Divide and Conquer II
Proof of Correctness
Solving Recurrences

Outline for Today

Divide and Conquer II

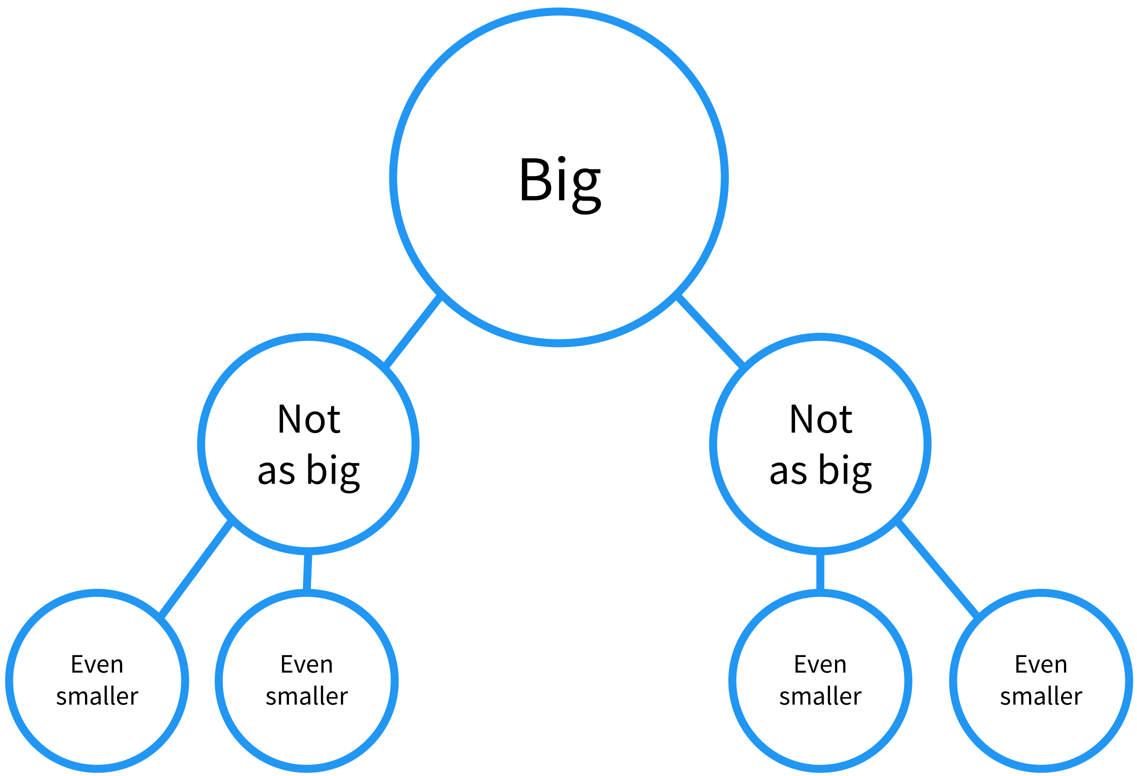
- [Example] Integer multiplication (revisited)
- [Example] Find the Number (revisited)
- [Example] Mergesort
- Solving recurrences
  - Recursion Tree method
  - Iteration method
  - Master method
- [Example] Median and selection
  - Substitution method

Divide and Conquer

Divide and Conquer

An algorithm design paradigm
Divide: break current problem into smaller sub-problems.

Conquer: solve the smaller sub-problems recursively and collect the results to solve the current problem.



Integer Multiplication

- **Original large problem:** multiply two n-digit numbers
- **What are the subproblems?**

$$1234 \times 5678$$

$$= (12 \times 100 + 34) \times (56 \times 100 + 78)$$

$$= (\underbrace{12 \times 56}_{1})100^2 + (\underbrace{12 \times 78}_{2} + \underbrace{34 \times 56}_{3})100 + (\underbrace{34 \times 78}_{4})$$

One 4-digit problem → *Four 2-digit sub-problems*

Integer Multiplication

- **Original large problem:** multiply two n-digit numbers
- **What are the subproblems?** More generally:

$$[x_1x_2 \ldots x_{n-1}x_n] \times [y_1y_2 \ldots y_{n-1}y_n]$$

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

$$= (\underset{1}{\mathbf{a} \times \mathbf{c}})10^n + (\underset{2}{\mathbf{a} \times \mathbf{d}} + \underset{3}{\mathbf{b} \times \mathbf{c}})10^{n/2} + (\underset{4}{\mathbf{b} \times \mathbf{d}})$$

One n-digit problem → *Four n/2-digit sub-problems*

Pseudo-Code

```
algorithm multiply(x, y, n):
  if n == 1: return x·y

  Rewrite x as a·10^(n/2) + b
  Rewrite y as c·10^(n/2) + d

  set ac = multiply(a, c, n/2)
  set ad = multiply(a, d, n/2)
  set bc = multiply(b, c, n/2)
  set bd = multiply(b, d, n/2)

  return ac·10^n + (ad+bc)·10^(n/2) + bd
```

x, y are n-digit numbers

Base case: when x and y are 1-digit, we can directly return their product, e.g., by referencing the multiplication table

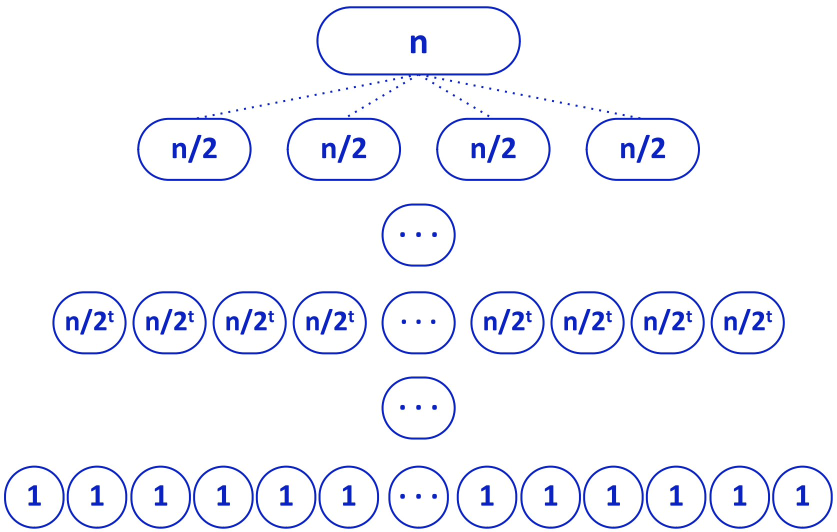
a, b, c, d are n/2-digit numbers

Call the algorithm recursively to get answers of the sub-problems

Add-up to get final answer

How Efficient is the Algorithm?

Question: How many **basic operations** the algorithm needs to do in the **worst case**?



Level 0: 1 problem of size n — $1 \times n = n$

Level 1: 4^1 problems of size n/2 — $4 \times n/2 = 2n$

Level t: 4^t problems of size $n/2^t$ — $4^t \times n/2^t = 2^t n$

Level $\log_2 n$: $\underline{\ n^2\ }$ problems of size 1 — $4^{\log_2 n} \times 1 = 2^{\log_2 n} \times n$

$$\left(1 + 2 + 2^2 + 2^3 + \cdots + 2^{\log_2 n}\right) n = \boldsymbol{2n^2 - n} = O(n^2)$$

Computational Complexity: $O(n^2)$

Basic Operations

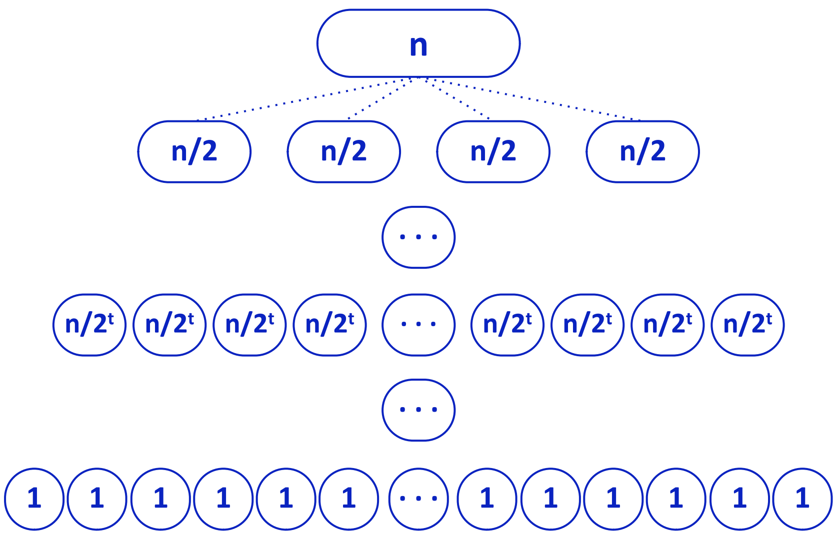
Why multiplying two numbers can not be treated as doing one operation?

The computational complexity measures the number of **Basic Operations** needed to finish calculation.

What is basic operation? An operation that can be done by 1 step of register operation (recall what you have learned in **computer architecture** and **assembly language**)

Inlcuding:
Additor: +, -
Compairor: <, >, ==
Shifter: <<, >>
Value assignment

How Efficient is the Algorithm?



Level 0: 1 problem of size n $\quad 1 \times \mathbf{c} \cdot n = cn$

Level 1: 4^1 problems of size n/2 $\quad 4 \times \mathbf{c} \cdot n/2 = 2cn$

Level t: 4^t problems of size $n/2^t$ $\quad 4^t \times \mathbf{c} \cdot n/2^t = 2^t cn$

Level $\log_2 n$: $\underline{n^2}$ problems of size 1 $\quad 4^{\log_2 n} \times \mathbf{c} \cdot 1 = 2^{\log_2 n} cn$

$$\left(1 + 2 + 2^2 + 2^3 + \cdots + 2^{\log_2 n}\right) cn = \mathbf{2cn^2 - cn} = O(n^2)$$

Computational Complexity: $O(n^2)$

Reducing to Three Sub-Problems

These *three* subproblems give us everything we need to compute our desired quantities:

1 **ac**

2 **bd**

3 **(a+b)(c+d)**

(a+b) and (c+d) are both going to be n/2-digit numbers!

⇩

This means we still have half-sized subproblems!

Compute our final result by combining these three subproblems:

$$(ac)10^n + (ad + bc)10^{n/2} + (bd)$$

1   3 - 1 - 2   2

Pseudo-Code

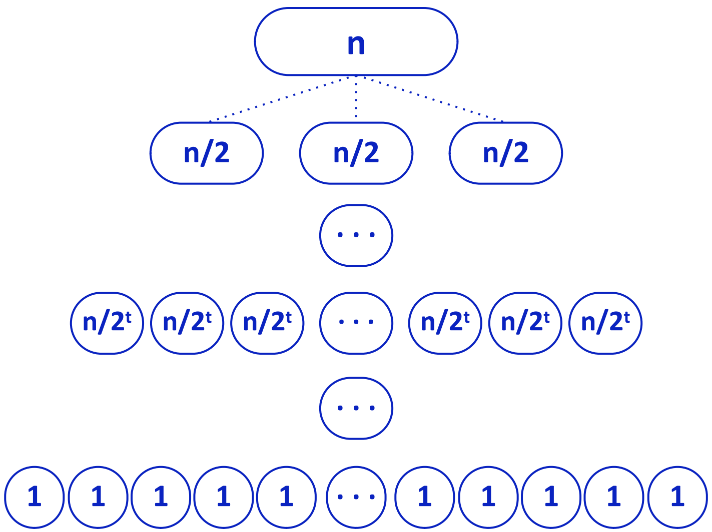
```
algorithm karatsuba_multiply(x, y, n):
  if n == 1: return x·y

  Rewrite x as a·10^(n/2) + b
  Rewrite y as c·10^(n/2) + d

  set ac   = multiply(a, c, n/2)
  set ad   = multiply(a, d, n/2)
  set abcd = multiply(a+b, c+d, n/2)

  return ac·10^n + (abcd-ac-bd)·10^(n/2) + bd
```

Only 3 n/2-digit sub-problems

Add-up to get final answer

How Efficient is the Algorithm?

For the new algorithm, we replace branching factor of 4 to 3



Level 0: 3^0 problem of size n — $1 \times n = n$

Level 1: 3^1 problems of size n/2 — $3 \times n/2 = (3/2)n$

Level t: 3^t problems of size $n/2^t$ — $3^t \times n/2^t = (3/2)^t n$

Level $\log_2 n$: ___$n^{1.6}$___ problems of size 1

$3^{\log_2 n} \times 1 = (3/2)^{\log_2 n} \times n$

$$\left(1 + \frac{3}{2} + (\frac{3}{2})^2 + (\frac{3}{2})^3 + \cdots + (\frac{3}{2})^{\log_2 n}\right) n = \boldsymbol{3n^{\log_2 3} - 2n}$$

$$= \cancel{3}\boldsymbol{n^{1.6}} - \cancel{2}\boldsymbol{n} = O(n^{1.6})$$

Computational Complexity: $O(n^{1.6})$

Binary-Search is also D-n-C! (really?!)

Question: Given a sorted array **`A[0:n-1]`**, locate number **x** in the array.

n numbers in total, i.e., length(A)==n

1	3	4	5	7	8	9	11	14	16	23	28	31	34	37	41	50

```
algorithm binary_search(A, x):
  set L = 0, R = n-1
  while L <= R:
      set i = L + ⌊(R-L)/2⌋
      if A[i] == x:  //one basic operation
          return i;
      else if A[i] < x:
          set L = i + 1;
      else if A[i] > x:
          set R = i - 1;
  return -1;
```

A Recursive Version of the Algorithm

Question: Given a sorted array **`A[0:n-1]`**, locate number **`x`** in the array.

n numbers in total, i.e., length(A)==n

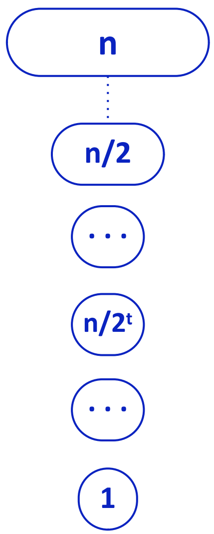
1	3	4	5	7	8	9	11	14	16	23	28	31	34	37	41	50

```
call binary_search(A, x, 0, n-1)
```

```
algorithm binary_search(A, x, L, R):
  if L > R:                    //one basic operation
     return -1;
  set i = L + ⌊(R-L)/2⌋        //four basic operations
  if A[i] == x:                //one basic operation
     return i;
  if A[i] < x:                 //one basic operation
     return binary_search(A, x, i+1, R);
  if A[i] > x:                 //one basic operation
     return binary_search(A, x, L, i-1);
```

How Efficient is the Algorithm?

Question: How many **basic operations** the algorithm needs to do in the **worst case**?



Level 0: 1 problem, c operation $1 \times c = c$

Level 1: 1 problem, c operation $1 \times c = c$

Level t: 1 problem, c operation $1 \times c = c$

Level $\log_2 n$: 1 problem, c operation $1 \times c = c$

- Why $\log_2 n$ levels? Because if $n/2^t=1$, we have $t=\log_2 n$
  - i.e., you need to cut n in half $\log_2 n$ times to get to size 1
- In each (sub-)problem, the number of basic operations is a constant c

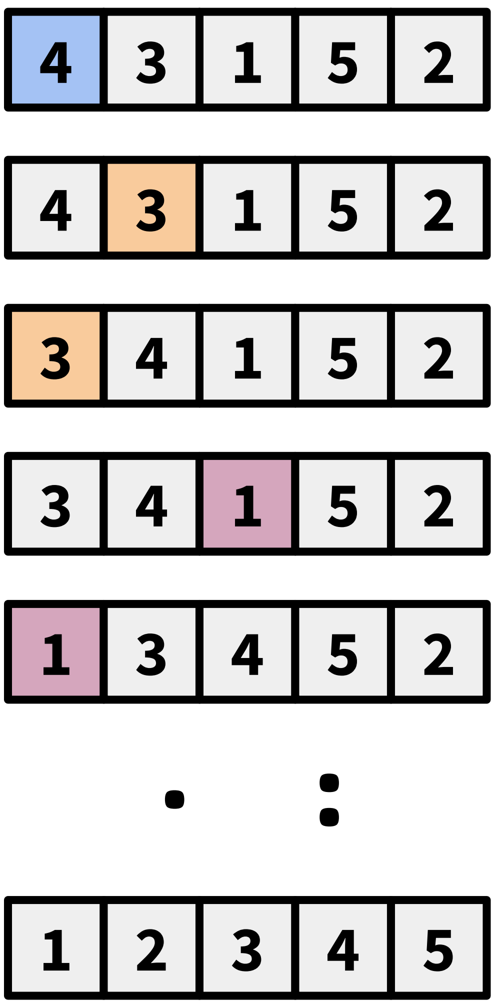
$$c + c + \cdots + c = \boldsymbol{c} \log_2 \boldsymbol{n} = O(\log n)$$

Computational Complexity: `O(log(n))`

5-Minute Break

Insertion Sort

Insertion sort



Let's sort an unsorted list of numbers **`A`**. The sublist **`A[0:0]`** is trivially sorted.

Look at the second element, **`A[1]`**.

Insert the element into a new position such that the sublist **`A[0:1]`** is sorted.

Now look at the third element, **`A[2]`**.

Insert it such that the sublist **`A[0:2]`** is sorted.

The entire array **`A[0:4]`** is sorted.

Insertion sort

```
algorithm insertion_sort(list A):
  for i = 0 to length(A)-1:
    let cur_value = A[i]
    let j = i - 1
    while j ≥ 0 and A[j] > cur_value:
      A[j+1] = A[j]
      j = j - 1
    A[j+1] = cur_value
```

Insertion sort

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

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

Proving Correctness

Algorithms often initialize, modify, or delete new data.

To prove an algorithm is correct, you have to prove it's correct for any input size **n**.

However, input size **n** can be infinite, impossible to prove for each and all possible input size **n**.

Use Mathematical Induction!

Mathematical Induction

Mathematical Induction:

We have a claim C(n), we verify that C(0) is True, we then suppose that when n=k, C(k) is true, and prove that when n=k+1, C(k+1) will be true; then we can claim that C(n) is true for all possible n.

Deciding the Invariant for Mathematical Induction:

The key to construct a valid proof using mathematical induction is to find a good invariant, i.e., a property that does not change during the algorithm.

This unchanging property is called an **invariant**.

Invariant for Insertion Sort

```
algorithm insertion_sort(list A):
  for i = 0 to length(A)-1:
    let cur_value = A[i]
    let j = i - 1
    while j ≥ 0 and A[j] > cur_value:
      A[j+1] = A[j]
      j = j - 1
    A[j+1] = cur_value
```

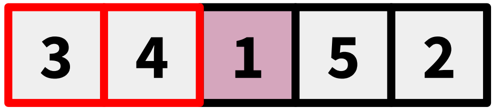
Algorithm takes care of each element of the array one by one, from the first to the last

Invariant for Insertion Sort

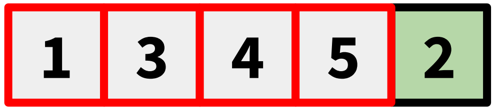
Invariant of the outer for-loop: At the start of iteration i of the outer for-loop, the first i elements of the list are (always) sorted.

Sanity checks:

At the start of the third iteration (i.e. the iteration when i = 2), the first 2 elements of the list are sorted. True.



At the start of the fifth iteration (i.e. the iteration when i = 4), the first 4 lements of the list are sorted. True.



Proving Correctness

Less formally (explain it to your co-worker) ...

At the start of the first iteration, the first element of the array is sorted.

By construction, the i^{th} iteration puts element **`A[i]`** in the right place.

At the start of the final iteration (i = length(A), aka the end of the algorithm), the first length(A) elements are sorted.

More formally (rigorously) …

Outer invariant (for-loop): At the start of iteration i of the outer for-loop, the first i elements of the list are sorted.

Inner invariant (while-loop): At the start of iteration j of the inner while-loop, **`A[0:j,j+2:i]`** contains the same elements as the original sublist **`A[0:i-1]`**, still sorted, such that all of the values in the right sublist **`A[j+2:i]`** are greater than **`cur_value`**.

Proving Correctness

More formally (rigorously, prove using induction twice) ...

Lemma: If **`A[0:i-1]`** has already been sorted at the start of iteration i-1 of the loop, then **`A[0:i]`** will be sorted at the start of iteration i of the loop.

Proof:

To prove this statement, we examine the inner loop invariant, by induction.

- The invariant holds at the start of the iteration j = i-1 of the inner while-loop. To see why, notice that **`A[0:j,j+2:i]`** describes the same sublist as **`A[0:i-1,i+1:i]`**(since we initialized j to i-1), which trivially contains the same elements as the original sublist **`A[0:i-1]`**, still sorted, since the right sublist **`A[i+1:i]`** is empty.
- Furthermore, since the right sublist is empty, all of its values are all vacuously greater than **`cur_value`**.

```
algorithm insertion_sort(list A):
  for i = 0 to length(A)-1:
    let cur_value = A[i]
    let j = i - 1
    while j ≥ 0 and A[j] > cur_value:
      A[j+1] = A[j]
      j = j - 1
    A[j+1] = cur_value
```

```
algorithm insertion_sort(list A):
  for i = 0 to length(A)-1:
    let cur_value = A[i]
    let j = i - 1
    while j ≥ 0 and A[j] > cur_value:
      A[j+1] = A[j]
      j = j - 1
    A[j+1] = cur_value
```

Proof of lemma, cont.:

Now, we will prove the inductive step. Suppose that the invariant holds at the start of an arbitrary iteration j = y (inductive hypothesis). We prove that it still holds at the start of iteration j = y-1. There are two cases of the while-loop condition to consider when j=y-1:

- The condition returns True.

  First, **`A[j]`** is copied to **`A[j+1]`**, as a result, **`A[j]`** = **`A[j+1]`**, i.e., **`A[y-1]`** = **`A[y]`** (note that current j=y-1). By inductive hypothesis, we have **`A[0:y,y+2:i]`** satisfies the invariant. As a result, **`A[0:y-1,y+1:i]`** also satisfies the invariant. Since j=y-1, so **`A[0:j,j+2:i]`** now satisfies the invariant for j = y-1, maintaining the invariant for the next iteration.

- The condition returns False.

  The loop terminates. Since either (1) j is -1 or (2) **`cur_value`** is greater than **`A[j]`**, then **`A[0:j],cur_value`** must be sorted (recall the invariant guarantees that **`A[0:j]`** is sorted). Furthermore, since all of the values in the right sublist **`A[j+2:i]`** are sorted and greater than **`cur_value`**, then **`A[0:j],cur_value,A[j+2:i]`** must be sorted. Thus, at the termination of the loop, **`A[0:i]`** (the first i+1 elements) is sorted.

Proving Correctness

Theorem: Insertion sort sorts the input list.

Proof:

At the start of the first iteration of the outer for-loop, **`A[0:-1]`** (an empty sublist) is trivially sorted.

By our lemma, if **`A[0:x-1]`** is sorted at the start of iteration i = x of the loop, then **`A[0:x]`** will be sorted at the start of iteration i = x+1 of the loop.

The loop terminates at the start of iteration **`length(A)`**, which implies that **`A[0:length(A)-1]`** is sorted when the loop ends, which proves the theorem.

Proving Correctness

Both the lemma and theorem follow a consistent format:

Initialization: The loop invariant starts out as true.

Maintenance: If the loop invariant is true at step i, then it's true at step i+1.

Termination: If the loop invariant is true at the end of the algorithm, this tells you something about what you're trying to prove.

Insertion sort

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

```
algorithm insertion_sort(list A):
  for i = 1 to length(A):
    let cur_value = A[i]
    let j = i - 1
    while j > 0 and A[j] > cur_value:
      A[j+1] = A[j]
      j = j - 1
    A[j+1] = cur_value
```

O(n) work per iteration

O(n) iterations

Total work: $O(n^2)$

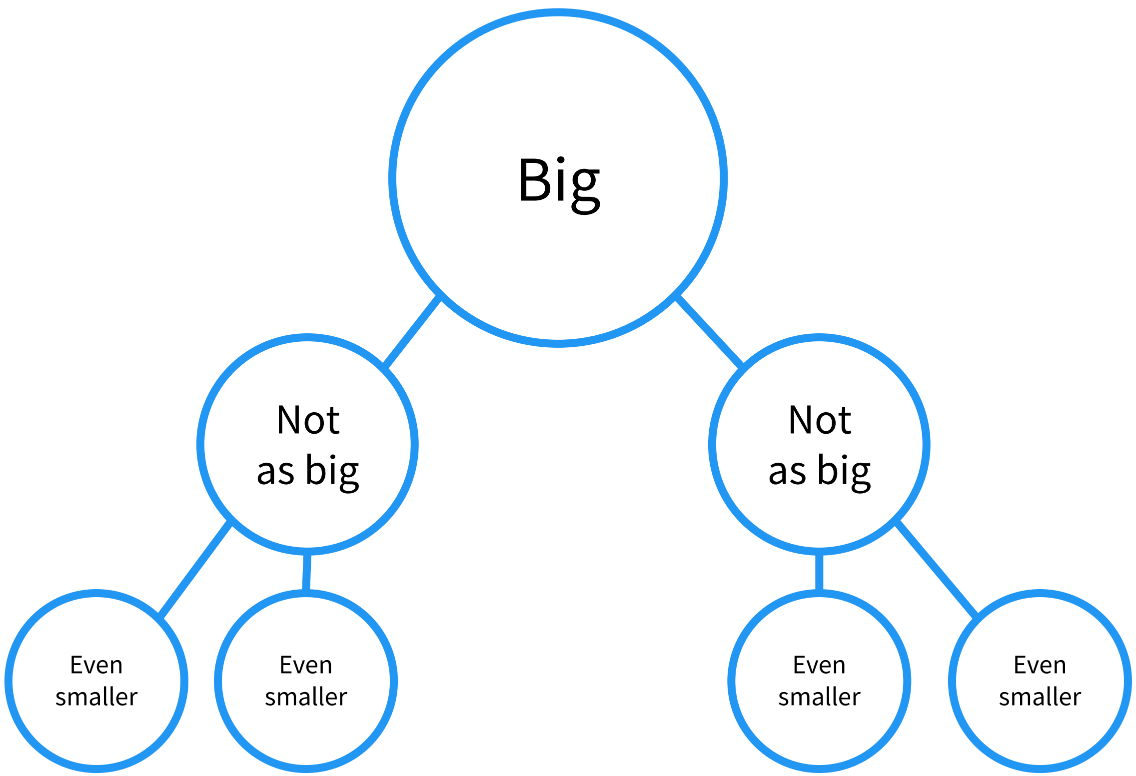
5-Minute Break

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?

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)
  )
```

$\Theta(n)$ operations to slip A into left and right lists

$\Theta(n)$ operations to merge two lists

$T(\lceil n/2 \rceil)$ operations to sort left list

$T(\lfloor n/2 \rfloor)$ operations to sort left list

Let T(n) be the number of basic operations of mergesort(A) when input size is n:

$$T(0) = T(1) = \Theta(1)$$

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

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?

$$\cancel{T(0) = \Theta(1)}$$

$$T(1) = \Theta(1) = c_1$$

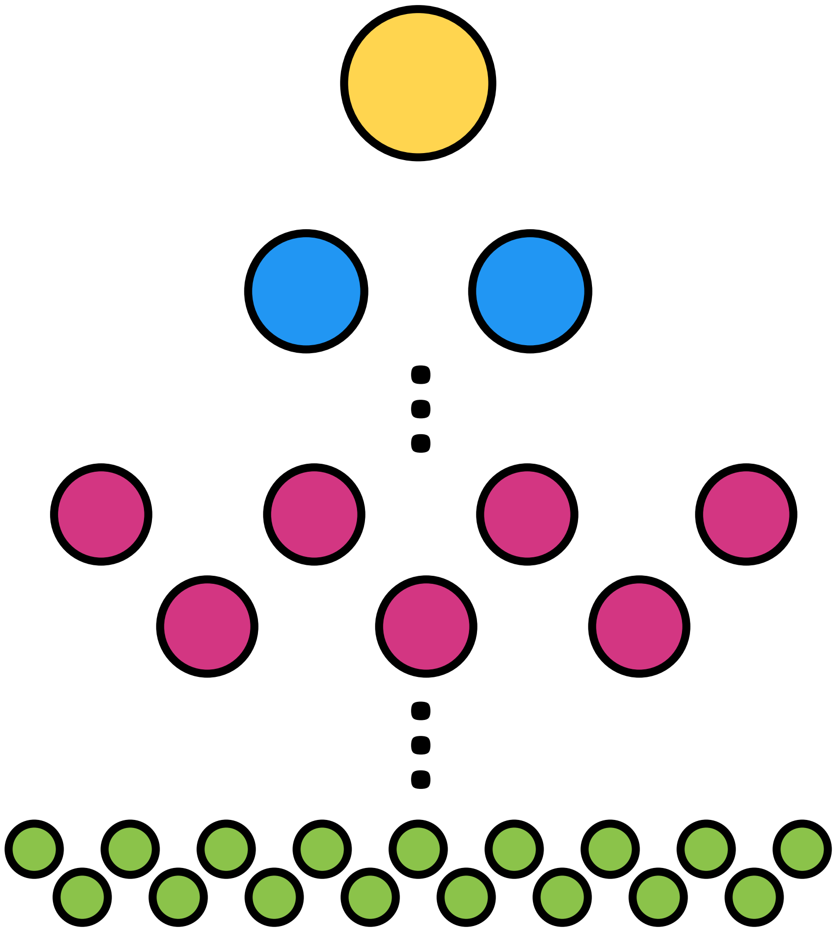
$$\begin{aligned} T(n) &= T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) \\ &= 2T(n/2) + c_2 n \end{aligned}$$

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

$$T(1) \leq c$$

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

Recursion Tree Method



1 problem of size cn — **cn**

2 problems of size cn/2 — **cn**

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

$2^{\log_2 n} = n$ problems of size c — **cn**

$\log_2 n$ levels

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

Iteration Method

Recall, our recurrence relation:

$$T(1) \leq c$$

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

$$\begin{aligned} T(n) &\leq 2 \cdot T(n/2) + cn \\ &\leq 2 \cdot (2T(n/4) + cn/2) + cn \\ &= 4 \cdot T(n/4) + 2cn \\ &\leq 4 \cdot (2T(n/8) + cn/4) + 2cn \\ &= 8 \cdot T(n/8) + 3cn \\ &\dots \\ &\leq 2^k T(n/2^k) + kcn \end{aligned}$$

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

So $k = \log_2 n$

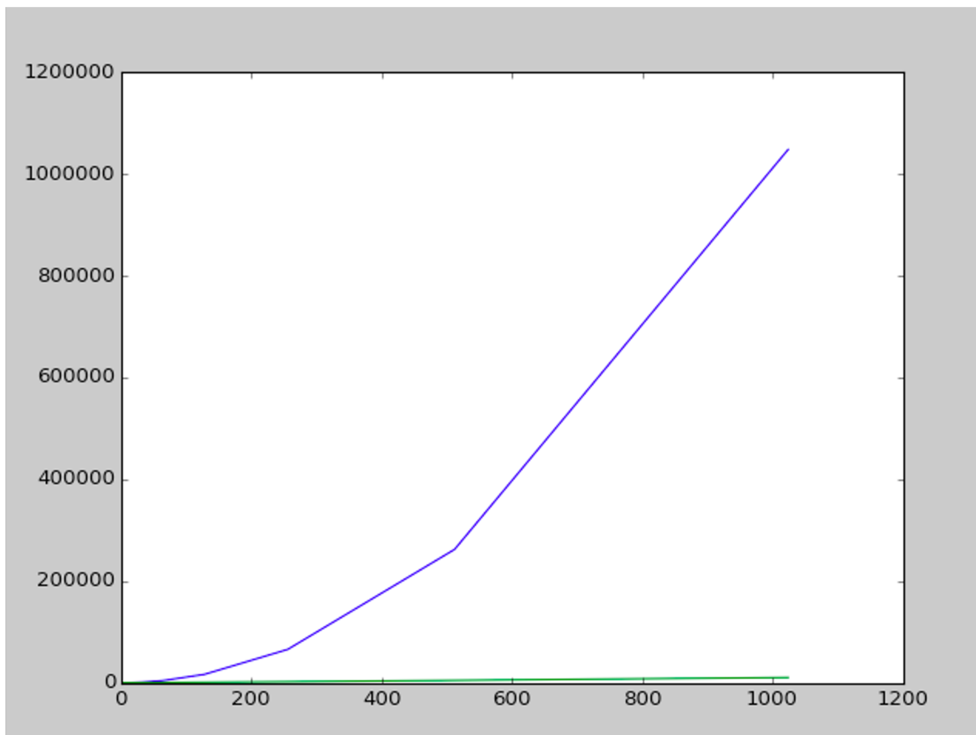
$$\begin{aligned} T(n) &\leq 2^k T(n/2^k) + kcn \\ &= 2^{\log_2 n} T(n/2^{\log_2 n}) + cn\log_2 n \\ &= nT(1) + cn\log_2 n \\ &\leq cn + cn\log_2 n \\ &= O(n\log n) \end{aligned}$$

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!!



Solving Recurrences

Solving Recurrences

We've seen four recursive algorithms, recursion relations:

`naive_recursive_multiply`

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

`karatsuba_multiply`

$T(n) = 3T(n/2) + O(n)$
$= O(n^{\log_2 3}) = O(n^{1.6})$

What's the pattern???

`mergesort`

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

`binary_search`

$T(n) = T(n/2) + O(1)$
$= O(\log n)$

Master Method

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

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

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 four recursive algorithms.

`naive_recursive_multiply`

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

$a = 4$
$b = 2$
$d = 1$

$a > b^d \rightarrow O(n^{\log_b a})$

Wouldn't change
if d = 0

`karatsuba_multiply`

$T(n) = 3T(n/2) + O(n)$
$= O(n^{\log_2 3}) = O(n^{1.6})$

$a = 3$
$b = 2$
$d = 1$

$a > b^d \rightarrow O(n^{\log_b a})$

Wouldn't change
if d = 0

`mergesort`

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

$a = 2$
$b = 2$
$d = 1$

$a = b^d \rightarrow O(n^d \log n)$

`binary_search`

$T(n) = T(n/2) + O(1)$
$= O(\log n)$

$a = 1$
$b = 2$
$d = 0$

$a = b^d \rightarrow O(n^d \log n)$

Master Method

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

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.

General proof of Master Method is one of our homework.

Solving Recurrences

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

Recursion Tree Method

Iteration Method

Master Method

5-Minute Break

Median and Selection

Beyond Master Method

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

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

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**
8	7	3	2	6	1	0	4	9	5

```
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

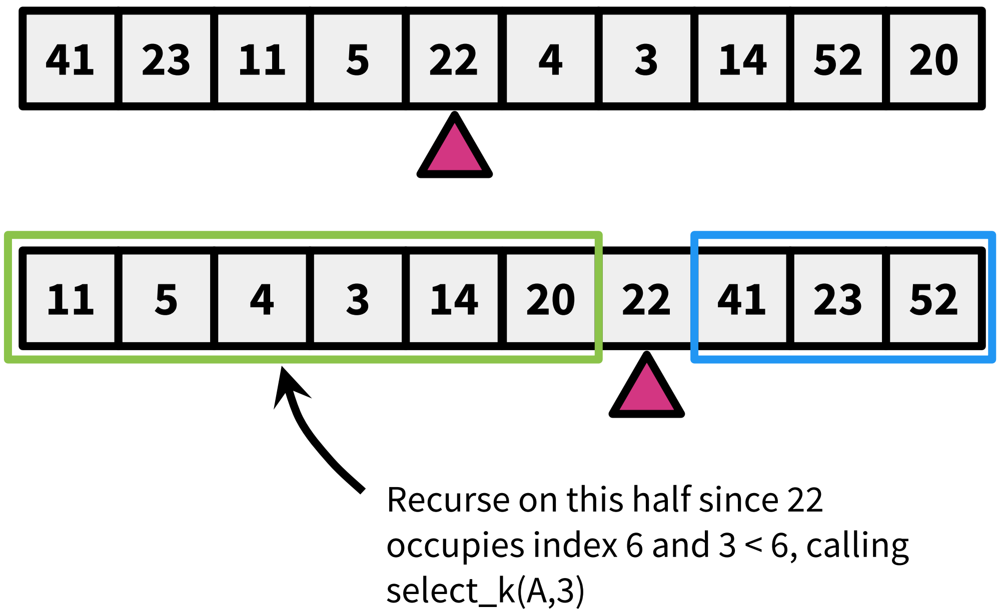
```
algorithm naive_select_k(list A, k):
  A = mergesort(A)
  return A[k]
```

Runtime: `O(nlogn)`

Select-k Algorithm

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

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



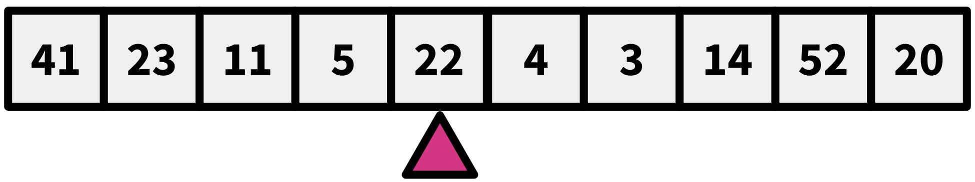
Randomly (for now) choose 22 to be the pivot.

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

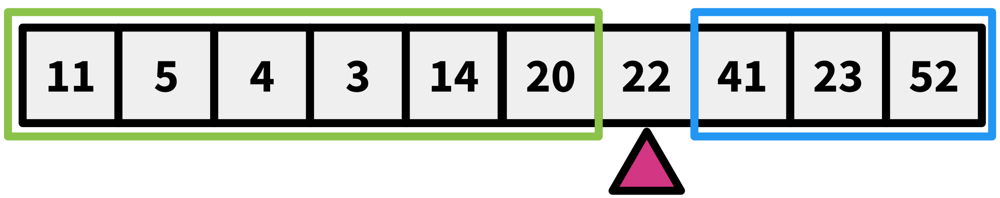
Select-k Algorithm

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

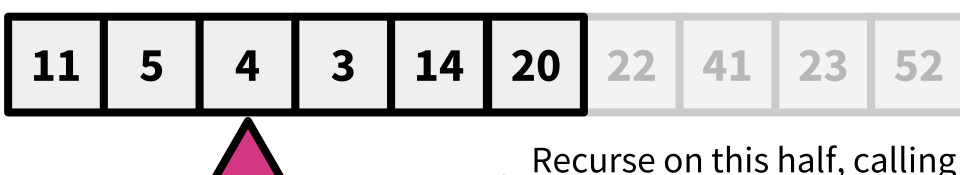
Suppose we call `select_k(A,3)`.



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

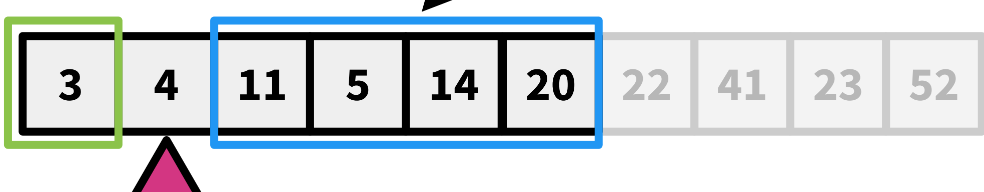


Partition around 22, such that all values to its left are less than it and all values to its right are greater than it.



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

Recurse on this half, calling `select_k(A[2:],1)` since we want the value at index 1 in the right list.



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 list size ≤ n, `select_k(A,k)` returns the k^{th} smallest element of **A**.

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

Suppose the inductive hypothesis holds for size ≤ 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 $(k - \text{length}(L) - 1)^{st}$ smallest element is the correct thing to return.

By induction, select_k is correct.

Analyzing Runtime

Recall `p = random_choose_pivot(A)`.

Why is this algorithm $O(n^2)$?

Computation complexity **measures the worst case**.

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

Partition into m=n/5 groups

For each group, select its median

Select the median of medians

```
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

If length is small, naïve select k directly

Otherwise, choose pivot smartly and partition

Recursive calls

```
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)`**?

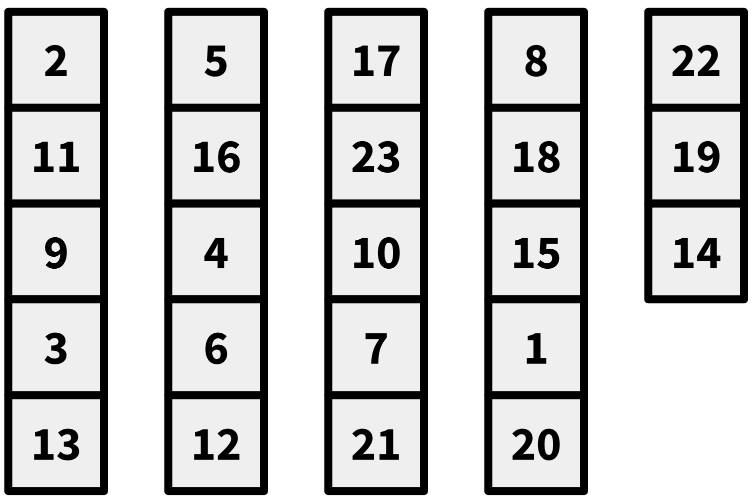
Main idea: each of the arrays L and R are pretty balanced. Thus, while the **median of medians** might not be the actual median, it's pretty close.

Analyzing Runtime

2	11	9	3	13	5	16	4	6	12	…	19	14

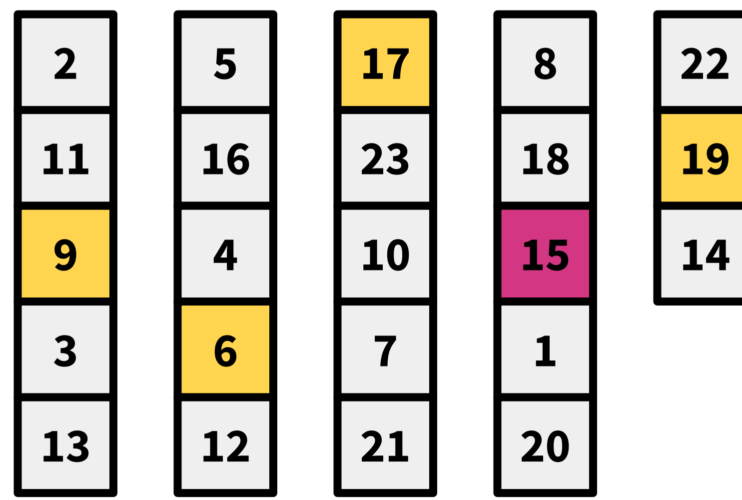
Divide A into m = $\lceil n/5 \rceil$ groups of at most 5 elements

at most 5 elements

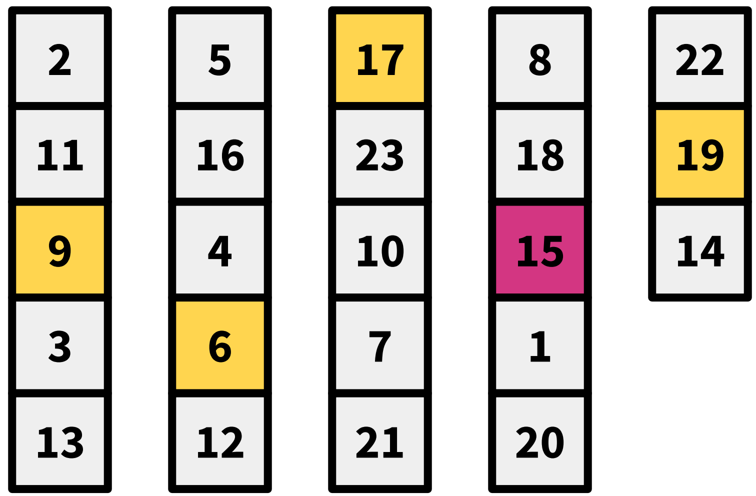


m = $\lceil n/5 \rceil$ groups

Find the median of each of the groups (yellow) and recursively call `select_k` to find the median of these medians (pink).

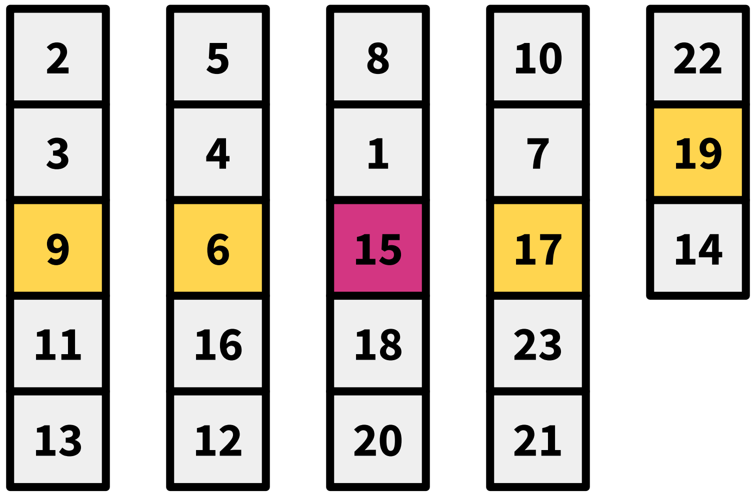


Analyzing Runtime



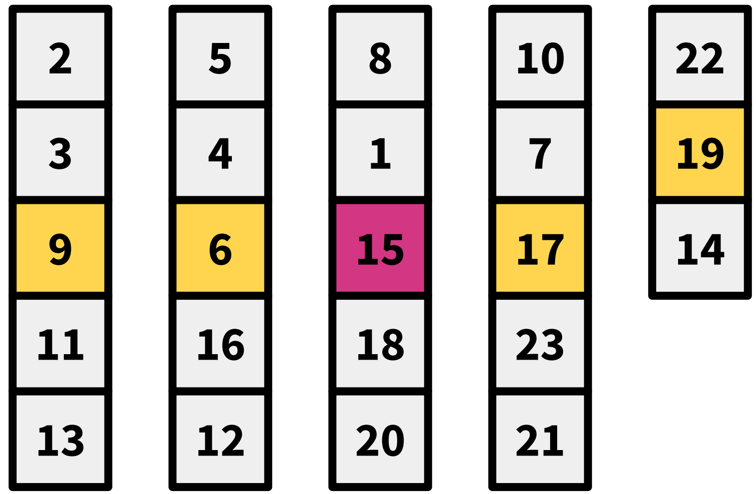
Clearly the median of medians (15) is not necessarily the actual median (12), but we claim that it's guaranteed to be pretty close.

Analyzing Runtime



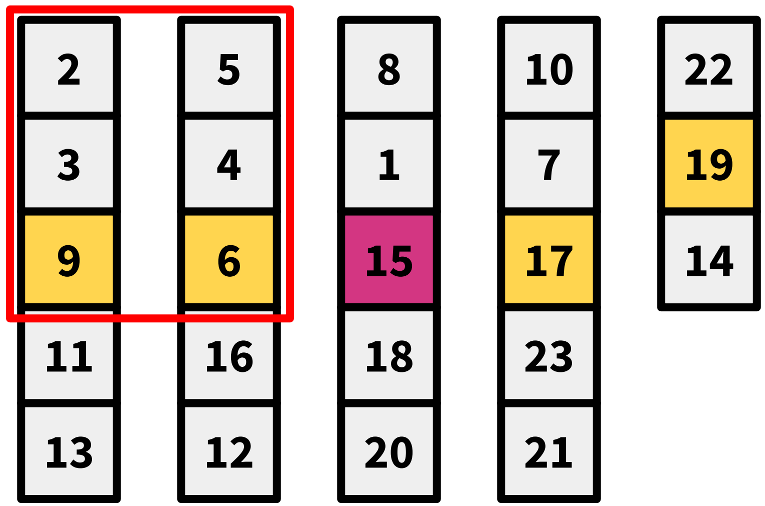
To see why, let's partition elements within each of the groups around the group's median, and partition the groups around the group with the median of medians.

Analyzing Runtime



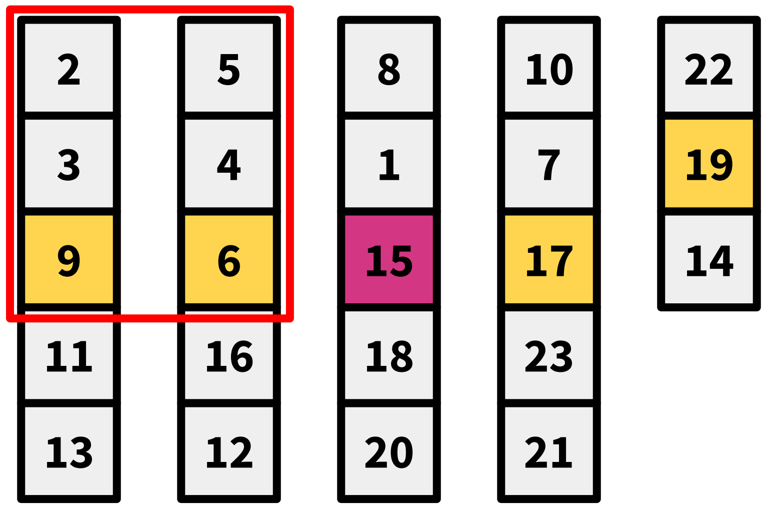
How many elements are smaller than the median of medians?

Analyzing Runtime



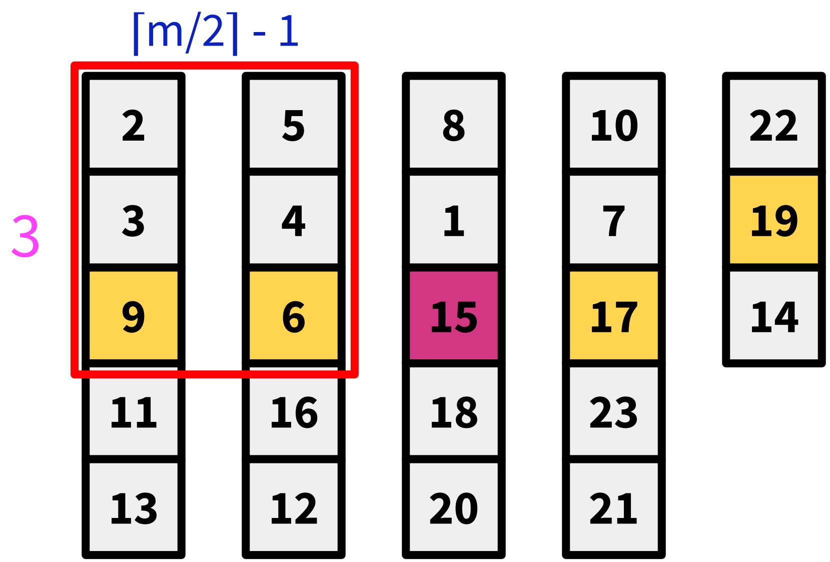
At least these guys (2, 3, 4, 5, 6, 9): everything above and to the left. There might be more (1, 7, 8, 11, 12, 13, 14), but we are guaranteed that *at least* these guys will be smaller.

Analyzing Runtime



How many are there?

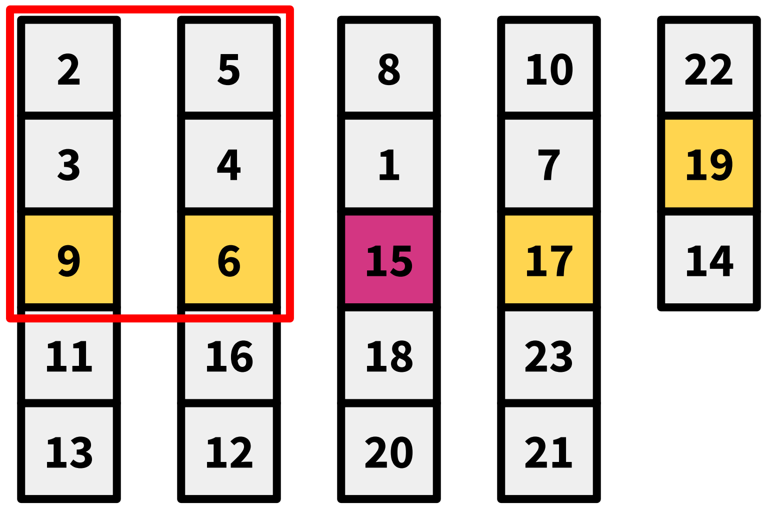
Analyzing Runtime



At least $3 \cdot (\lceil m/2 \rceil - 1)$

Analyzing Runtime

Everything besides this



How many elements are larger than the median of medians?

At most $n - 1 - 3 \cdot (\lceil m/2 \rceil - 1) \le 7n/10 + 2$.

Because $m = \lceil n/5 \rceil$

Analyzing Runtime

We just showed that …

$$3 \cdot (\lceil m/2 \rceil - 1) \leq |L|$$

`smartly_choose_pivot` will choose a pivot greater than at least $3 \cdot (\lceil m/2 \rceil - 1)$ elements.

$$|R| \leq 7n/10 + 2$$

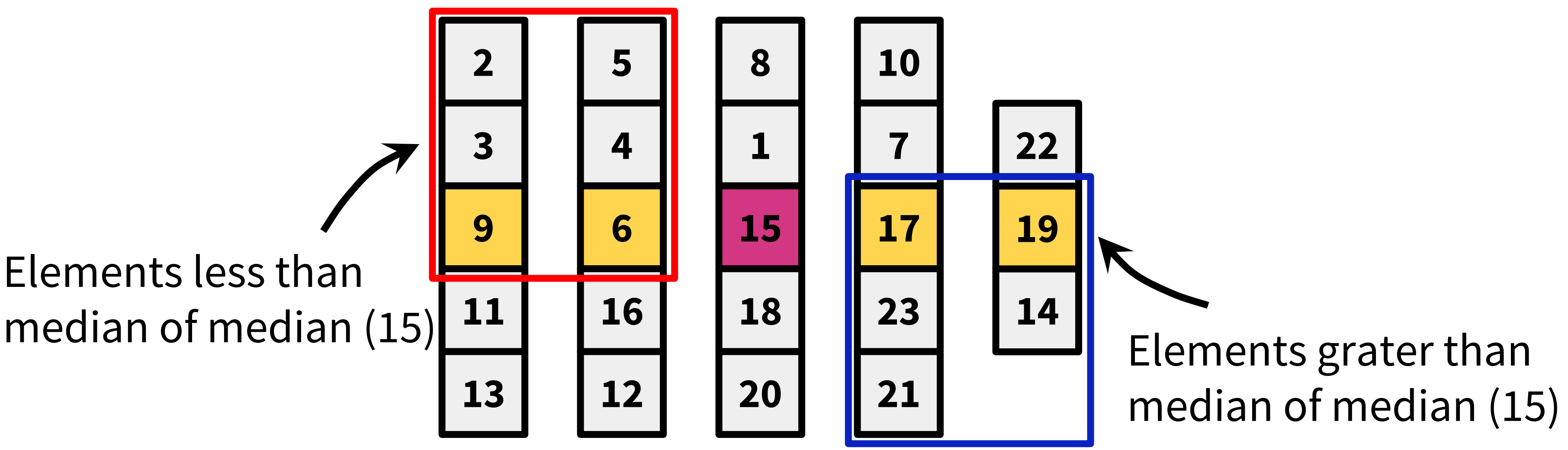
`smartly_choose_pivot` will choose a pivot less than at most $7n/10 + 2$ elements.

Analyzing Runtime

We can just as easily show the inverse.

$$3\cdot(\lceil m/2 \rceil - 1) \le |L| \le 7n/10 + 2$$

$$3\cdot(\lceil m/2 \rceil - 1) \le |R| \le 7n/10 + 2$$



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 + 2 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 + 2 elements.

Analyzing Runtime

```
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])
```

$c_1 \cdot n = O(n)$ — Partition into m=n/5 groups

$c_2 \cdot m = c_2 \cdot n/5 = c_2 \cdot n = O(n)$ — For each group, select its median

$T(\lceil n/5 \rceil)$ — Select the median of medians

$O(n) + T(\lceil n/5 \rceil)$

```
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)
```

$c = O(1)$ — If length is small, naïve select k directly

$O(n) + T(\lceil n/5 \rceil)$ — Otherwise, choose pivot smartly and partition

$T(\lceil 7n/10 + 2 \rceil)$ — Recursive calls

$T(n) = O(n) + T(\lceil n/5 \rceil) + T(\lceil 7n/10 + 2 \rceil)$

Analyzing Runtime

Recurrence relation: $T(n) \leq c \cdot n + T(\lceil n/5 \rceil) + T(\lceil 7n/10 + 2 \rceil)$.

Partitioning, computing n/5 medians

Computing the median of n/5 medians.

Recurrence on L or R.

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. Initial State, if $1 \leq n \leq 100$, then $T(n) \leq c \leq kn$ will be true as long as we pick $k \geq c$.

Induction assumption, assume for some $n \geq 100$, the claim holds for all $1 \leq n' < n$, i.e., $T(n') \leq kn'$. Note that $1 \leq \lceil n/5 \rceil, \lceil 7n/10 + 2 \rceil < n$. Then:

$$
\begin{aligned}
T(n) &\leq T(\lceil n/5 \rceil) + T(\lceil 7n/10 + 2 \rceil) + cn \\
&\leq k \lceil n/5 \rceil + k \lceil 7n/10 + 2 \rceil + cn \\
&= k(n/5 + 1) + k(7n/10 + 2 + 1) + cn \\
&= 9kn/10 + 4k + cn \\
&= kn + (4k + cn - kn/10)
\end{aligned}
$$

Let $4k+cn-kn/10 \leq 0$, then $k \geq 50/3c$.
The initial state requires $k \geq c$, the induction state requires $k \geq 50/3c$, and Such k indeed exists! For example, if we pick $k = 50c$, then $4k + cn - kn/10 \leq 0$ 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)

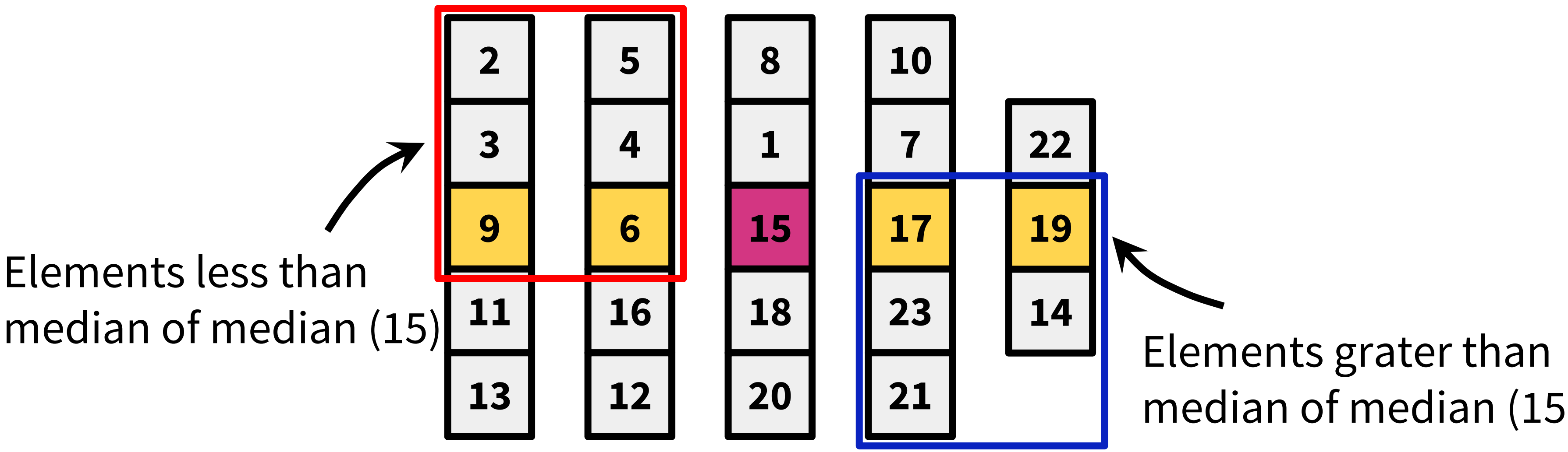
Proceed by induction to prove the bound holds, noting what constraints arise on your undetermined constants (e.g. k).

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. You may need to find tighter values for the numbers during your analysis.

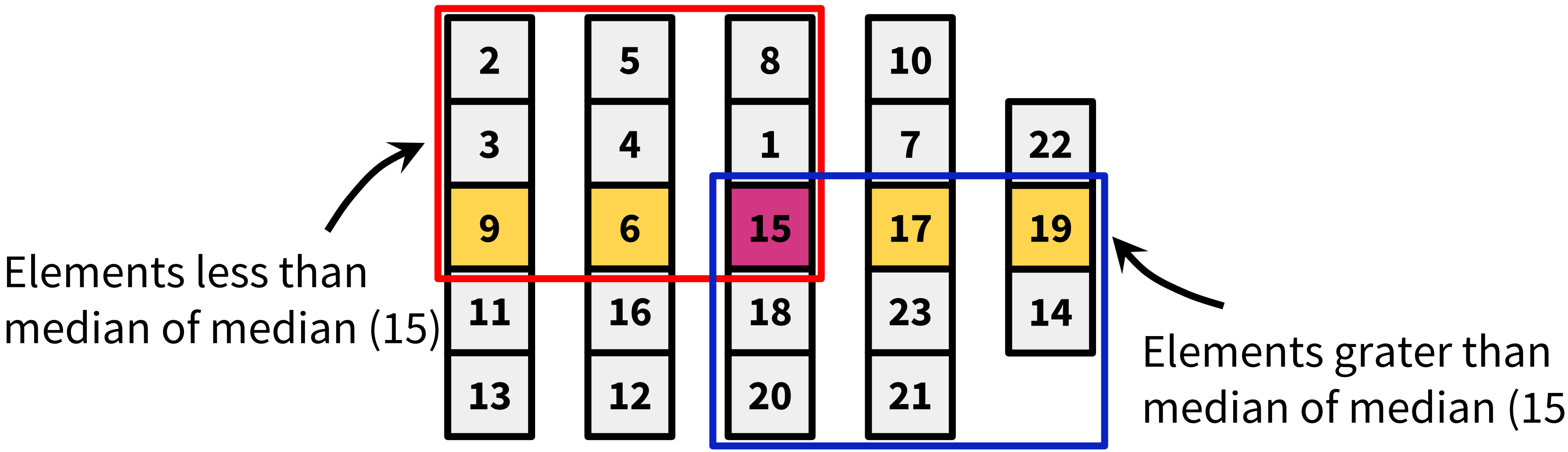
$$3\cdot(\lceil m/2 \rceil - 1) \leq |L| \leq 7n/10 + 2$$

$$3\cdot(\lceil m/2 \rceil - 1) \leq |R| \leq 7n/10 + 2$$



$$3 \cdot (\lceil m/2 \rceil - 1) \leq |L| \leq 7n/10 + 2$$

$$3 \cdot (\lceil m/2 \rceil - 1) \leq |R| \leq 7n/10 + 2$$



$$3 \cdot \lceil m/2 \rceil - 1 \leq |L| \leq 7n/10 - 3$$

$$3 \cdot \lceil m/2 \rceil - 1 \leq |R| \leq 7n/10 - 3$$

This may gives you tighter bounds for inductive proof.

Summary

- Divide and Conquer: Binary Search, Integer Multiplication, Merge Sort, Select K.
- Providing Correctness: Insertion Sort, Select K.
- Solving Recurrences (when sub-problems have the same size): Recursion Tree Method, Iteration Method, Master Method.
- Solving Recurrences (when sub-problems have different size): Substitution method.

Summary

- Divide and Conquer: Binary Search, Integer Multiplication, Merge Sort, Select K.
- Providing Correctness: Insertion Sort, Select K.
- Solving Recurrences (when sub-problems have the same size): Recursion Tree Method, Iteration Method, Master Method.
- Solving Recurrences (when sub-problems have different size): Substitution method.

Acknowledgement: Part of the materials are adapted from Mary Wootter, Virginia Williams and David Eng's lectures on algorithms. We appreciate their contributions.