Divide and Conquer: Sorting and Recurrences

Admin

- Midterm Oct 28th 10:40am
- No class Oct 28th
- Midterm is 24 hours, take home
- Current plan is that midterm will be everything through dynamic programming (not sure about network flows)
- Slides, books from course are OK; collaboration (of course) and web searches are not

Recap: Merge Sort

MERGE-SORT(L)

IF (list L has one element)

  RETURN L.

Divide the list into two halves A and B.

$A \leftarrow$ MERGE-SORT(A). $\longleftarrow$ $T(n / 2)$

$B \leftarrow$ MERGE-SORT(B). $\longleftarrow$ $T(n / 2)$

$L \leftarrow$ MERGE(A, B). $\longleftarrow$ $\Theta(n)$

RETURN L.

Merge-Sort Running Time Recurrence

- Let $T(n)$ represent how long Merge Sort takes on an input of size n
- $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + O(n)$
- **Base case:** $T(1) = 1$; often ignored
- We will ignore the floors and ceilings (we'll discuss later)

- So the recurrence simplifies to:
  - $T(n) = 2T(n/2) + O(n)$
  - The answer to this ends up being $T(n) = O(n \log n)$
  - Today we will learn different ways to derive it

Recurrences: Unfolding

Method 1. Unfolding the recurrence

- Assume $n = 2^\ell$ (that is, $\ell = \log n$)
- Because we don't care about constant factors and are only upper-bounding, we can always choose smallest power of 2 greater than that is, $n < n' = 2^\ell < 2n$

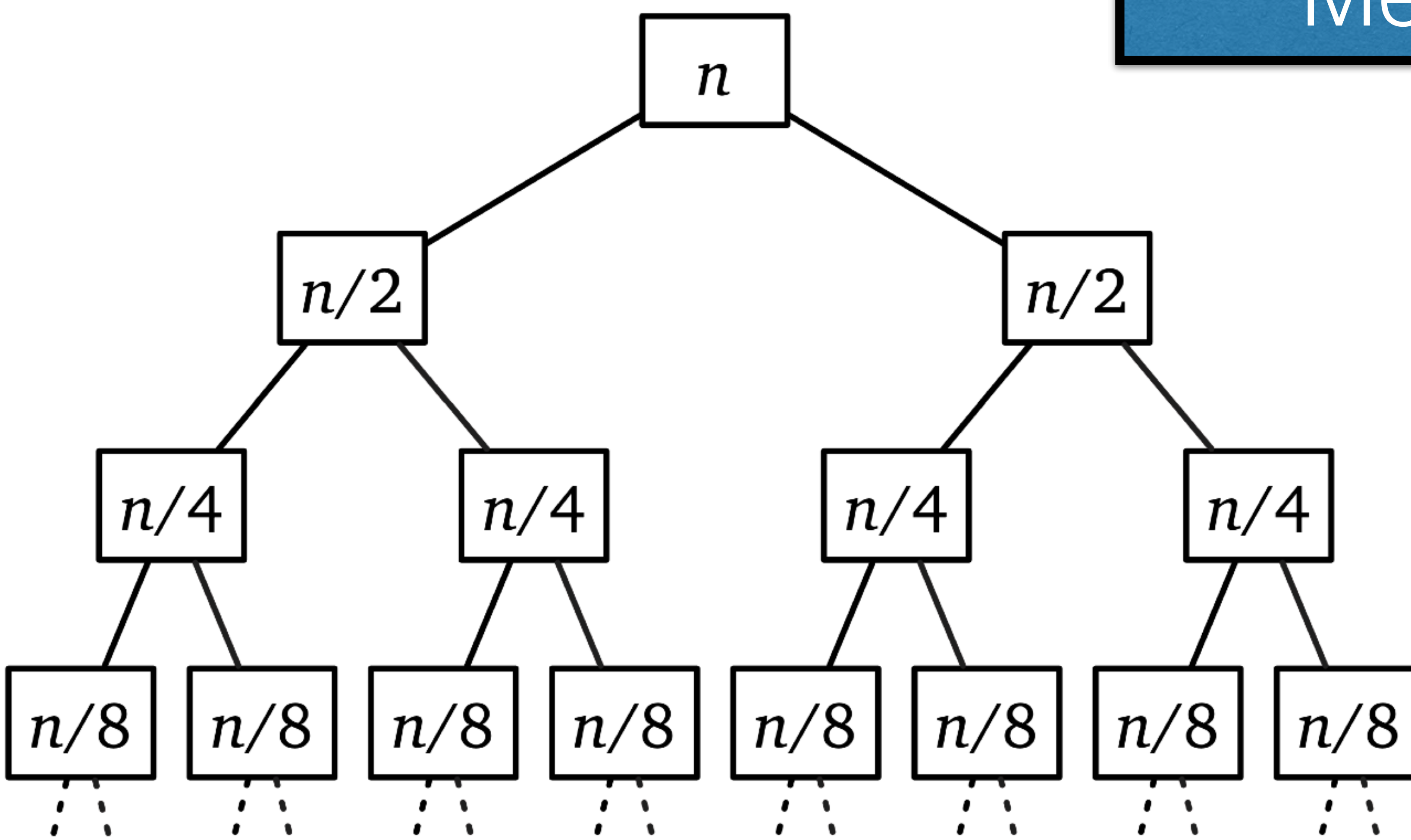
$$
\begin{aligned}
T(n) &= 2T(n/2) + cn \\
&= 2T(2^{\ell-1}) + c2^\ell \\
&= 2(2T(2^{\ell-2}) + c2^{\ell-1}) + c2^\ell = 2^2T(2^{\ell-2}) + 2 \cdot c2^\ell \\
&= 2^3T(2^{\ell-3}) + 3 \cdot 2^\ell \\
&= \ldots = 2^\ell T(2^0) + c\ell 2^\ell = O(n \log n)
\end{aligned}
$$

Recurrences: Recursion Tree

Method 2. Recursion Trees

- Work done at each level $2^i \cdot (n/2^i) = n$
- Total $\log_2 n$ levels

Recommended Method!



Recursion Tree

- This is really a method of visualization
- Very similar to unrolling, but much easier to keep track of what's going on
- It's not (quite) a proof, but generally it is sufficient for running times in this class
  - "Solve the recurrence" can be done by drawing the recursion tree and explaining the solution

Recurrences: Guess & Verify

Method 3. Guess and Verify

- Eyeball recurrence and make a guess
- Verify guess using induction

Guess & Verify Recurrences

- **Method 3.** Requires some practice and creativity
- Verification by induction may run into issues
  - Example, $T(n) = 2T(n/2) + 1$
  - Guess?
    - $T(n) \le cn$
  - Check $T(n) \le cn + 1 \not\le cn$ for any $c > 0$
  - Is the guess wrong? Not asymptotically, can fix it up by adding lower-order terms
  - New guess $T(n) \le cn - d$ (why minus?)
    - $T(n) \le cn - 2d + 1 \quad \le cn - d$ for any $d \ge 1$
  - c must be chosen large enough to satisfy boundary conditions

Divide & Conquer: Quicksort

- Choose a pivot element from the array
- Partition the array into two parts: left less than the pivot, right greater than the pivot
- Recursively quicksort the first and last 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	$\boxed{P}$	L
Partition:	A	G	O	E	I	N	L	M	$\boxed{P}$	T	X	S	R
Recurse Left:	A	E	G	I	L	M	N	O	$\boxed{P}$	T	X	S	R
Recurse Right:	A	E	G	I	L	M	N	O	$\boxed{P}$	R	S	T	X

Divide & Conquer: Quicksort

- Choose a pivot element from the array
- Partition the array into two parts: left less than the pivot, right greater than the pivot
- Recursively quicksort the first and last 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

- Choose a pivot element from the array
- Partition the array into two parts: left less than the pivot, right greater than the pivot
- Recursively quicksort the first and last subarrays
- **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

- 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 $\Theta(n \log n)$
- 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! we'll learn how in next lecture
- 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

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?**

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?
- **Idea:** easy if A and B are sorted!

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). $\longleftarrow T(n / 2)$

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

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

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

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

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
  - $T(n) = T(n/2) + 1$
- 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

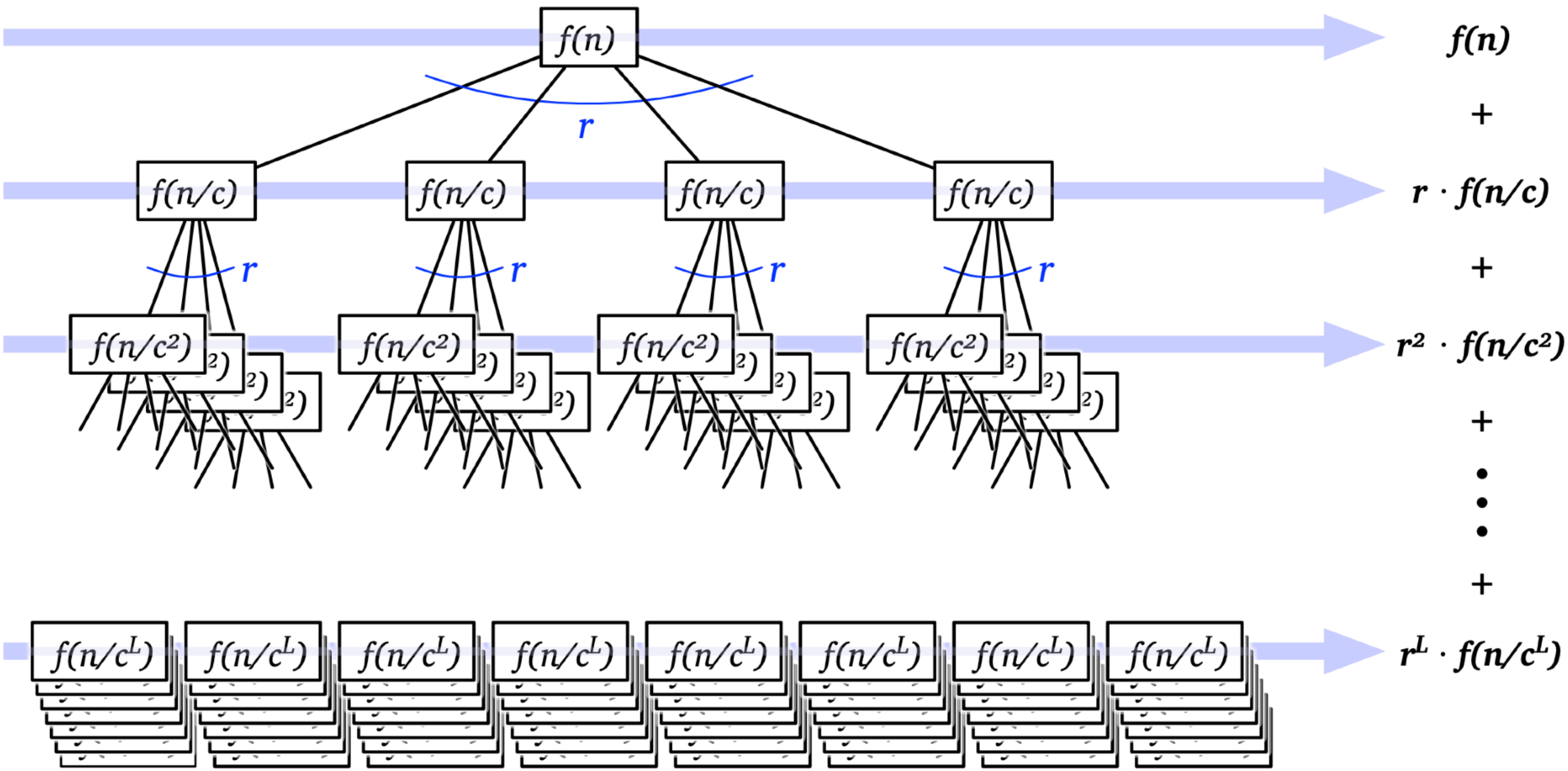


Figure 1.9. A recursion tree for the recurrence $T(n) = r\,T(n/c) + f(n)$

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: $O(n^{\log_c r} f(1))$

Easy Cases to Evaluate

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

- **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 terms is constant factor larger), then the cost at leaves dominates:
$$T(n) = O(n^{\log_c r})$$

[Akra–Bazzi '98]: Master Theorem

(Master Theorem.) Let $a \geq 1,\ b > 1$ are constants and $f(n) \geq 0$. Let $T(n)$ be defined on the nonnegative integers by the recurrence $T(n) = r\ (n/c) + f(n)$, where we interpret n/c as $\lfloor n/c \rfloor$ or $\lceil n/c \rceil$.

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

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

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: $O(n \log n)$ compares
  - Binary heap: $O(n \log k)$ compares

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

- **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 $(A_{>p}, k - r)$

Example on board

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 in the middle $8/10$ths of the array
  - What is our recurrence?
  - $T(n) \leq T(9n/10) + O(n)$
  - $T(n) = O(n)$

Formalizing a Good Pivot

- Recurrence for pivot of rank r
  - $T(n) = \max\{T(r), T(n-r)\} + O(n)$
- We don't know r, so assuming the worst:
  - $T(n) = \max\limits_{1 \le r \le n} \max\{T(r), T(n-r)\} + O(n)$
  - Simplify: use ℓ = length of recursive subproblem
  - $T(n) = \max\limits_{1 \le \ell \le n-1} T(\ell) + O(n)$
  - For what ℓ do we get a linear solution?

How to Choose a Good Pivot?

$$T(n) = \max_{1 \leq n-1} T(\ell) + O(n)$$

- If we reduce subproblem size by constant factor each time, we get a linear solution
- That is, $\ell \leq \alpha n$ for some constant $\alpha < 1$
- $T(n) \leq T(\alpha n) + O(n)$ for some constant $\alpha < 1$
- Expands to a decreasing geometric series
- Largest term at root dominates: $T(n) = O(n)$

Take away.

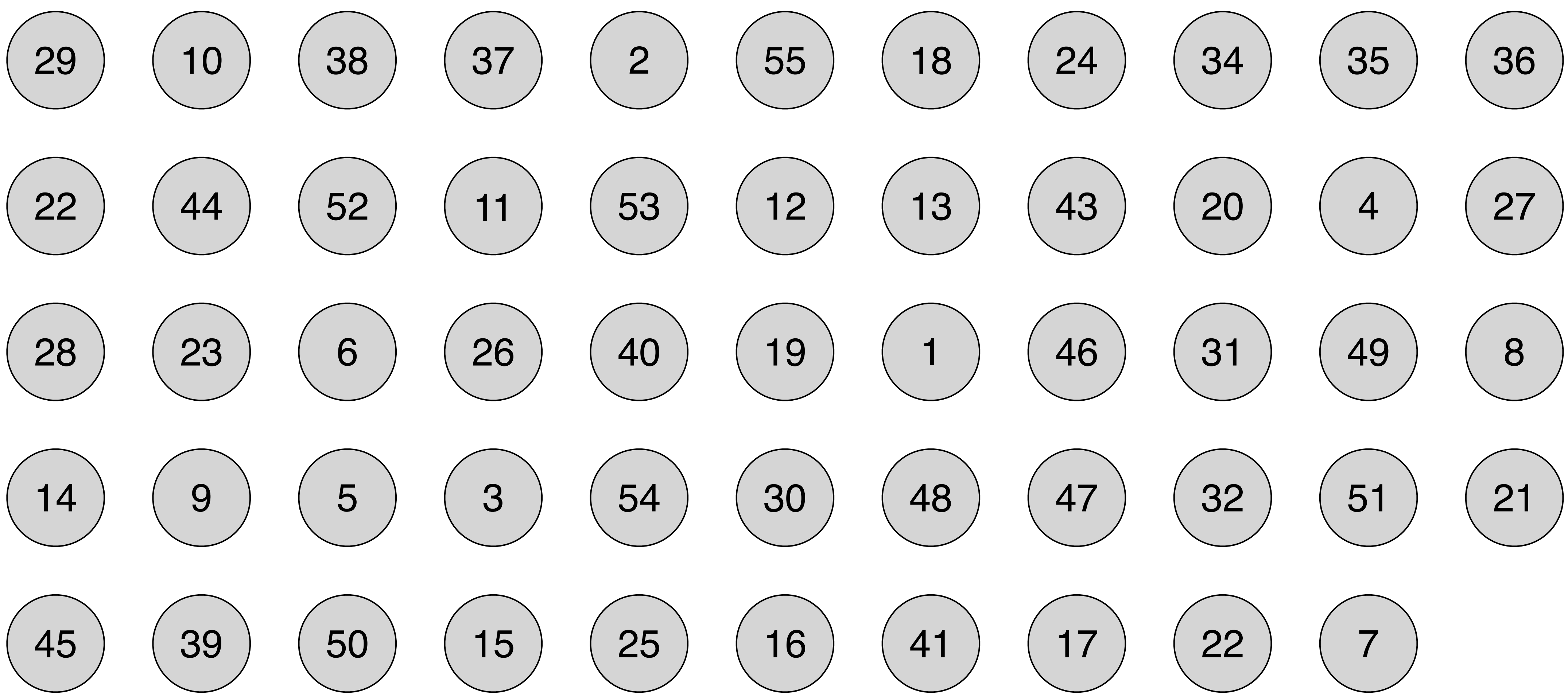
- We want a pivot that partitions such that where larger subproblem is constant factor smaller than n
- If we can find an **"approximate median"** in linear time, we can find the median in linear time as well!

Our high-level goal

- Find a pivot that's in the middle $8/10$ths of the array
- But the array is unsorted? How do we do that?
- Want to *always* be successful

Finding an Approximate Median

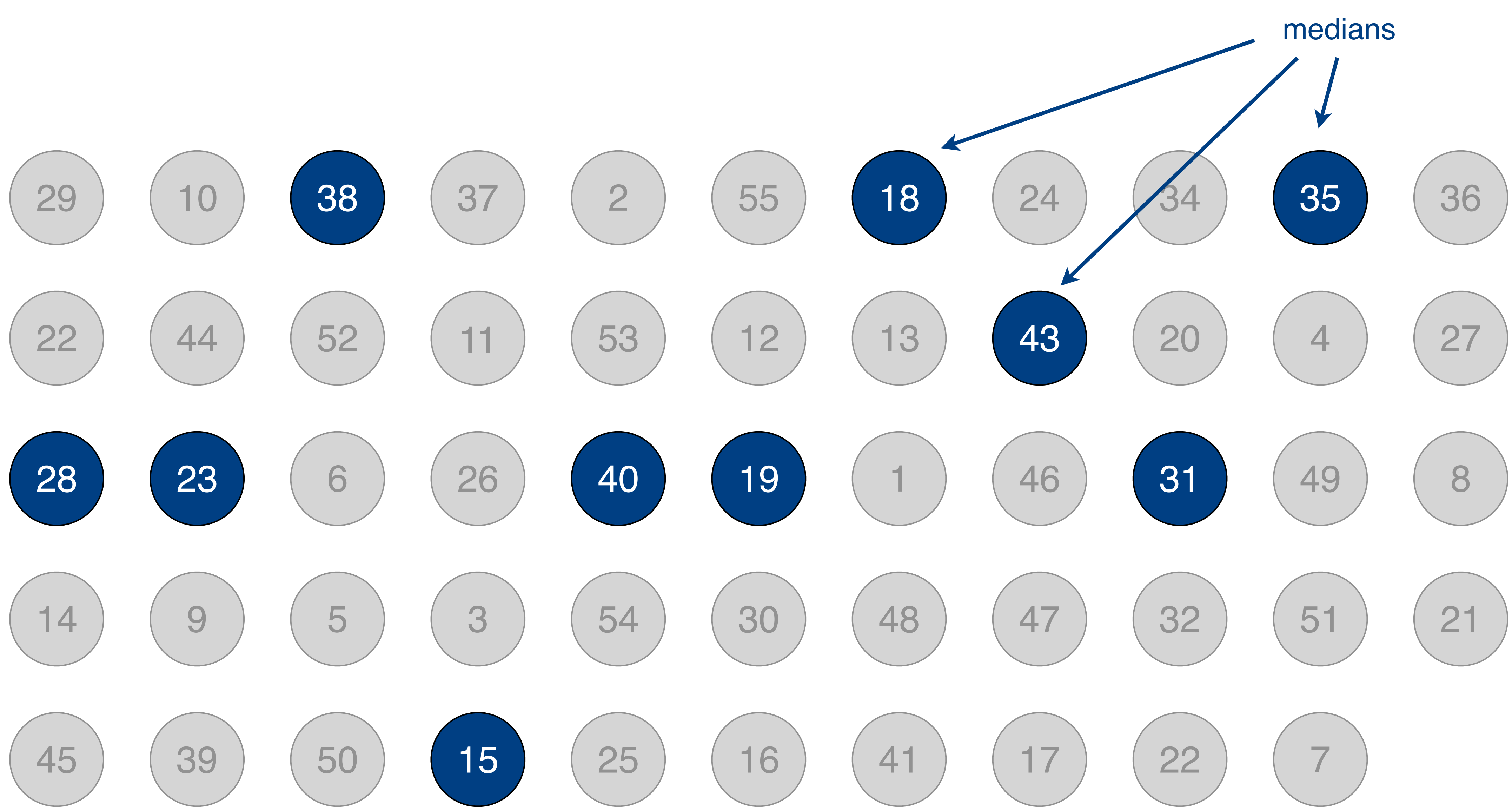
- Divide the array of size n into $\lceil n/5 \rceil$ groups of 5 elements (ignore leftovers)
- Find median of each group



n = 54

Finding an Approximate Median

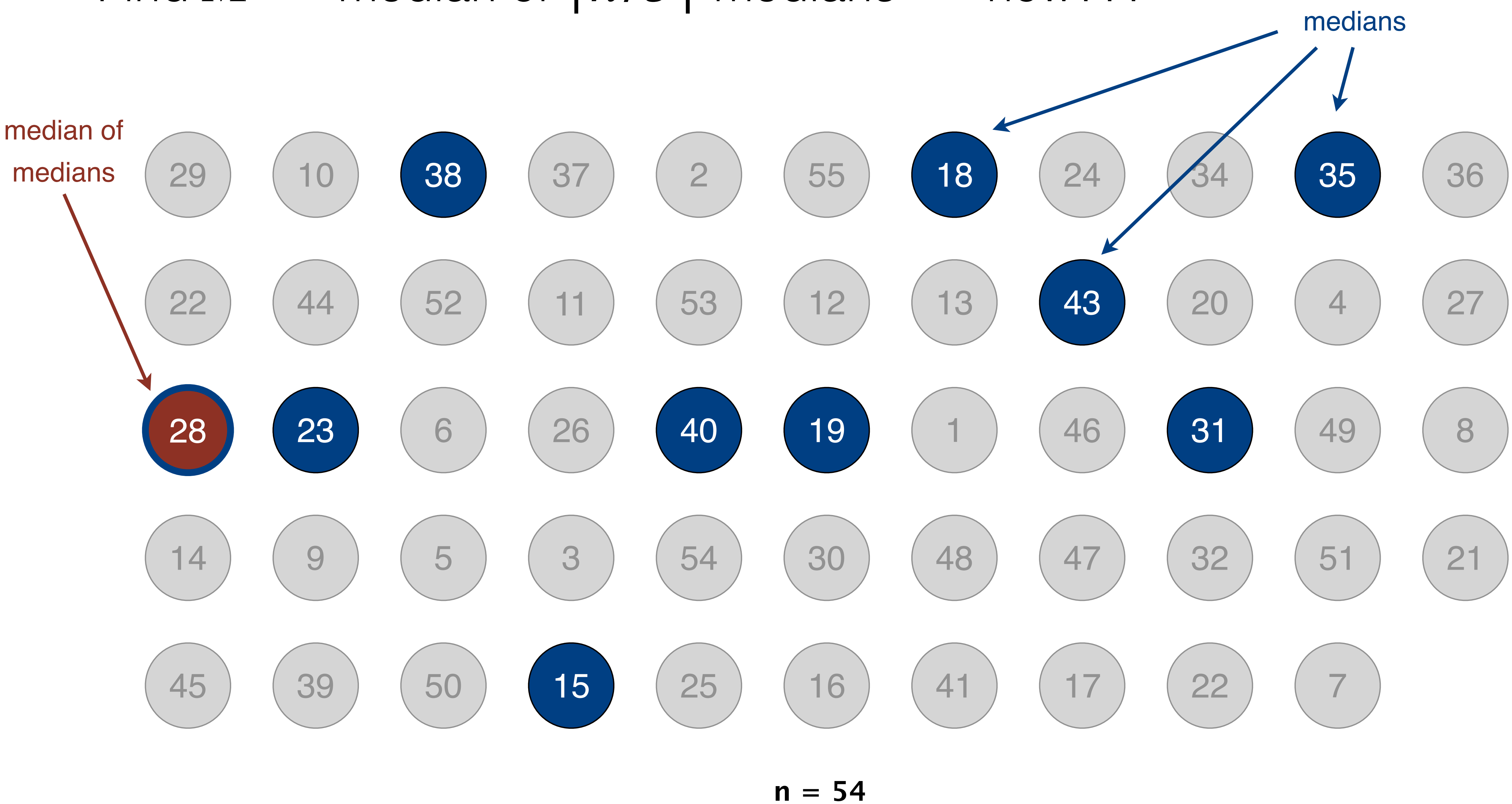
- Divide the array of size n into $\lceil n/5 \rceil$ groups of 5 elements (ignore leftovers)
- Find median of each group



n = 54

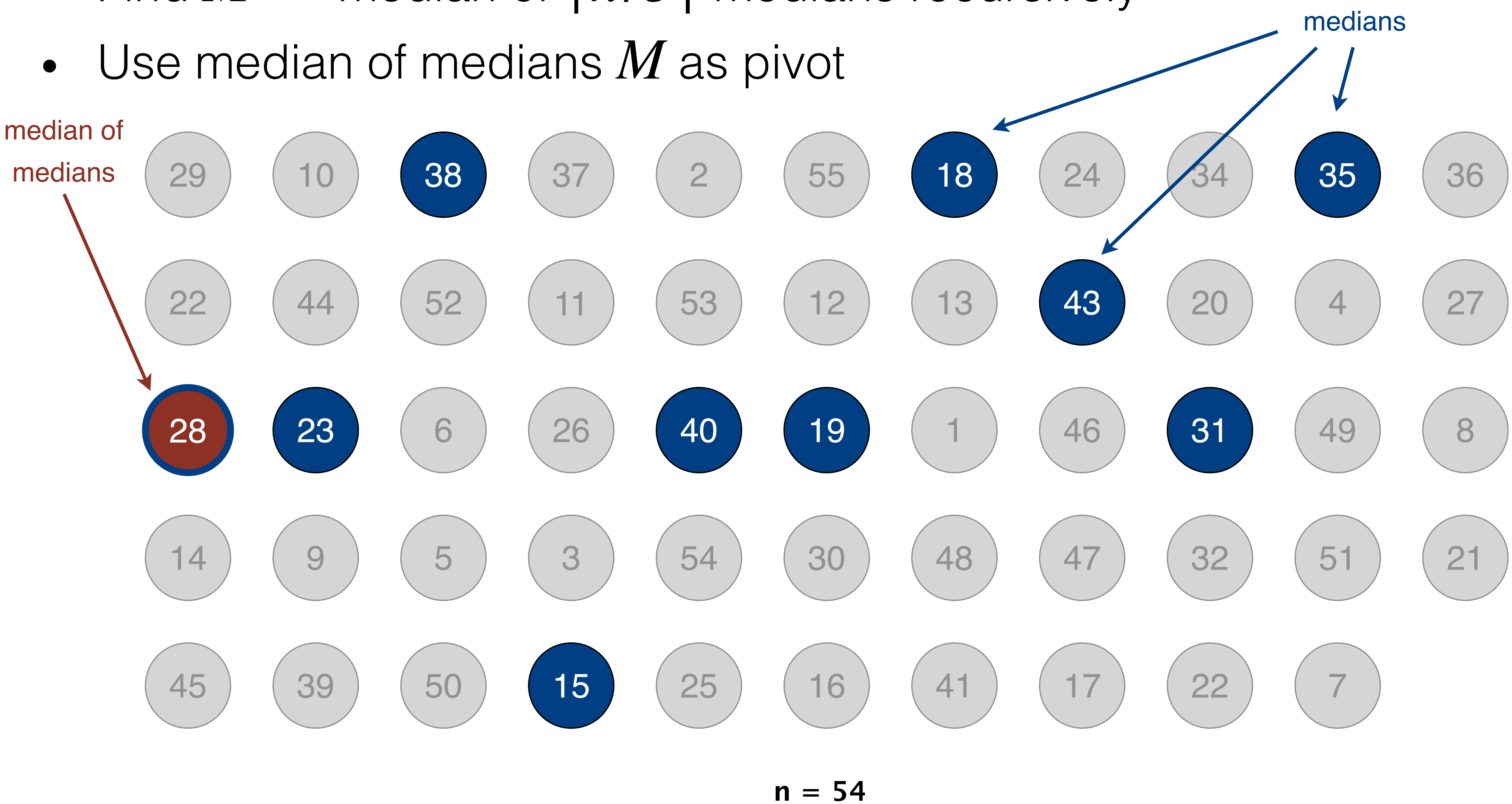
Finding an Approximate Median

- 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???



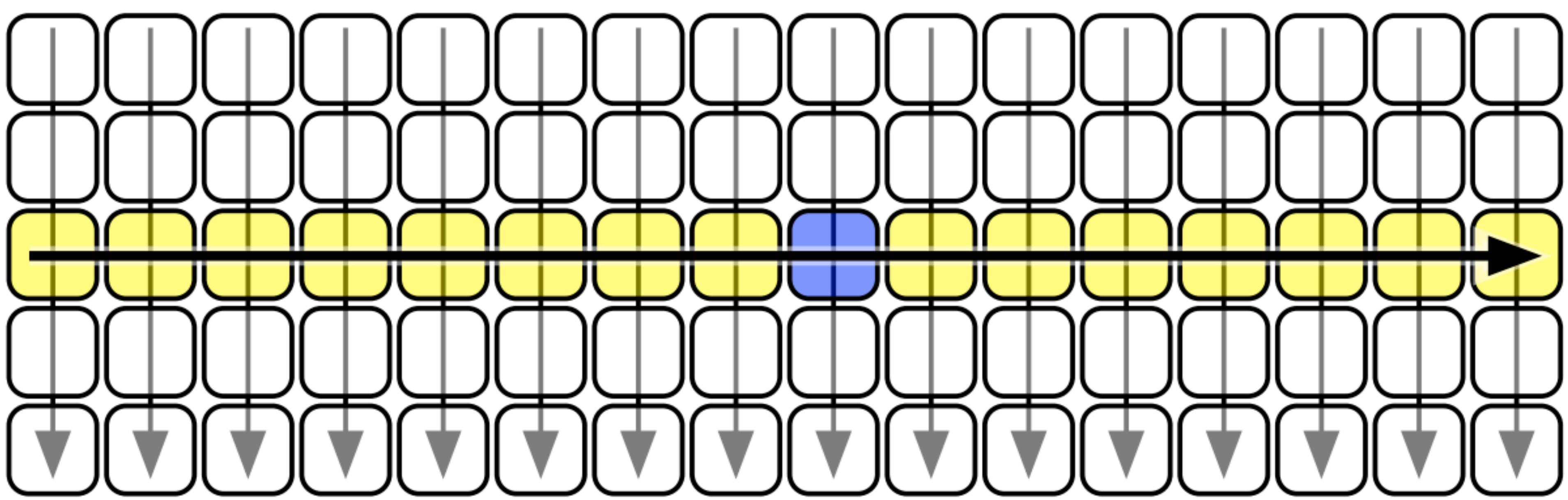
Finding an Approximate Median

- 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



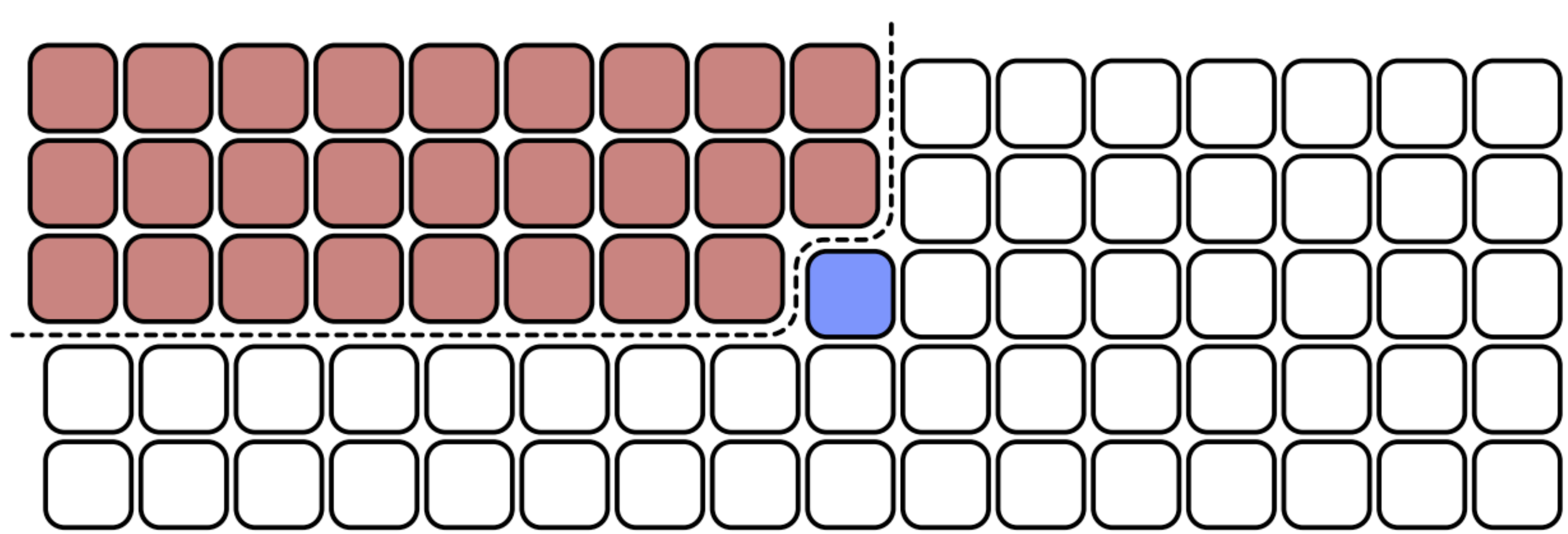
Visualizing MoM

- 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
- 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. ∎

Analysis: Running Time

- **Question.** How to compute median of median recursively?
- 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', $\lceil n/5 \rceil$)

Not recursive; $O(n)$

Not recursive; $O(n)$

Analysis: Running Time

- **Recurrence just for MoM:**
    - $T(n) = T(n/5) + O(n)$
- 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', $\lceil n/5 \rceil$)

Analysis: Overall

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

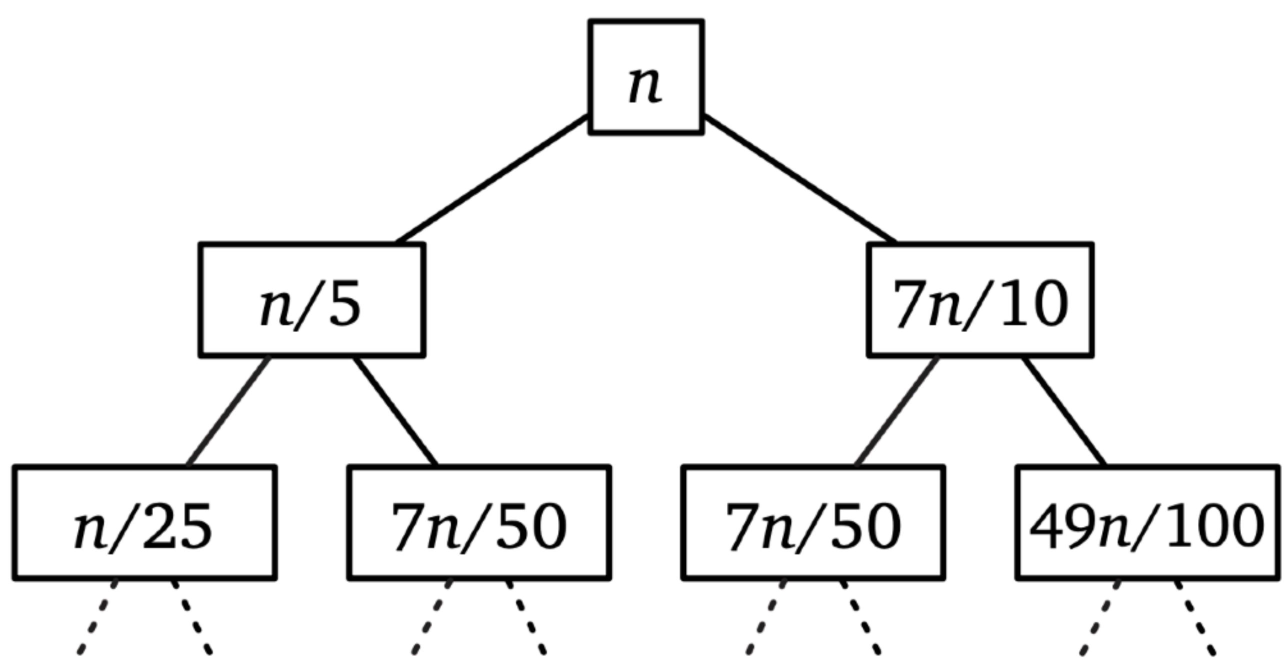
$T(n/5) + O(n)$

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

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 $n \le n' \le 2n$; then
- We can upper bound our asymptotic using n' and lower bound using $n'/2$
- In particular, let $T(n) \le T(n')$
- And $T(n) \ge T(n'/2)$
- 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)