Divide-and-Conquer Algorithms

Part Four

Announcements

- Problem Set 2 due right now.
  - Can submit by Monday at 2:15PM using one late period.
- Problem Set 3 out, due July 22.
  - Play around with divide-and-conquer algorithms and recurrence relations!
  - Covers material up through and including today's lecture.

CS161

Design and Analysis of Algorithms

$T(n) \leq T(n / 5) + T(7n / 10) + O(n)$

ndout containing
ns and identities. We
ps you navigate some

Handouts

00: Course Information
01: Syllabus
02: Problem Set Advice

Resources

Lecture Videos
Gradebook

Outline for Today

- **The Selection Problem**
  - A problem halfway between searching and sorting.
- **A Linear-Time Selection Algorithm**
  - A nonobvious algorithm with a nontrivial runtime.
- **The Substitution Method**
  - Solving recurrences the Master Theorem can't handle.

Order Statistics

- Given a collection of data, the ***k*th order statistic** is the kth smallest value in the data set.
- For the purposes of this course, we'll use zero-indexing, so the smallest element would be given by the 0^{th} order statistic.
- To give a robust definition: the kth order statistic is the element that would appear at position k if the data were sorted.

1	6	1	8	0	3	3	9

The Selection Problem

- The **selection problem** is the following: Given a data set S (typically represented as an array) and a number k, return the kth order statistic of that set.
- Has elements of searching and sorting: Want to *search* for the kth-smallest element, but this is defined relative to a sorted ordering.

32	17	41	18	52	98	24	65

- For today, we'll assume all values are distinct.

An Initial Solution

- Any ideas how to solve this?
- Here is one simple solution:
  - Sort the array.
  - Return the element at the kth position.
- Unless we know something special about the array, this will run in time $O(n \log n)$.
- Can we do better?

A Useful Subroutine: **Partition**

- Given an input array, a **partition algorithm** chooses some element p (called the **pivot**), then rearranges the array so that
  - All elements less than or equal to p are before p.
  - All elements greater p are after p.
  - p is in the position it would occupy if the array were sorted.
- The algorithm then returns the index of p.
- We'll talk about how to choose which element should be the pivot later; right now, assume the algorithm chooses one arbitrarily.

Partitioning an Array

32	17	41	18	52	98	24	65

18	17	24	32	41	65	52	98

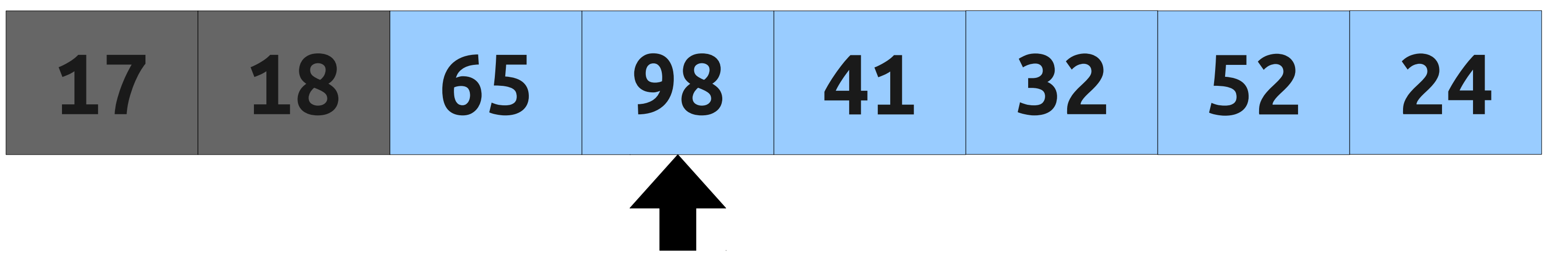
Partitioning an Array

32	17	41	18	52	98	24	65

41	65	18	32	52	17	24	98

Partitioning and Selection

- There is a close connection between partitioning and the selection problem.



- Let k be the desired index and p be the pivot index after a partition step. Then:
  - If $p = k$, return A[k].
  - If $p > k$, recursively select element k from the elements before the pivot.
  - If $p < k$, recursively select element $(k - p - 1)$ from the elements after the pivot.

```
procedure select(array A, int k):
  let p = partition(A)
  if p = k:
    return A[p]
  else if p > k:
    return select(A[0 … p-1], k)
  else (if p < k):
    return select(A[p+1 … length(A)-1], k - p - 1)
```

Some Facts

- The partitioning algorithm on an array of length n can be made to run in time $\Theta(n)$.
  - Check the Problem Set Advice handout for an outline of an algorithm to do this.
- Partitioning algorithms give no guarantee about which element is selected as the pivot.
- Each recursive call does $\Theta(n)$ work, then makes a recursive call on a smaller array.

Analyzing the Runtime

- The runtime of our algorithm depends on our choice of pivot.
- In the best-case, if we pick a pivot that ends up at position k, the runtime is $\Theta(n)$.
- In the worst case, we pick always pick pivot that is the minimum or maximum value in the array. The runtime is given by this recurrence:

$$\boxed{\begin{aligned} &T(1) = \Theta(1) \\ &T(n) = T(n-1) + \Theta(n) \end{aligned}}$$

Analyzing the Runtime

- Our runtime is given by this recurrence:

$$\boxed{\begin{aligned}&T(1) = \Theta(1)\\&T(n) = T(n-1) + \Theta(n)\end{aligned}}$$

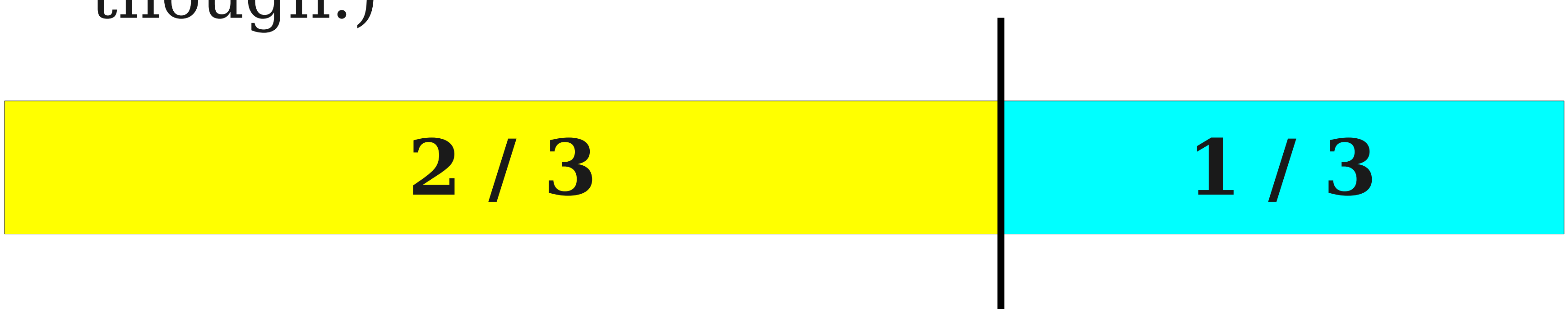
- Can we apply the Master Theorem?
- This recurrence solves to $\Theta(n^2)$.
  - First call does roughly n work, second does roughly $n - 1$, third does roughly $n - 2$, etc.
  - Total work is $n + (n - 1) + (n - 2) + \ldots + 1$.
  - This is $\Theta(n^2)$.

The Story So Far

- If we have no control over the pivot in the partition step, our algorithm has runtime $\Omega(n)$ and $O(n^2)$.
- Using heapsort, we could guarantee $O(n \log n)$ behavior.
- Can we improve our worst-case bounds?

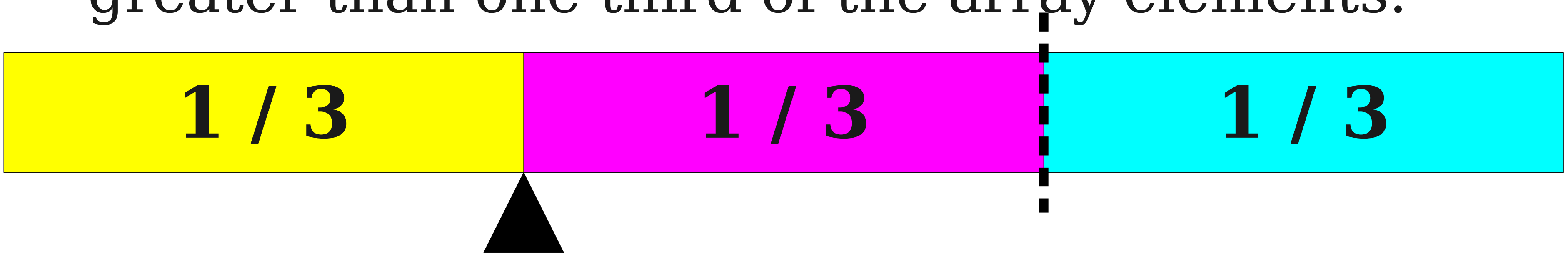
Finding a Good Pivot

- Recall: We recurse on one of the two pieces of the array if we don't immediately find the element we want.
- A good pivot should split the array so that each piece is some constant fraction of the size of the array.
  - (Those sizes don't have to be the same, though.)



An Initial Insight

- Here's an idea we can use to find a good pivot:
  - Recursively find the median of the first two-thirds of the array.
  - Use that median as a pivot in the partition step.
- **Claim:** guarantees a two-thirds / one-third split in the partition step.
- The median of the first two thirds of the array is smaller than one third of the array elements and greater than one third of the array elements.



Analyzing the Runtime

- Our algorithm
  - Recursively calls itself on the first 2/3 of the array.
  - Runs a partition step.
  - Then, either immediately terminates, or recurses in a piece of size n / 3 or a piece of size $2n$ / 3.
- This gives the following recurrence:

$$\boxed{\begin{aligned} &T(1) = \Theta(1) \\ &T(n) \leq 2T(2n / 3) + \Theta(n) \end{aligned}}$$

Analyzing the Runtime

- We have the following recurrence:

$$\boxed{\begin{aligned} &T(1) = \Theta(1) \\ &T(n) \leq 2T(2n / 3) + \Theta(n) \end{aligned}}$$

- Can we apply the Master Theorem?
- What are a, b, and d?
  - $\boldsymbol{a = 2}$, $\boldsymbol{b = 3 / 2}$, and $\boldsymbol{d = 1}$.
- Since $\log_{3/2} 2 > 1$, the runtime is

$$O\left(n^{\log_{3/2} 2}\right) \approx O\left(n^{1.26}\right)$$

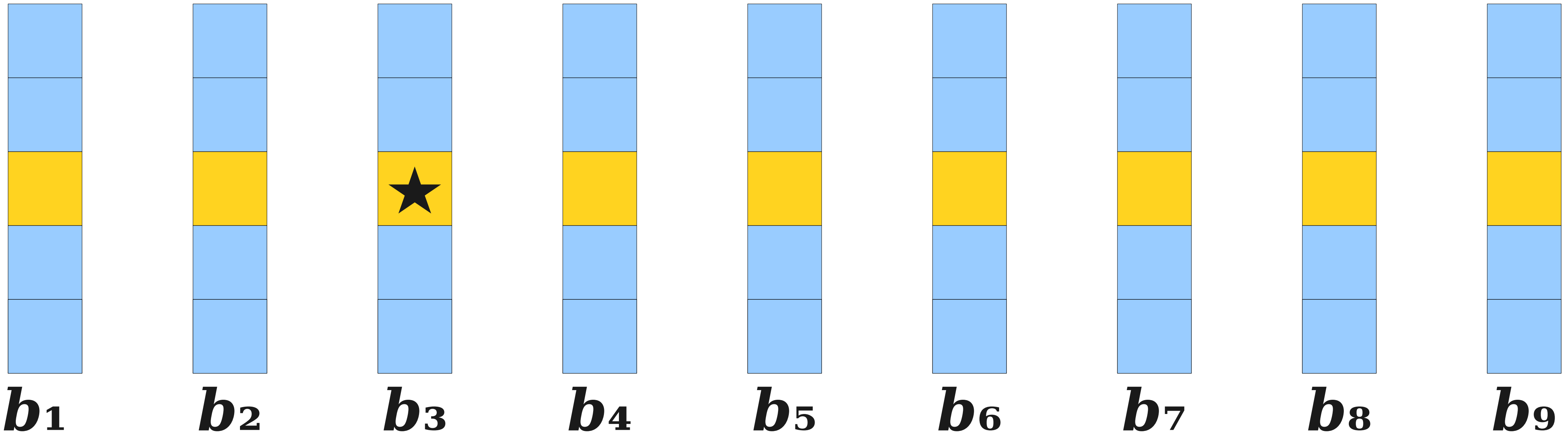
A Better Idea

- The following algorithm for picking a good pivot is due to these computer scientists:
  - Manuel Blum (Turing Award Winner)
  - Robert Floyd (Turing Award Winner)
  - Vaughan Pratt (Stanford Professor Emeritus)
  - Ron Rivest (Turing Award Winner)
  - Robert Tarjan (Turing Award Winner)
- If what follows does not seem at all obvious, *don't worry*!

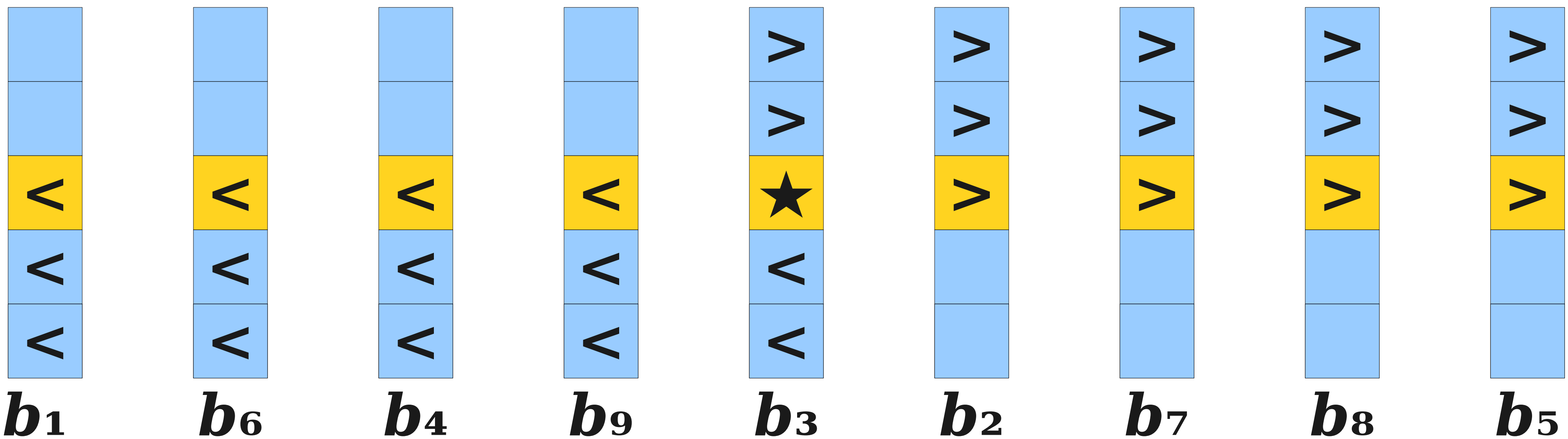
The Algorithm

- Break the input apart into block of five elements each, putting leftover elements into their own block.
- Determine the median of each of these blocks. (Note that each median can be found in time O(1), since the block size is a constant).
- Recursively determine the median of these medians.
- Use that median as a pivot.

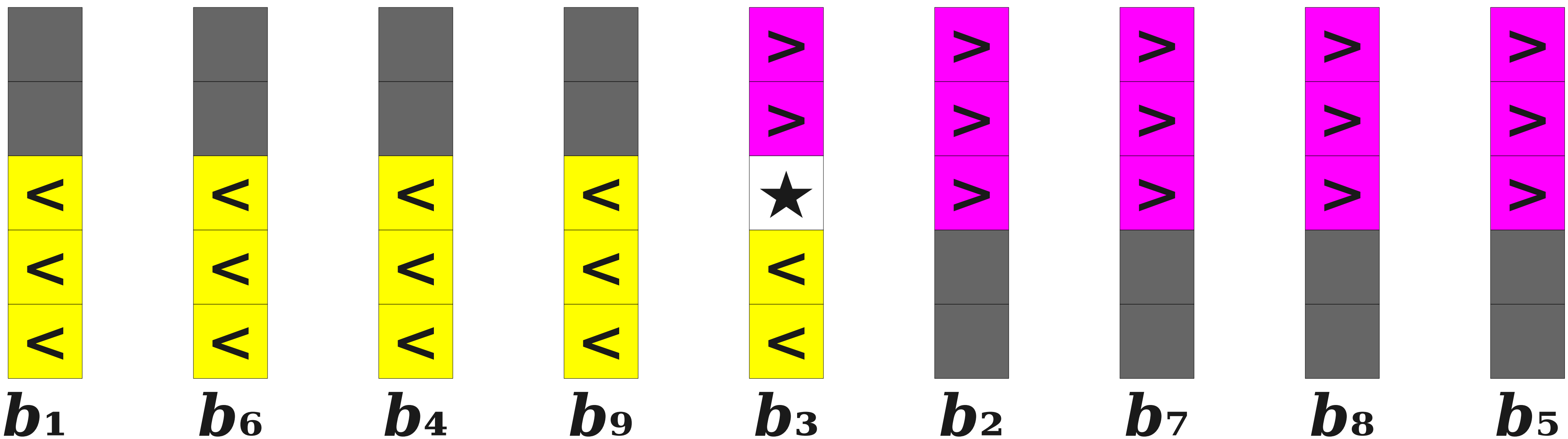
Why This Works



Why This Works



Why This Works



- The median-of-medians (★) is larger than 3/5 of the elements from (roughly) the first half of the blocks.
- ★ larger than about 3/10 of the total elements.
- ★ is smaller than about 3/10 of the total elements.
- Guarantees a 30% / 70% split.

The New Recurrence

- The median-of-medians algorithm does the following:
  - Split the input into blocks of size 5 in time $\Theta(n)$.
  - Compute the median of each block non-recursively. Takes time $\Theta(n)$, since there are about n / 5 blocks.
  - Recursively invoke the algorithm on this list of n / 5 blocks to get a pivot.
  - Partition using that pivot in time $\Theta(n)$.
  - Make up to one recursive call on an input of size at most $7n$ / 10.

$$\boxed{\begin{aligned}&T(1) = \Theta(1)\\&T(n) \le T(\lfloor n / 5 \rfloor) + T(\lfloor 7n / 10 \rfloor) + \Theta(n)\end{aligned}}$$

The New Recurrence

- Our new recurrence is

$$\boxed{\begin{array}{l} T(1) = \Theta(1) \\ T(n) \leq T(\lfloor n / 5 \rfloor) + T(\lfloor 7n / 10 \rfloor) + \Theta(n) \end{array}}$$

- Can we apply the Master Theorem here?
- Is the work increasing, decreasing, or the same across all levels?
- What do you expect the recurrence to solve to?

A Problem

- What is the value of this recurrence when $n = 4$?

$$\boxed{\begin{aligned}&T(1) = \Theta(1)\\&T(n) \le T(\lfloor n / 5 \rfloor) + T(\lfloor 7n / 10 \rfloor) + \Theta(n)\end{aligned}}$$

- ***It's undefined!***
- Why haven't we seen this before?

Fixing the Recurrence

- For very small values of *n*, this recurrence will try to evaluate T(0), even though that's not defined.
- To fix this, we will use the "fat base case" approach and redefine the recurrence as follows:

$$
\boxed{\begin{array}{ll} T(n) = \Theta(1) & \text{if } n \leq 100 \\ T(n) \leq T(\lfloor n / 5 \rfloor) + T(\lfloor 7n / 10 \rfloor) + \Theta(n) & \text{otherwise} \end{array}}
$$

- (There are still some small errors we'll correct later.)

Setting up the Recurrence

- We will show that the following recurrence is $O(n)$:

$$\begin{cases} T(n) = \Theta(1) & \text{if } n \le 100 \\ T(n) \le T(\lfloor n/5 \rfloor) + T(\lfloor 7n/10 \rfloor) + \Theta(n) & \text{otherwise} \end{cases}$$

- Making our standard simplifying assumptions about the values hidden in the Θ terms:

$$\begin{cases} T(n) \le c & \text{if } n \le 100 \\ T(n) \le T(\lfloor n/5 \rfloor) + T(\lfloor 7n/10 \rfloor) + cn & \text{otherwise} \end{cases}$$

Proving the O(n) Bound

- We cannot easily use the iteration method because of the floors and ceilings.
- The recursion-tree method is unlikely to be helpful because the tree shape is lopsided.
- Instead, we will use a technique called the **substitution method**.
- We will guess that T(n) ≤ kn for some constant k we will determine later.
- We will then use a proof by induction to show that for the right constants, T(n) ≤ kn is true.

$$\boxed{\begin{array}{ll} T(n) \le c & \text{if } n \le 100 \\ T(n) \le T(\lfloor n / 5 \rfloor) + T(\lfloor 7n / 10 \rfloor) + cn & \text{otherwise} \end{array}}$$

***Theorem:* T(n) = O(n).**

Proof: We guess that for all $n \ge 1$, $T(n) \le kn$ for some k that we will determine later; this means $T(n) = O(n)$.

We proceed by induction. As a base case, if $1 \le n \le 100$, then $T(n) \le c \le kn$ will be true as long as we pick $k \ge c$. For the inductive step, assume for some $n \ge 100$ that the claim holds for all $1 \le n' < n$. Note that $1 \le \lfloor n / 5 \rfloor < n$ and $1 \le \lfloor 7n / 10 \rfloor < n$. Then

$$\begin{aligned} T(n) &\le T(\lfloor n / 5 \rfloor) + T(\lfloor 7n / 10 \rfloor) + cn \\ &\le k\lfloor n / 5 \rfloor + k\lfloor 7n / 10 \rfloor + cn \\ &\le k(n / 5) + k(7n / 10) + cn \\ &= k(9n / 10) + cn \\ &= n(9k / 10 + c) \end{aligned}$$

If we pick k so $9k / 10 + c \le k$, then $T(n) \le kn$ holds. This is true when $c \le k / 10$, which happens when $10c \le k$. If we pick $k = 10c$, then $T(n) \le kn$, completing the induction. ■

The Substitution Method

- To use the substitution method, proceed as follows:
  - Make a guess of the form of your answer (for example, kn or $k_1\, n \log_{k_2} n$.
  - Proceed by induction to prove the bound holds, noting what constraints arise on the values of your undetermined constants.
  - If the induction succeeds, you will have values for your undetermined constants and are done.
  - If the induction fails, you either need to strengthen your assumption about the function or relax your bound.

Nitty-Gritty Details

- The recurrence we just analyzed was *close* to the real recurrence, but is slightly inaccurate due to some rough assumptions about the split size.
- Let's try to get a tighter bound on the split size.

A Better Analysis

- There are $\lceil n / 5 \rceil$ blocks, including the leftover elements.
- $\lceil \lceil n / 5 \rceil / 2 \rceil$ blocks have elements greater than or equal to the median-of-medians ★.
- The block containing ★ and the very last block are special cases. If we ignore them, there are at least $\lceil \lceil n / 5 \rceil / 2 \rceil - 2$ “normal” blocks greater than the median block.

A Better Analysis

- Each of the $\lceil\lceil n / 5\rceil / 2\rceil - 2$ "normal" blocks contributes three elements greater than ★.
- If we let X denote the number of elements greater than ★, we get

$$\begin{aligned} X &\ge 3(\lceil\lceil n / 5\rceil / 2\rceil - 2) \\ &\ge 3(n / 10 - 2) \\ &= 3n / 10 - 6 \end{aligned}$$

- Our recursive call can be on a subarray of size

$$\begin{aligned} n - X &\le n - (3n / 10 - 6) \\ &\le 7n / 10 + 6 \end{aligned}$$

The Real Recurrence Relation

- The most accurate recurrence relation for our algorithm is the following:

$$T(n) \leq c \qquad \text{if } n \leq 100$$
$$T(n) \leq T(\lceil n / 5 \rceil) + T(\lceil 7n / 10 + 6 \rceil) + cn \qquad \text{otherwise}$$

- Let's see if we can prove this is O(n).

$$\boxed{\begin{array}{ll} T(n) \le c & \text{if } n \le 100 \\ T(n) \le T(\lceil n/5 \rceil) + T(\lceil 7n/10 + 6 \rceil) + cn & \text{otherwise} \end{array}}$$

***Theorem:* T(n) = O(n).**

Proof: We will prove that for some constant k to be chosen later, $T(n) \le kn$ for all $n \ge 1$; $T(n) = O(n)$ follows. We proceed by induction. As our base case, if $1 \le n \le 100$, then $T(n) \le c \le kn$ if we choose $k \ge c$.

For the inductive step, assume that for some $n \ge 100$, the claim holds for all $1 \le n' < n$. Note that if $n \ge 100$ that $\lceil n / 5 \rceil < n$ and $\lceil 7n / 10 + 6 \rceil < n$. Therefore:

$$\begin{aligned} T(n) &\le T(\lceil n / 5 \rceil) + T(\lceil 7n / 10 + 6 \rceil) + cn \\ &\le k\lceil n / 5 \rceil + k(\lceil 7n / 10 + 6 \rceil) + cn \\ &\le k(n / 5 + 1) + k(7n / 10 + 6 + 1) + cn \\ &= 9kn / 10 + 8k + cn \\ &= kn + (8k + cn - kn / 10) \end{aligned}$$

If $(8k + cn - kn / 10) \le 0$, then $T(n) \le kn$. It's left as an exercise to the reader to check that this is true if we pick $k = 50c$. Thus $T(n) \le kn$, completing the induction. ■

A Note on O(n)

- This linear-time selection algorithm does run in time O(n), but there is a *huge* constant factor hidden here.
- Two reasons:
  - Work done by each call is large; finding the median of each block requires nontrivial work.
  - Problem size decays slowly across levels; each layer is roughly 10% smaller than its predecessor.
- Is there a way to get O(n) behavior without such a huge constant factor?

Next Time

- Randomized Algorithms
- A Faster Selection Algorithm
- Linearity of Expectation