Divide and Conquer II

Summer 2017 • Lecture 2

A Few Notes

Homework 1

- ~~Released tomorrow night.~~
- ~~Due Friday 7/7 at 11:59 p.m. on Gradescope.~~
- Remember, you must type your solutions!
- ~~You can use a max of 2 out of your 3 late days.~~
- Will cover material from Lectures 1 and 2.

~~Piazza~~

- ~~Excellent questions and discussion on Piazza~~!

Outline for Today

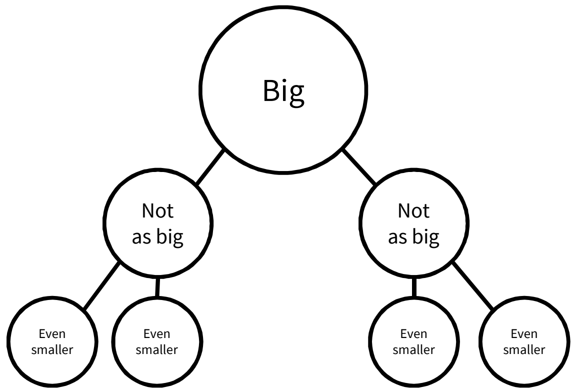
Divide and Conquer II
- [Example] Mergesort, revisited
- [Example] Integer multiplication
- Solving recurrences
  - Recursion Tree method
  - Iteration method
  - Master method
- [Example] Median and selection
  - Substitution method

Mergesort

Divide and Conquer

Divide: break current problem into smaller problems.

Conquer: solve the smaller problems and collate the results to solve the current problem.



Mergesort

Let's use divide and conquer to improve upon insertion sort!

4	8	1	5	3	2	6	7

Let's sort an unsorted list of numbers **A**.

1	4	5	8	2	3	6	7

Recursively sort each half, **`A[0:3]`** and **`A[4:7]`**, separately.

1	2	3	4	5	6	7	8

Merge the results from each half together.

Mergesort

```
algorithm mergesort(list A):
  if length(A) ≤ 1:
    return A
  let left = first half of A
  let right = second half of A
  return merge(
    mergesort(left),
    mergesort(right)
  )
```

Runtime: O(nlogn)

Mergesort

```
algorithm merge(list A, list B):
  let result = []
  while both A and B are nonempty:
    if head(A) < head(B):
      append head(A) to result
      pop head(A) from A
    else:
      append head(B) to result
      pop head(B) from B
  append remaining elements in A to result
  append remaining elements in B to result
  return result
```

Total work: O(a+b), where a and b are the lengths of lists A and B.

Mergesort

Question 1 How do we prove this algorithm always sorts the input list?

Question 2 How efficiently does this algorithm sort the input list?

Analyzing Runtime

Here's our first **recurrence relation**,

$$T(0) = T(1) = \Theta(1)$$
$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n)$$

Assumption 1: n is a power of two.

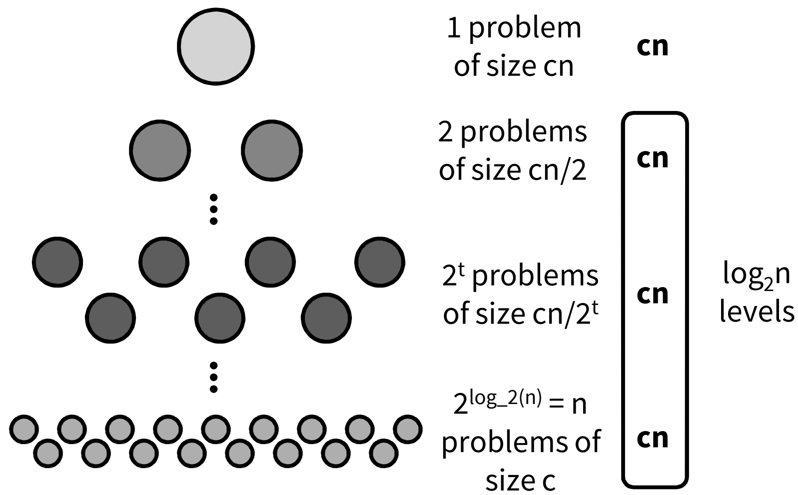
Why is it ok to make this assumption?

$$\cancel{T(0) = \Theta(1)}$$
$$T(1) = \Theta(1) = c_1$$
$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n)$$
$$= 2T(n/2) + c_2 n$$

Assumption 2: Let $c = \max\{c_1, c_2\}$

$$T(1) \le c$$
$$T(n) \le 2T(n/2) + cn$$

Recursion Tree Method



Total work: cn $\log_2$n + cn = **`O(nlogn)`**

Iteration Method

Recall, our recurrence relation:

$T(1) \leq c$

$T(n) \leq 2T(n/2) + cn$

$T(n) \leq 2 \cdot T(n/2) + cn$
$\leq 2 \cdot (2T(n/4) + cn/2) + cn$
$= 4 \cdot T(n/4) + 2cn$
$\leq 4 \cdot (2T(n/8) + cn/4) + 2cn$
$= 8 \cdot T(n/8) + 3cn$
…
$\leq 2^k T(n/2^k) + kcn$

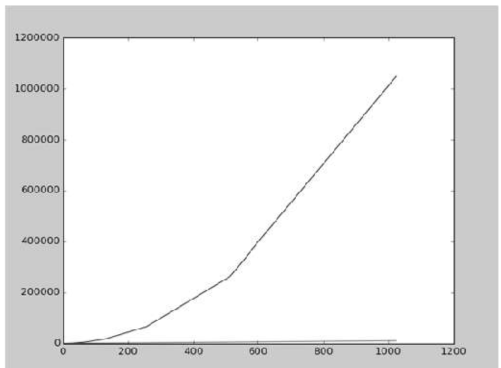
What is k? It's the number of times to divide n by 2 to get 1.

So $k = \log_2 n$

$T(n) \leq 2^k T(n/2^k) + kcn$
$= 2^{\log\_2(n)} T(n/2^{\log\_2(n)}) + cn\log_2 n$
$= nT(1) + cn\log_2 n$
$\leq cn + cn\log_2 n$
$=$ **`O(nlogn)`**

Analyzing Runtime

The best and worst-case runtime of `mergesort` is Θ(n log n).
The worst-case runtime of `insertion_sort` was Θ(n^2).
THIS IS A HUGE IMPROVEMENT!!



Integer Multiplication

Integer Multiplication

1 x 2 = 2

13 x 24 = 312

1357 x 2468 = 3,349,076

$\underbrace{13579246801593726048}_{n}$ x 24680135792604815937 = ???

How long would it take you to solve this problem?

About n^2 one-digit operations.

- At most n^2 multiplications
- At most n^2 additions (for carries)
- Addition of n different 2n-digit numbers

Integer Multiplication

Let's break up a 4-digit integer: 1357 = 13·100 + 57

1357 x 2468

= (13·100 + 57)(24·100 + 68)

= (13 x 24)·10000 + (13 x 68 + 57 x 24)·100 + (57 x 68)

One 4-digit multiplication →
Four 2-digit multiplications

Integer Multiplication

Let's break up an n-digit integer: $j = a \cdot 10^{n/2} + b$

$j \times k$

$= (a \cdot 10^{n/2} + b)(c \cdot 10^{n/2} + d)$

$= (a \times c) \cdot 10^{n} + (a \times d + b \times c) \cdot 10^{n/2} + (b \times d)$

One n-digit multiplication →
Four (n/2)-digit multiplications

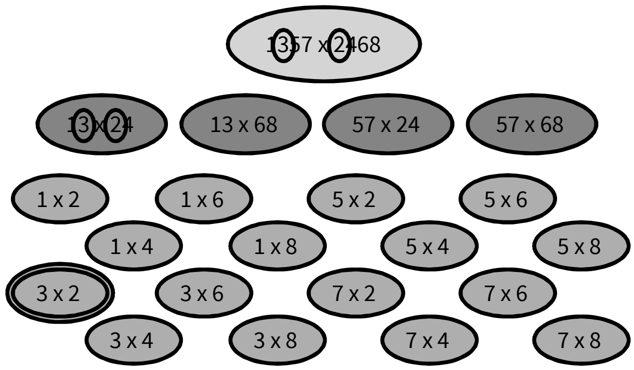
Integer Multiplication

```
algorithm naive_recursive_multiply(j, k):
  Rewrite j as a·10^(n/2) + b
  Rewrite k as c·10^(n/2) + d
  Recursively compute a·c, a·d, b·c, b·d
  Add them up (with shifts) to get j·k
```

Runtime: $O(n^2)$

Analyzing Runtime

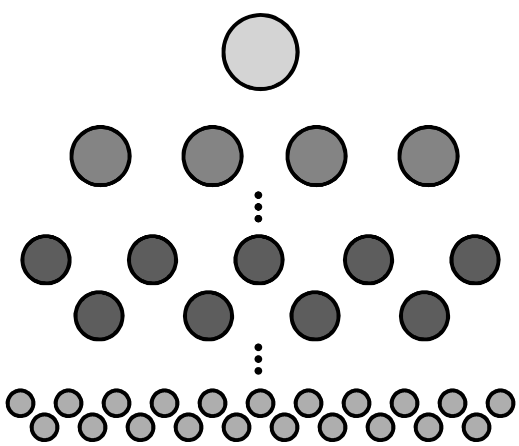
Hm. This is rather suspect …



Every pair of digits still gets multiplied together separately!

Runtime: O(n^2)

Recursion Tree Method



1 problem of size cn — **cn**

4 problems of size cn/2 — **2cn**

4^t problems of size $cn/2^t$ — **2^tcn**

$4^{log\_2(n)} = n^2$ problems of size c — **cn^2**

$\log_2 n$ levels

Runtime: O(n^2)

For now, take my word that $O(n^2 \log n)$ isn't tight.

Iteration Method

Let $T(n)$ be the runtime of `naive_recursive_multiply` on integers of length n.

Recurrence relation: $T(n) = 4T(n/2) + O(n)$

Ignore for now

$T(n) = 4 \cdot T(n/2)$

$= 4 \cdot (4 \cdot T(n/4))$ $\quad 4^2 \cdot T(n/2^2)$

$= 4 \cdot (4 \cdot (4 \cdot T(n/8)))$ $\quad 4^3 \cdot T(n/2^3)$

…

$= 2^{2t} \cdot T(n/2^t)$ $\quad 4^t \cdot T(n/2^t)$

…

$= n^2 \cdot T(1)$ $\quad 4^{\log\_2(n)} \cdot T(n/2^{\log\_2(n)})$

Runtime: $O(n^2)$

Again, take my word that $O(n^2 \log n)$ isn't tight.

Analyzing Runtime

So much work and still $O(n^2)$. This is sad :(

But wait … there's more!

Karatsuba's Algorithm (1960)

Let's break up an n-digit integer: $j = a \cdot 10^{n/2} + b$

$j \times k$

$= (a \cdot 10^{n/2} + b)(c \cdot 10^{n/2} + d)$

$= \underbrace{\boxed{(a \times c)}}_{1} \cdot 10^{n} + (\underbrace{\boxed{a \times d}}_{2} + \underbrace{\boxed{b \times c}}_{3}) \cdot 10^{n/2} + \underbrace{\boxed{(b \times d)}}_{4}$

We needed to spend 4 multiplications: one for each of 1, 2, 3, and 4.

Key insight: 2+3, 1, and 4 are part of the product (a+b)(c+d).

$(a + b)(c + d) = (ad + bc) + (ac) + (bd)$

$\boxed{(a + b)(c + d) - ac - bd} = ad + bc$

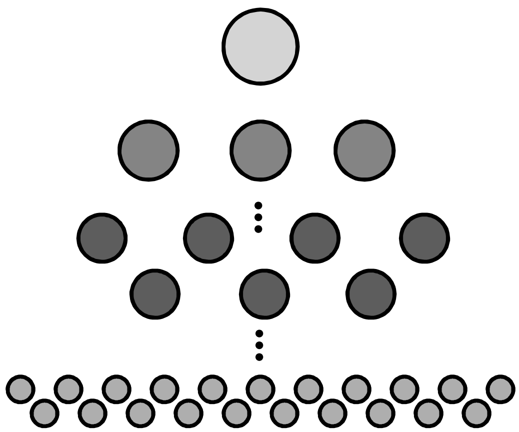
Now, we only need to spend 3 multiplications: one for each of 1 and 4, and a third one for (a+b)(c+d). From these products alone, we can infer 2 and 3.

Karatsuba's Algorithm

```
algorithm karatsuba_multiply(j, k):
  Rewrite j as a·10^(n/2) + b
  Rewrite k as c·10^(n/2) + d
  Recursively compute a·c, b·d, (a+b)(c+d)
  Let ad+bc = (a+b)(c+d)-ac-bd
  Add them up (with shifts) to get j·k
```

Runtime: $O(n^{\log\_2(3)}) = O(n^{1.585})$

Recursion Tree Method



1 problem of size cn — **cn**

3 problems of size cn/2 — **(3/2)cn**

3^t problems of size $cn/2^t$ — **$(3/2)^t$cn**

$3^{\log\_2(n)} = n^{\log\_2(3)}$ problems of size c — **$cn^{1.585}$**

$\log_2 n$ levels

Runtime: $O(n^{1.585})$

For now, take my word that $O(n^{1.585}\log n)$ isn't tight.

Integer Multiplication

$O(n^{1.585})$ runtime of Karatsuba's algorithm is an improvement over $O(n^2)$ runtime of the grade-school algorithm.

A few others outperform Karatsuba's algorithm.

- Toom-Cook algorithm (1963 and 1966) reduces 9 multiplications to 5, instead of 4 to 3, with runtime $O(n^{1.465})$.
- Schonhage-Strassen algorithm (1971) uses FFTs, with runtime O(nlog(n)loglog(n)).
- Furer's algorithm (2007) uses FFTs as well.

Fun fact: The word "algorithm" comes from Al-Khwarizmi, a Persian mathematician who wrote a book (~800 a.d.) about how to multiply Arabic numerals.

3 min break

Solving Recurrences

Solving Recurrences

We've seen three recursive algorithms.

`naive_recursive_multiply`

$T(n) = 4T(n/2) + O(n)$
$= O(n^2)$

`karatsuba_multiply`

$T(n) = 3T(n/2) + O(n)$
$= O(n^{log\_2(3)}) = O(n^{1.585})$

`mergesort`

$T(n) = 2T(n/2) + O(n)$
$= O(nlogn)$

What's the pattern???

Master Method

Suppose $T(n) = a \cdot T(n/b) + O(n^d)$.

$$T(n) = \begin{cases} O(n^d logn) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{log\_b(a)}) & \text{if } a > b^d \end{cases}$$

where

a is the number of subproblems,

b is the factor by which the input size shrinks, and

d parametrizes the runtime to create the subproblems and merge their solutions.

Master Method

We've seen three recursive algorithms.

`naive_recursive_multiply`

$T(n) = 4T(n/2) + O(n)$
$= O(n^2)$

a = 4
b = 2   a > b^d → $O(n^{\log\_b(a)})$
d = 1

Wouldn't change if d = 0

`karatsuba_multiply`

$T(n) = 3T(n/2) + O(n)$
$= O(n^{\log\_2(3)}) = O(n^{1.585})$

a = 3
b = 2   a > b^d → $O(n^{\log\_b(a)})$
d = 1

Wouldn't change if d = 0

`mergesort`

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

a = 2
b = 2   a = b^d → $O(n^d \log n)$
d = 1

Master Method

We can prove the Master Method by writing out a generic proof using a recursion tree [on the board].

Draw out the tree.

Determine the work per level.

Sum across all levels.

The three cases of the Master Method correspond to whether the recurrence is top heavy, balanced, or bottom heavy.

Solving Recurrences

So far, we've seen three approaches to solving recurrences.

- Recursion Tree Method
- Iteration Method
- Master Method

The Master Theorem

Master Method

Suppose $T(n) = a \cdot T(n/b) + O(n^c)$.

$$T(n) = \begin{cases} O(n^c) & \text{if } a < b^c \\ O(n^c \log n) & \text{if } a = b^c \\ O(n^{\log\_b(a)}) & \text{if } a > b^c \end{cases}$$

where

a is the number of subproblems,

b is the factor by which the input size shrinks, and

c parametrizes the runtime to create the subproblems and merge their solutions.

The Master Theorem

Theorem 5.1 *Let a be an integer greater than or equal to 1 and b be a real number greater than 1. Let c be a positive real number and d a nonnegative real number. Given a recurrence of the form*

$$T(n) = \begin{cases} aT(n/b) + n^c & \text{if } n > 1 \\ d & \text{if } n = 1 \end{cases}$$

then for n a power of b,

1. *if* $\log_b a < c$, $T(n) = \Theta(n^c)$,
2. *if* $\log_b a = c$, $T(n) = \Theta(n^c \log n)$,
3. *if* $\log_b a > c$, $T(n) = \Theta(n^{\log_b a})$.

The Master Theorem

Proof: In this proof, we will set $d = 1$, so that the bottom level of the tree is equally well computed by the recursive step as by the base case. It is straightforward to extend the proof for the case when $d \neq 1$.

Let's think about the recursion tree for this recurrence. There will be $\log_b n$ levels. At each level, the number of subproblems will be multiplied by a, and so the number of subproblems at level i will be a^i. Each subproblem at level i is a problem of size (n/b^i). A subproblem of size n/b^i requires $(n/b^i)^c$ additional work and since there are a^i problems on level i, the total number of units of work on level i is

$$a^i(n/b^i)^c = n^c\left(\frac{a^i}{b^{ci}}\right) = n^c\left(\frac{a}{b^c}\right)^i.$$

In general, we have that the total work done is

$$\sum_{i=0}^{\log_b n} n^c\left(\frac{a}{b^c}\right)^i = n^c\sum_{i=0}^{\log_b n}\left(\frac{a}{b^c}\right)^i$$

The Master Theorem

In general, we have that the total work done is

$$\sum_{i=0}^{\log_b n} n^c\left(\frac{a}{b^c}\right)^i = n^c\sum_{i=0}^{\log_b n}\left(\frac{a}{b^c}\right)^i$$

1. *if* $\log_b a < c$, $T(n) = \boxed{\Theta(n^c)}$,
2. *if* $\log_b a = c$, $T(n) = \Theta(n^c \log n)$,
3. *if* $\log_b a > c$, $T(n) = \Theta(n^{\log_b a})$.

In case 1, (part 1 in the statement of the theorem) this is n^c times a geometric series with a ratio of less than 1. Theorem 4.4 tells us that

$$n^c\sum_{i=0}^{\log_b n}\left(\frac{a}{b^c}\right)^i = \boxed{\Theta(n^c)}$$

The Master Theorem

In general, we have that the total work done is

$$\sum_{i=0}^{\log_b n} n^c \left(\frac{a}{b^c}\right)^i = n^c \sum_{i=0}^{\log_b n} \left(\frac{a}{b^c}\right)^i$$

1. *if* $\log_b a < c$, $T(n) = \Theta(n^c)$,
2. *if* $\log_b a = c$, $T(n) = \boxed{\Theta(n^c \log n)}$,
3. *if* $\log_b a > c$, $T(n) = \Theta(n^{\log_b a})$.

In Case 2 we have that $\frac{a}{b^c} = 1$ and so

$$n^c \sum_{i=0}^{\log_b n} \left(\frac{a}{b^c}\right)^i = n^c \sum_{i=0}^{\log_b n} 1^i = n^c(1 + \log_b n) = \boxed{\Theta(n^c \log n)}$$

The Master Theorem

In general, we have that the total work done is

$$\sum_{i=0}^{\log_b n} n^c \left(\frac{a}{b^c}\right)^i = n^c \sum_{i=0}^{\log_b n} \left(\frac{a}{b^c}\right)^i$$

1. *if* $\log_b a < c$, $T(n) = \Theta(n^c)$,
2. *if* $\log_b a = c$, $T(n) = \Theta(n^c \log n)$,
3. *if* $\log_b a > c$, $T(n) = \boxed{\Theta(n^{\log_b a})}$.

In Case 3, we have that $\frac{a}{b^c} > 1$. So in the series

$$\sum_{i=0}^{\log_b n} n^c \left(\frac{a}{b^c}\right)^i = n^c \sum_{i=0}^{\log_b n} \left(\frac{a}{b^c}\right)^i$$

the largest term is the last one, so by Theorem 4.4,the sum is $\Theta\left(n^c \left(\frac{a}{b^c}\right)^{\log_b n}\right)$. But

$$\begin{aligned} n^c \left(\frac{a}{b^c}\right)^{\log_b n} &= n^c \frac{a^{\log_b n}}{(b^c)^{\log_b n}} \\ &= n^c \frac{n^{\log_b a}}{n^{\log_b b^c}} \\ &= n^c \frac{n^{\log_b a}}{n^c} \\ &= n^{\log_b a} \end{aligned}$$

Thus the solution is $\boxed{\Theta(n^{\log_b a})}$.

Median and Selection

Beyond Master Method

The Master Method only works when the sub-problems are the same size.

Here, we'll investigate a recursive algorithm that the Master Method can't solve.

Select-k Algorithm

In the `select_k` algorithm, we will attempt to return the k^{th} smallest element of an unsorted list of values **A**.

41	23	11	5	22	4	3	14	52	20

```
select_k(A,0) => 3      select_k(A,0) => min(A)
select_k(A,4) => 14     select_k(A,⌈n/2⌉-1) => median(A)
select_k(A,9) => 52     select_k(A,n-1) => max(A)
```

A Slower Select-k Algorithm

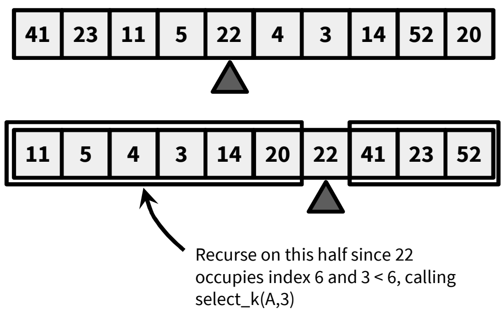
```
algorithm naive_select_k(list A, k):
  A = mergesort(A)
  return A[k]
```

Runtime: `O(nlogn)`

Select-k Algorithm

Main idea: choose a pivot, partition around it, and recurse.

Suppose we call `select_k(A,3)`.



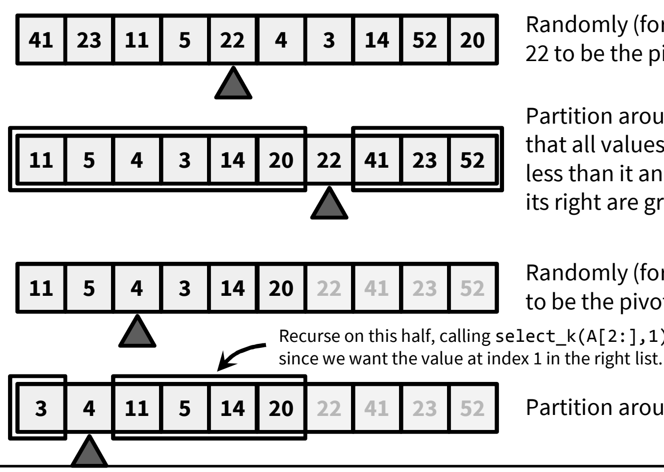
Randomly (for now) choose 22 to be the pivot.

Partition around 22, such that all values to its left are less than it and all values to its right are greater than it.

Select-k Algorithm

Main idea: choose a pivot, partition around it, and recurse.

Suppose we call `select_k(A,3)`.



Randomly (for now) choose 22 to be the pivot.

Partition around 22, such that all values to its left are less than it and all values to its right are greater than it.

Randomly (for now) choose 4 to be the pivot.

Partition around 4.

Select-k Algorithm

```
algorithm partition(list A, p):
  L, R = []
  for i = 0 to length(A)-1:
    if i == p: continue
    else if A[i] <= A[p]:
      L.append(A[i])
    else if A[i] > A[p]:
      R.append(A[i])
  return L, A[p], R
```

Runtime: $O(n)$

Select-k Algorithm

```
algorithm select_k(list A, k):
  if length(A) == 1: return A[0]
  p = random_choose_pivot(A)
  L, A[p], R = partition(A, p)
  if length(L) == k:
    return A[p]
  else if length(L) > k:
    return select_k(L, k)
  else if length(L) < k:
    return select_k(R, k-length(L)-1)
```

Runtime: $O(n^2)$

We'll talk about why this is the case later.

Select-k Algorithm

Question 1 How do we prove this algorithm always returns the k^{th} smallest element of **A**?

Question 2 How efficiently does this algorithm return the k^{th} smallest element?

Proving Correctness

Informally (explain it to your co-worker) ...

(Ignore the fact that there's no error-checking so `select_k(A,10)` where `length(A) <= 10` breaks the algorithm.)

Inductive hypothesis: At the return of each recursive call of size < n, `select_k(A,k)` returns the k^{th} smallest element of **A**.

When length(A) == 1, then returning the only element is correct.

Suppose the inductive hypothesis holds for n. We want to show that it holds for n + 1. There are three cases:

(1) length(L) = k: A[p] is the correct thing to return.
(2) length(L) > k: the k^{th} smallest element of L is the correct thing to return.
(3) length(L) < k: the (k - length(L) - 1)st smallest element is the correct thing to
to
return.

By induction, select_k is correct.

Analyzing Runtime

Recall `p = random_choose_pivot(A)`.
Why is this algorithm $O(n^2)$?

- Suppose we called `select_k(A,0)`, i.e. we want the min element, and we get unlucky with our selected pivot.

We can fix this by choosing our pivot more carefully.

Select-k Algorithm

```
algorithm smartly_choose_pivot(list A):
  groups = split A into m=⌈length(A)/5⌉
           groups, of size ≤ 5 each
  candidate_pivots = []
  for i = 0 to m-1:
    p_i = median(groups[i])  # O(1)
    candidate_pivots.append(p_i)
  A[p] = select_k(candidate_pivots, m/2)
  return index_of(A[p])
```

Select-k Algorithm

```
algorithm select_k(list A, k):
  if length(A) ≤ 100:
    return naive_select_k(A, k)
  p = smartly_choose_pivot(A)
  L, A[p], R = partition(A, p)
  if length(L) == k:
    return A[p]
  else if length(L) > k:
    return select_k(L, k)
  else if length(L) < k:
    return select_k(R, k-length(L)-1)
```

Runtime: `O(n)` ← But why? This is not obvious at all...

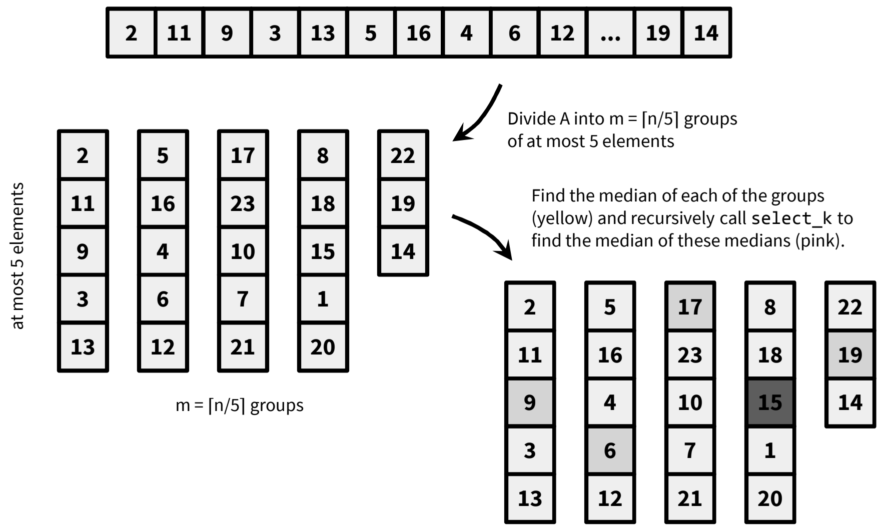
Analyzing Runtime

Instead of `p = random_choose_pivot(A)`, now we have `p = smartly_choose_pivot(A)`.

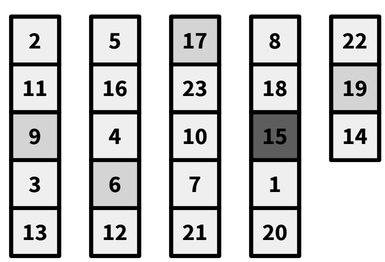
Why is this algorithm **`O(n)`**?

Main idea: each of the arrays L and R are pretty balanced. Thus, while the median of medians might not be the actual median, it's pretty close.

Analyzing Runtime



Analyzing Runtime



Clearly the median of medians (15) is not necessarily the actual median (12), but we claim that it's guaranteed to be pretty close.

Analyzing Runtime

2	5	8	10	22
3	4	1	7	19
9	6	15	17	14
11	16	18	23	
13	12	20	21	

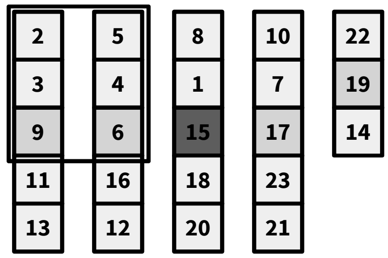
To see why, let's partition elements within each of the groups around the group's median, and partition the groups around the group with the median of medians.

Analyzing Runtime

2	5	8	10	22
3	4	1	7	19
9	6	15	17	14
11	16	18	23	
13	12	20	21	

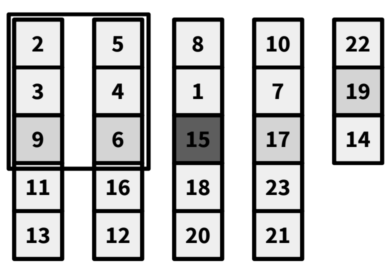
How many elements are smaller than the median of medians?

Analyzing Runtime



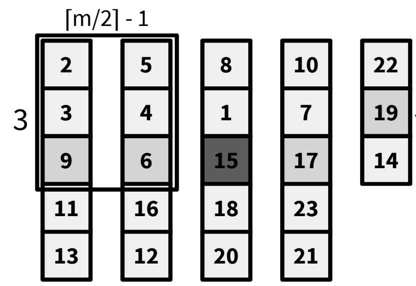
At least these guys (2, 3, 4, 5, 6, 9): everything above and to the left. There might be more (1, 7, 8, 11, 12, 13, 14), but we are guaranteed that *at least* these guys will be smaller.

Analyzing Runtime



How many are there?

Analyzing Runtime



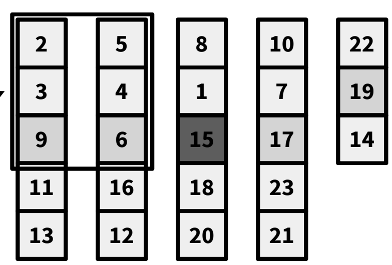
This is the “leftovers group”; if its contents were [12, 14, 16], then it would not have 3 elements less than the median of medians; thus the -1 below.

At least 3·(⌈m/2⌉ - 1 - 1)

One of groups could have been the leftovers group

Analyzing Runtime

Everything besides this



How many elements are larger than the median of medians?
At most n - 1 - 3·(⌈m/2⌉ - 1 - 1) ≤ 7n/10 + 5.

Analyzing Runtime

We just showed that …

`smartly_choose_pivot` will choose a pivot greater than at least 3·(⌈m/2⌉ - 2) elements.

$$3\cdot(\lceil m/2 \rceil - 2) \le |L| \le 7n/10 + 5$$

$$3\cdot(\lceil m/2 \rceil - 2) \le |R| \le 7n/10 + 5$$

`smartly_choose_pivot` will choose a pivot less than at most 7n/10 + 5 elements.

Analyzing Runtime

We can just as easily show the inverse.

$$3\cdot(\lceil m/2 \rceil - 2) \le |L| \le 7n/10 + 5$$

$$3\cdot(\lceil m/2 \rceil - 2) \le |R| \le 7n/10 + 5$$

Analyzing Runtime

What's the greatest number of elements that can be smaller than p?

- `random_choose_pivot` might choose the largest element, so n-1.
- `smartly_choose_pivot` will choose an element greater than at most 7n/10 + 5 elements.

What's the greatest number of elements that can be larger than p?

- `random_choose_pivot` might choose the smallest element, so n-1.
- `smartly_choose_pivot` will choose an element smaller than at most 7n/10 + 5 elements.

Analyzing Runtime

Recurrence relation: $T(n) \le c \cdot n + T(\lceil n/5 \rceil) + T(\lceil 7n/10 + 5 \rceil)$.

Partitioning, computing n/5 medians

Computing the median of n/5 medians.

Recursing on L or R.

But what if n = 4?

- We introduce a "fat base case" where $T(n) = \Theta(1) \le c$ for $n \le 100$.

Recall that the Master Method only works when the sub-problems are the same size.

To prove this recurrence relation yields a runtime of **`O(n)`**, we will employ substitution method.

Analyzing Runtime

Theorem: $T(n) = O(n)$

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

We proceed by induction. As a base case, if $1 \leq n \leq 100$, then $T(n) \leq c \leq kn$ will be true as long as we pick $k \geq c$.

For the inductive step, assume for some $n \geq 100$ that the claim holds for all $1 \leq n' < n$. Note that $1 \leq \lceil n/5 \rceil, \lceil 7n/10 + 5 \rceil < n$. Then:

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

If we pick $k = 50c$, then $7k + cn - kn/10 \leq 0$ and $T(n) \leq kn$ holds, completing the induction. ■

Substitution Method

To use substitution method, proceed as follows:

- Make a guess of the form of your answer (e.g. kn)
- Proceed by induction to prove the bound holds, noting what constraints arise on your undetermined constants (e.g. k).
- If you induction succeeds, you will have values for your undetermined constants.
- If the induction fails, then it doesn't necessarily imply that your guess fails to bound the recurrence.