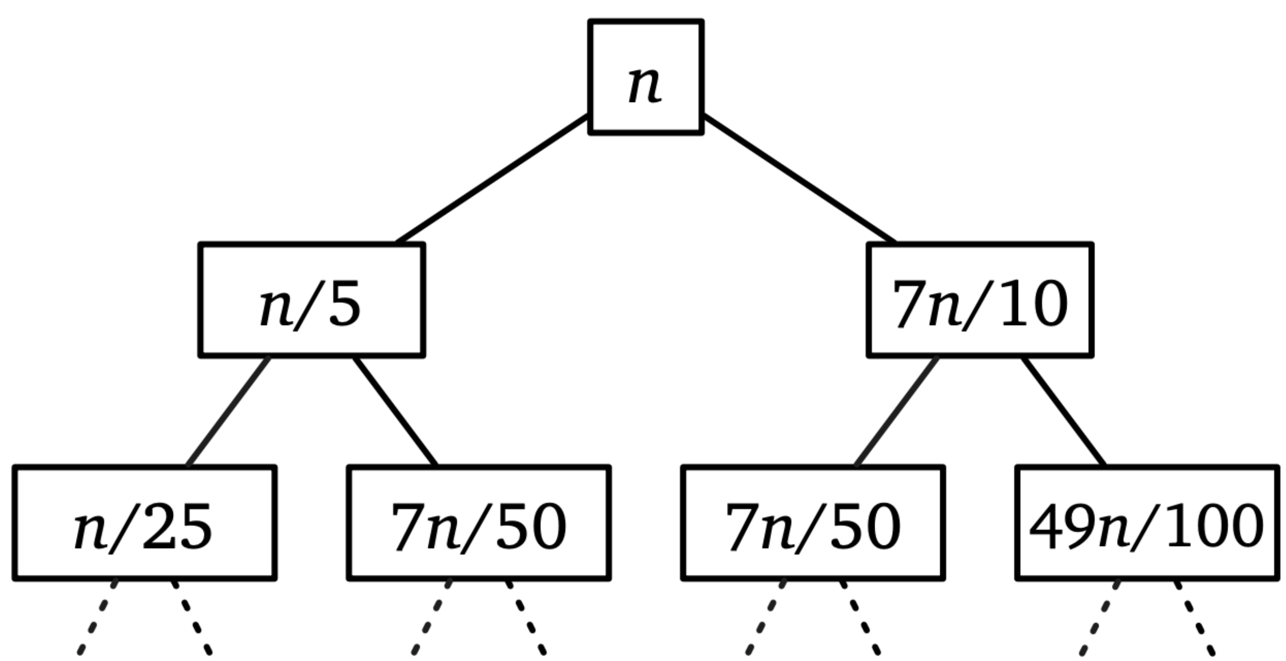
$T(n/5) + O(n)$

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 may analyze this when we do randomized algorithms

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)