Module 10 – Algorithm Analysis

SIT320 – Advanced Algorithms

Dr. Nayyar Zaidi

Part 1: Recap from SIT221

- Part 1a: Sorting Algorithms
  - InsertionSort: does it work and is it fast?
  - MergeSort: does it work and is it fast?
- Part Ib: How do we measure the runtime of an algorithm?
  - Worst-case analysis
  - Asymptotic Analysis

Sorting

- Important primitive
- **For today**, we'll pretend all elements are distinct.

6	4	3	8	1	5	2	7

1	2	3	4	5	6	7	8

Benchmark: insertion sort

- Say we want to sort:

6	4	3	8	5

- Insert items one at a time.

- How would we actually implement this?

Insertion Sort

```
def InsertionSort(A):
    for i in range(1,len(A)):
        current = A[i]
        j = i-1
        while j >= 0 and A[j] > current:
            A[j+1] = A[j]
            j -= 1
        A[j+1] = current
```

InsertionSort

example

6	4	3	8	5

Start by moving A[1] toward the beginning of the list until you find something smaller (or can't go any further):

6	4	3	8	5
4	6	3	8	5

Then move A[2]:

4	6	3	8	5
3	4	6	8	5

Then move A[3]:

3	4	6	8	5
3	4	6	8	5

Then move A[4]:

3	4	6	8	5
3	4	5	6	8

Then we are done!

Insertion Sort: running time

```
def InsertionSort(A):
    for i in range(1,len(A)):
        current = A[i]
        j = i-1
        while j >= 0 and A[j] > current:
            A[j+1] = A[j]
            j -= 1
        A[j+1] = current
```

n-1 iterations of the outer loop

In the worst case, about n iterations of this inner loop

Running time scales like n^2

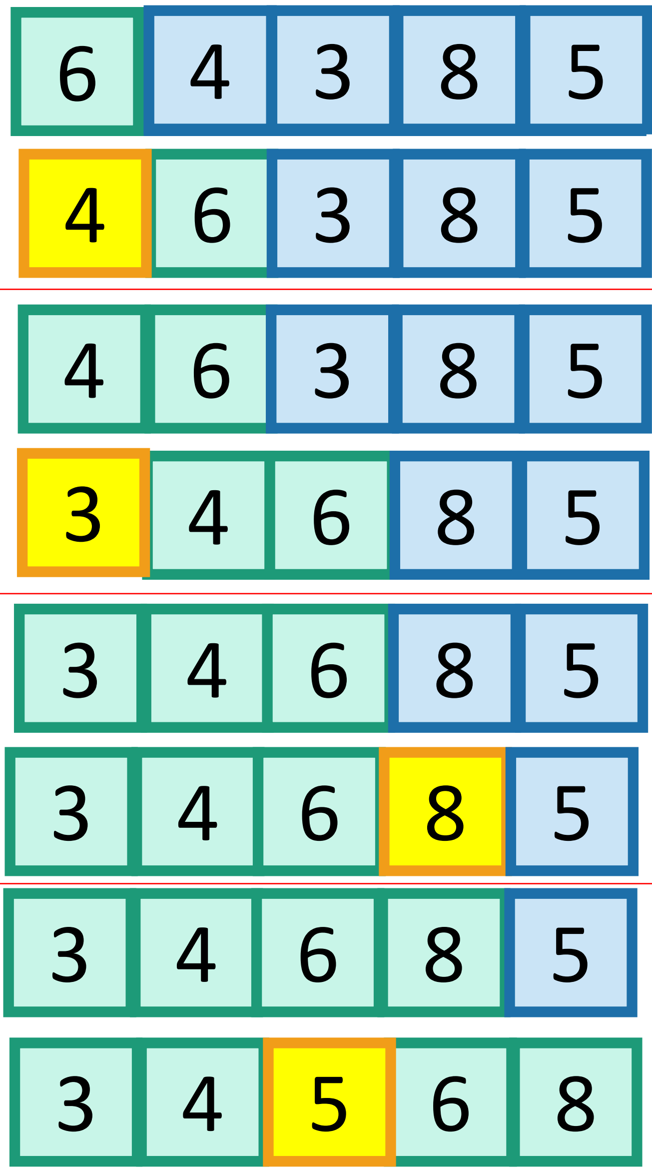
Why does this work?

- Say you have a sorted list, | 3 | 4 | 6 | 8 |, and another element 5.

- Insert 5 right after the largest thing that's still smaller than 5. (Aka, right after 4).

- Then you get a sorted list: | 3 | 4 | 5 | 6 | 8 |

So just use this logic at every step.



The first element, [6], makes up a sorted list.

So correctly inserting 4 into the list [6] means that [4,6] becomes a sorted list.

The first two elements, [4,6], make up a sorted list.

So correctly inserting 3 into the list [4,6] means that [3,4,6] becomes a sorted list.

The first three elements, [3,4,6], make up a sorted list.

So correctly inserting 8 into the list [3,4,6] means that [3,4,6,8] becomes a sorted list.

The first four elements, [3,4,6,8], make up a sorted list.

So correctly inserting 5 into the list [3,4,6,8] means that [3,4,5,6,8] becomes a sorted list.

YAY WE ARE DONE!

Recall: proof by induction

- Maintain a loop invariant.
- Proceed by induction.

A loop invariant is something that should be true at every iteration.

- **Four steps in the proof by induction:**
  - Inductive Hypothesis: The loop invariant holds after the i^{th} iteration.
  - Base case: the loop invariant holds before the 1^{st} iteration.
  - Inductive step: If the loop invariant holds after the i^{th} iteration, then it holds after the $(i+1)^{st}$ iteration
  - Conclusion: If the loop invariant holds after the last iteration, then we win.

Formally: induction

A "loop invariant" is something that we maintain at every iteration of the algorithm.

- Loop invariant(i): `A[:i+1]` is sorted.
- Inductive Hypothesis:
  - The loop invariant(i) holds at the end of the i^{th} iteration (of the outer loop).
- Base case (i=0):
  - Before the algorithm starts, `A[:1]` is sorted. ✓
- Inductive step:
- Conclusion:
  - At the end of the n-1'st iteration (aka, at the end of the algorithm), `A[:n] = A` is sorted.
  - That's what we wanted!✓

This logic (see Lecture Notes for details)

4	6	3	8	5

The first two elements, [4,6], make up a sorted list.

This was iteration i=2.

3	4	6	8	5

So correctly inserting 3 into the list [4,6] means that [3,4,6] becomes a sorted list.

To summarize

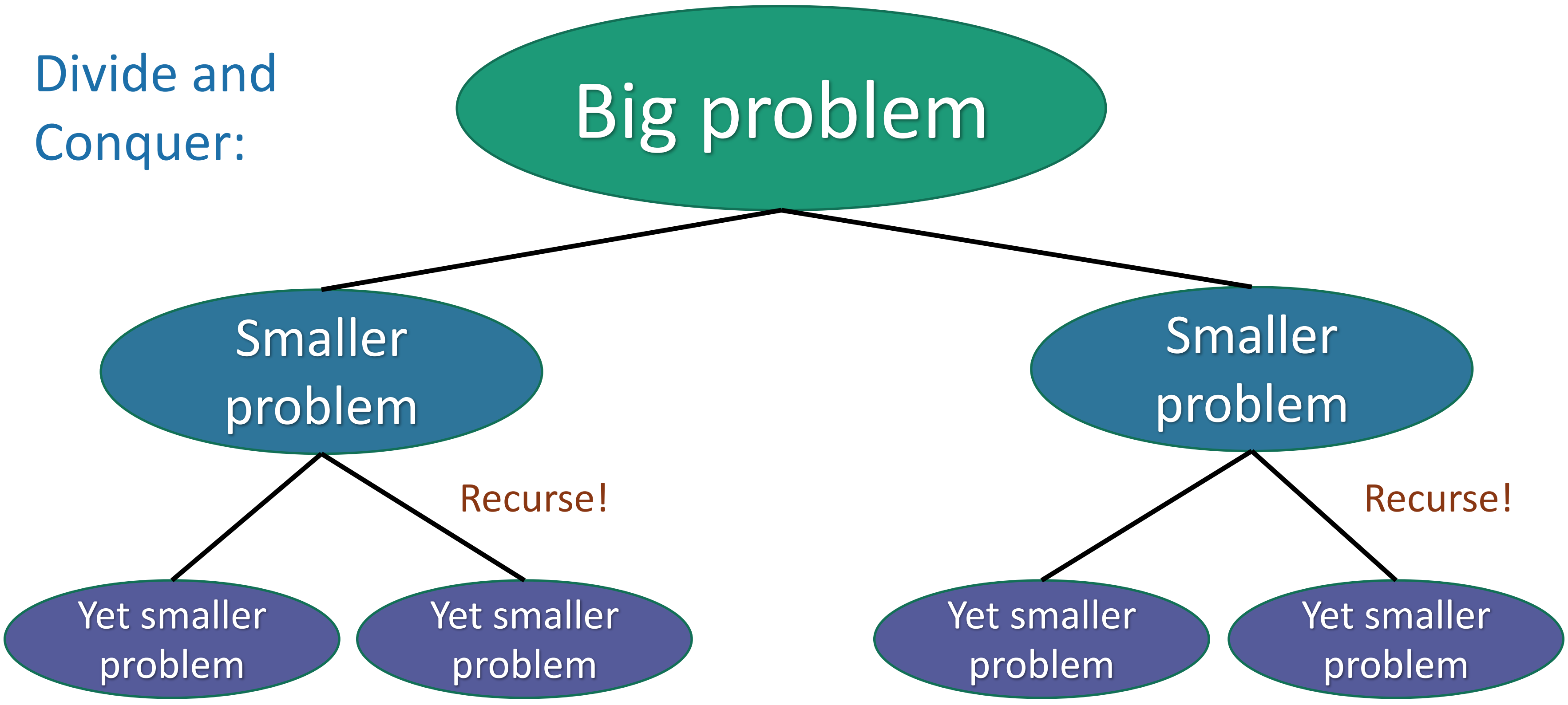
InsertionSort is an algorithm that correctly sorts an arbitrary n-element array in time that scales like n^2.

Can we do better?

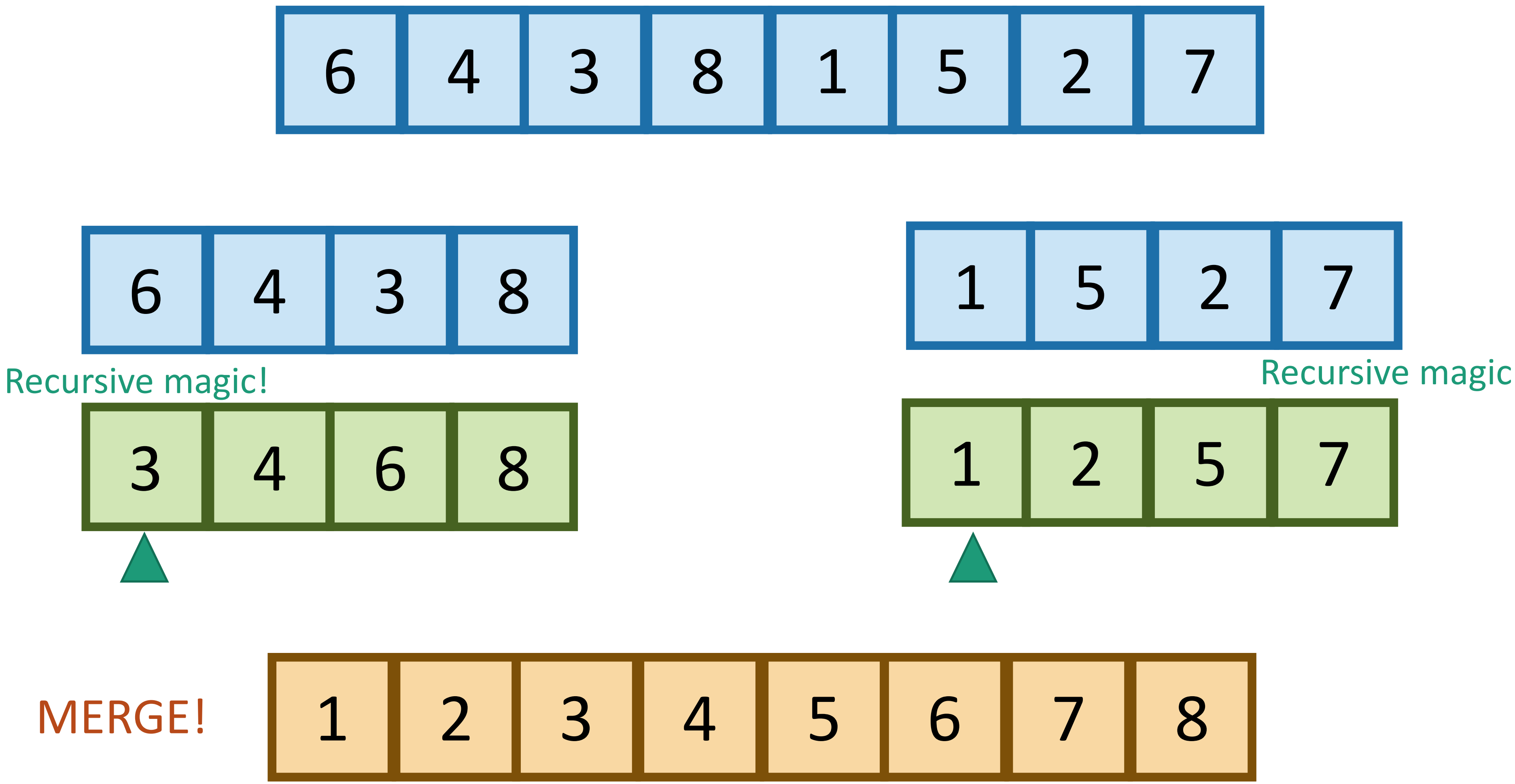
Can we do better?

- MergeSort: a divide-and-conquer approach
- Recall from last time:

Divide and Conquer:



MergeSort



How would you
do this in-place?

MergeSort Pseudocode

MERGESORT(A):

- n = length(A)
- **if** n $\leq$ 1:
  - **return** A

  If A has length 1,
  It is already sorted!
- L = MERGESORT(A[0 : n/2])

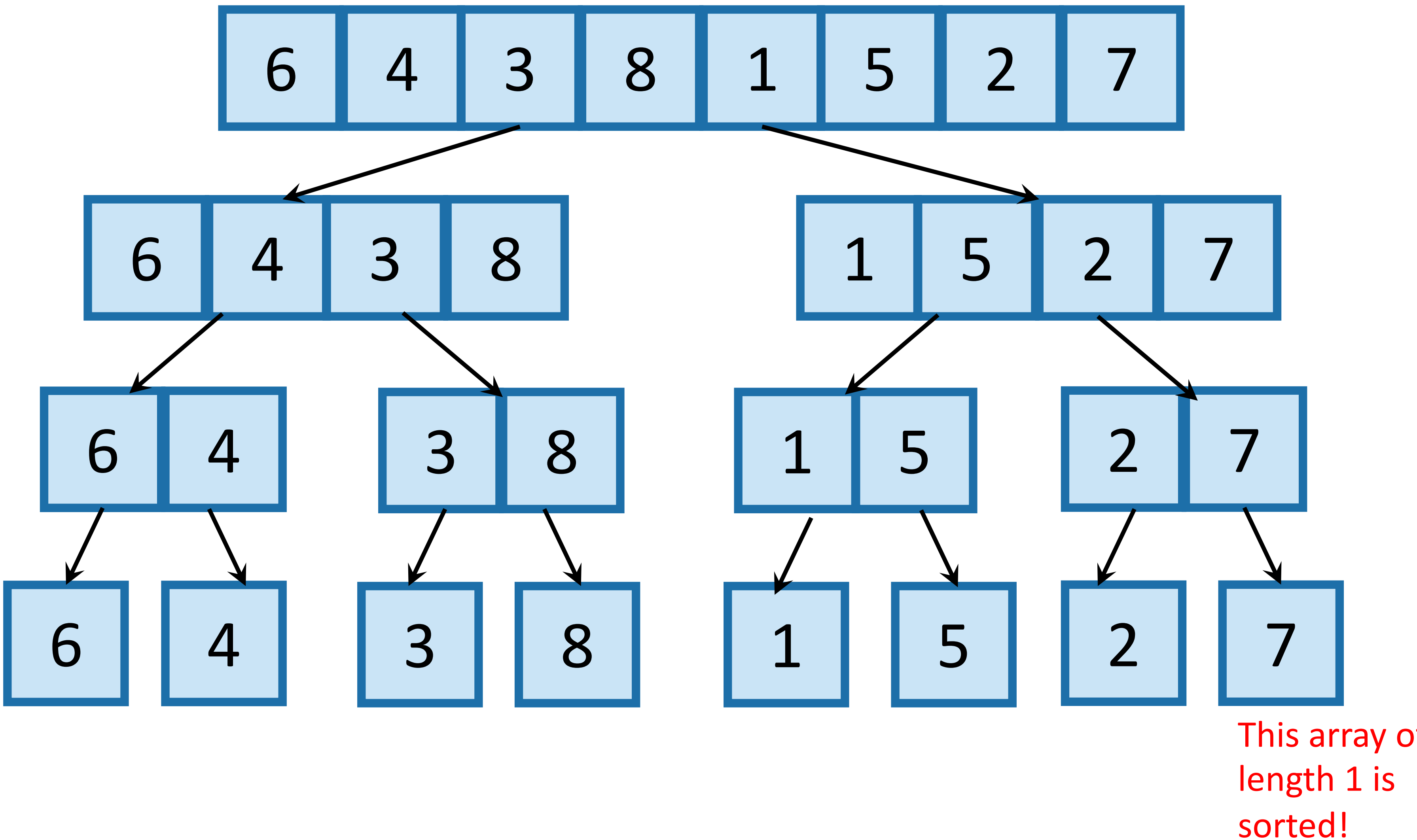
  Sort the left half
- R = MERGESORT(A[n/2 : n])

  Sort the right half
- **return** MERGE(L,R)

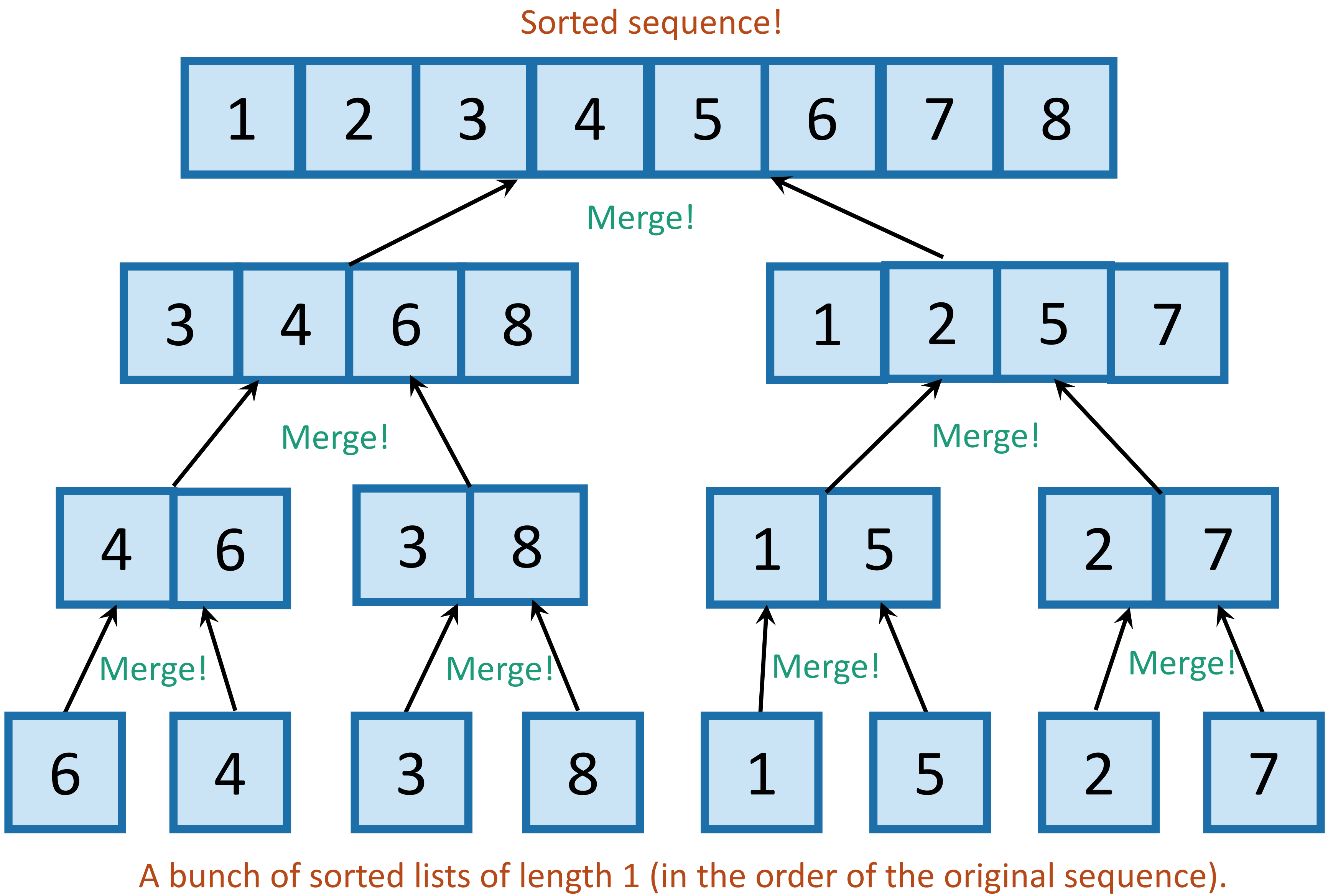
  Merge the two halves

What actually happens?

First, recursively break up the array all the way down to the base cases



Then, merge them all back up!



Two questions

1. Does this work?
2. Is it fast?

Empirically:

1. **Seems to.**
2. **Maybe?**

It works   Let's assume $n = 2^t$

Again we'll use induction. This time with an invariant that will remain true after every recursive call.

- Inductive hypothesis:

"In every recursive call,
MERGESORT returns a sorted array."

- Base case (n=1): a 1-element array is always sorted.
- Inductive step: Suppose that L and R are sorted. Then MERGE(L,R) is sorted.
- Conclusion: "In the top recursive call, MERGESORT returns a sorted array."

- n = length(A)
- **if** $n \leq 1$:
  - **return** A
- L = MERGESORT(A[1 : n/2])
- R = MERGESORT(A[n/2+1 : n])
- **return** MERGE(L,R)

It’s fast Let’s keep assuming $n = 2^t$

CLAIM:

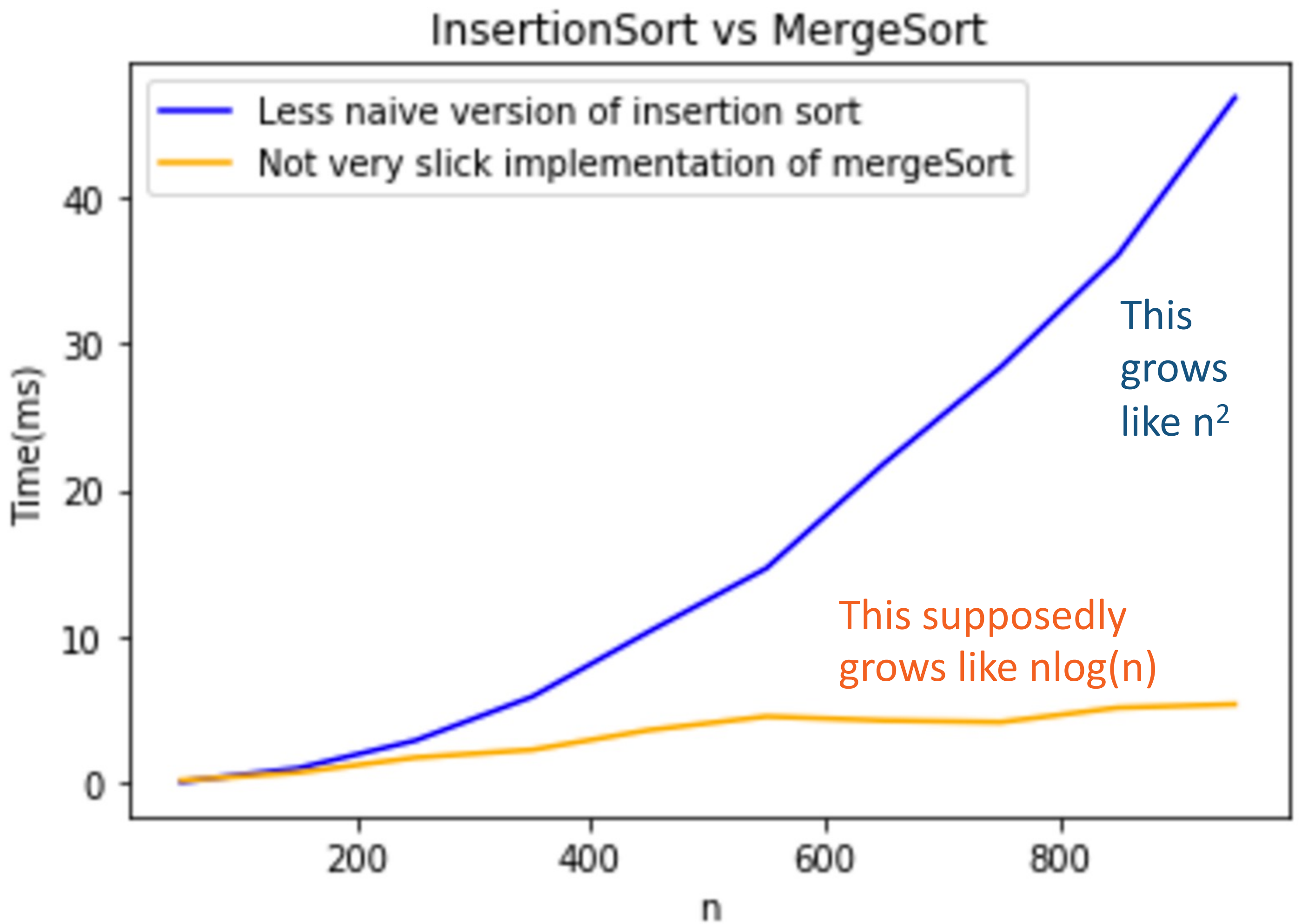
MERGESORT requires at most $11n(\log(n) + 1)$ operations to sort n numbers.

What exactly is an “operation” here?
We’re leaving that vague on purpose.
Also I made up the number 11.

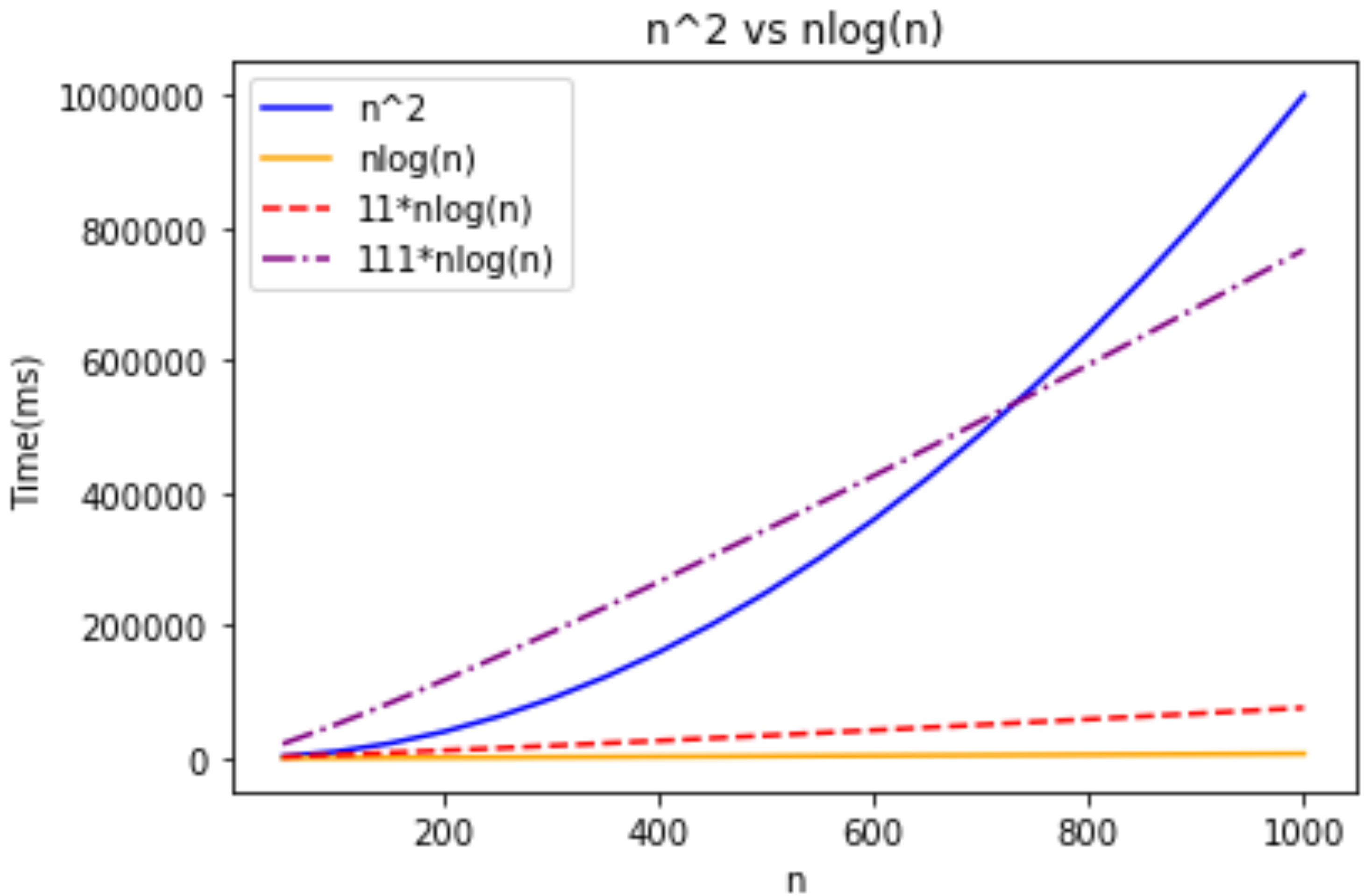
How does this compare to InsertionSort?

Scaling like n^2 vs scaling like $n\log(n)$?

Empirically



The constant doesn't matter:
eventually, $n^2 > 111111 \cdot n\log(n)$



Quick log refresher

All logarithms in this course are base 2

- log(n) : how many times do you need to divide n by 2 in order to get down to 1?

32
16
8
4
2
1

log(32) = 5

64
32
16
8
4
2
1

log(64) = 6

log(128) = 7
log(256) = 8
log(512) = 9
.
.
.
log(**number of particles in the universe**) < 280

Moral: log(n) grows very slowly with n.

It's fast!

CLAIM:

MERGESORT requires at most $11n(\log(n) + 1)$ operations to sort n numbers.

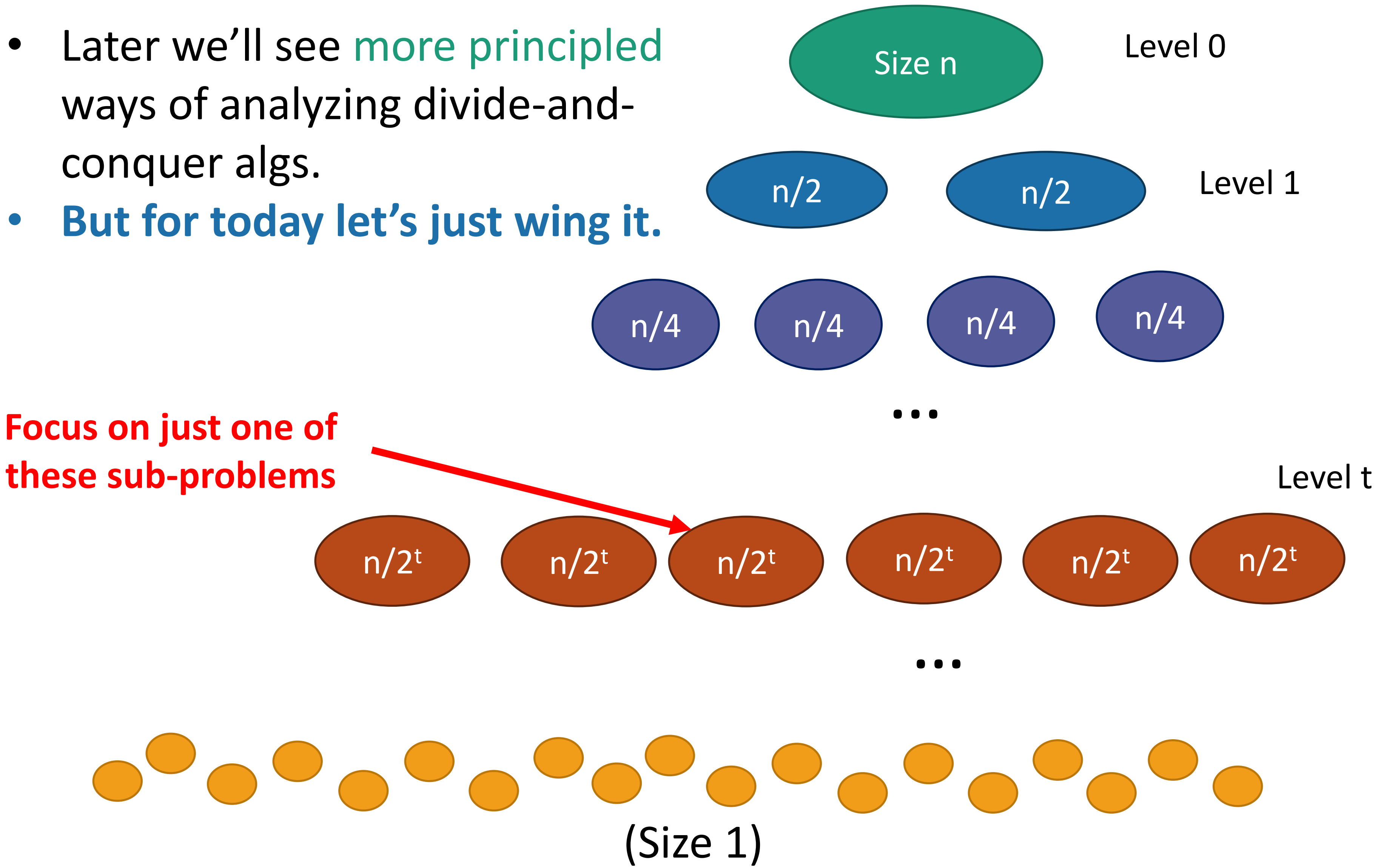
Much faster than InsertionSort for large n!

(No matter how the algorithms are implemented).

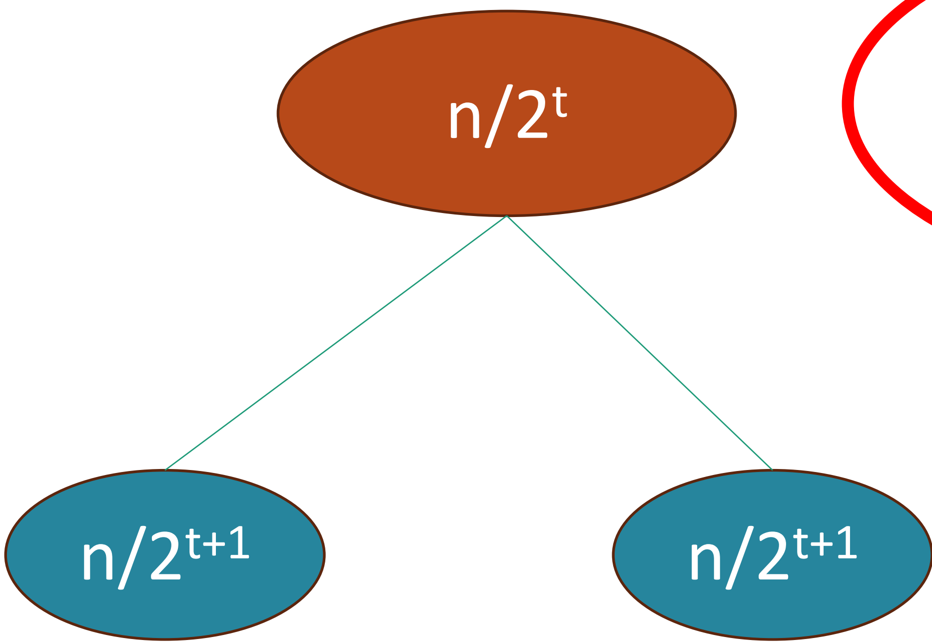
(And no matter what that constant "11" is).

Let’s prove the claim

- Later we’ll see more principled ways of analyzing divide-and-conquer algs.
- **But for today let’s just wing it.**



How much work in this sub-problem?



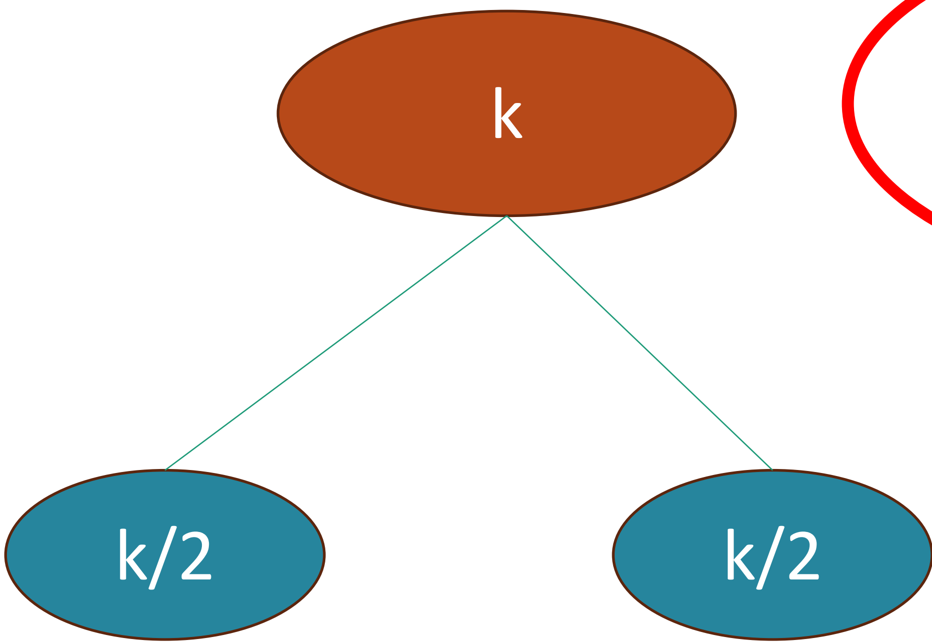
Time spent MERGE-ing the two subproblems

+

Time spent within the two sub-problems

How much work in this sub-problem?

Let $k=n/2^t$...

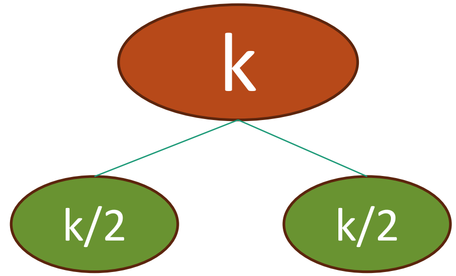


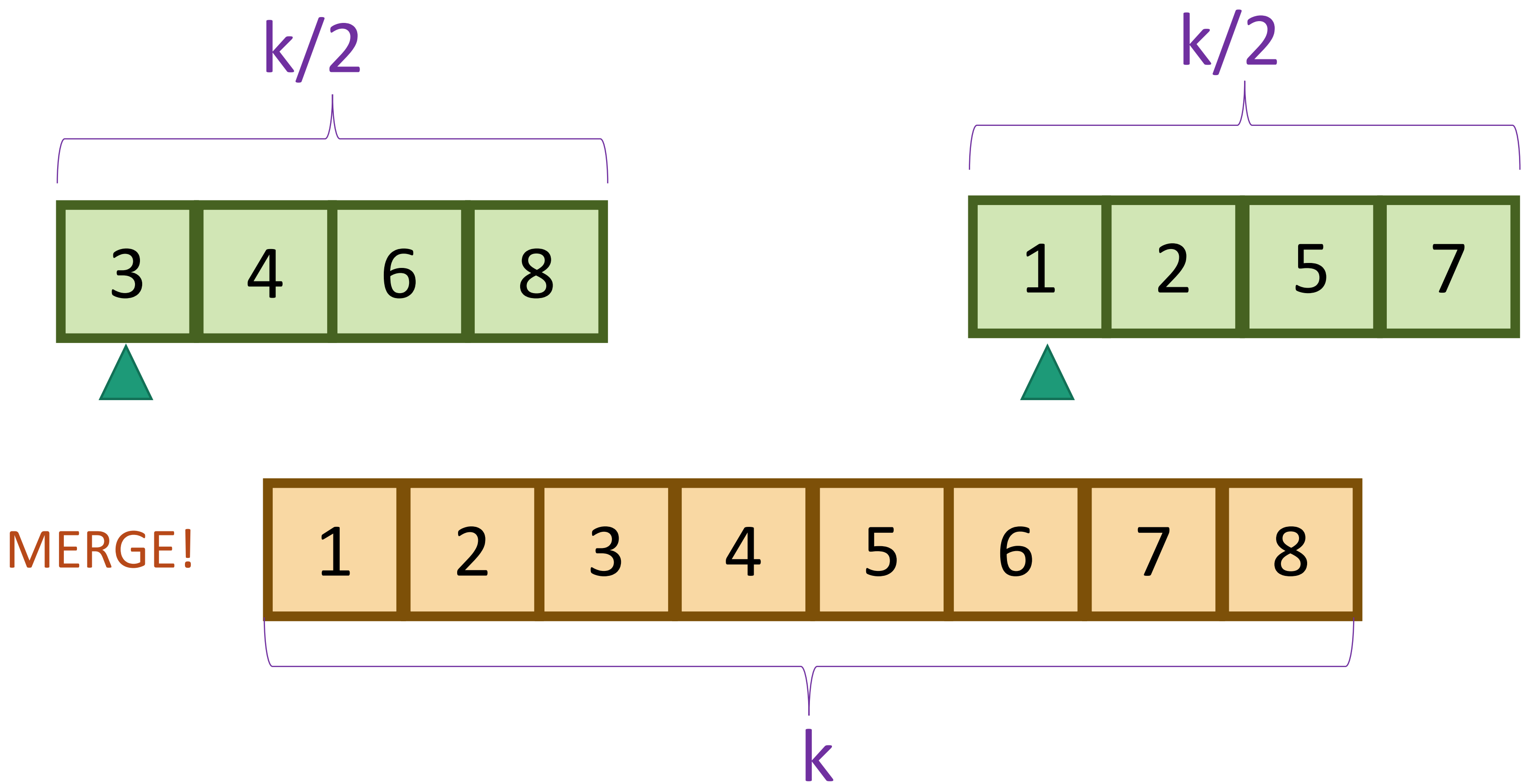
Time spent MERGE-ing the two subproblems

+

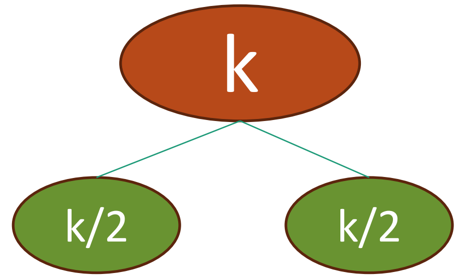
Time spent within the two sub-problems

How long does it take to MERGE?





How long does it take to MERGE?



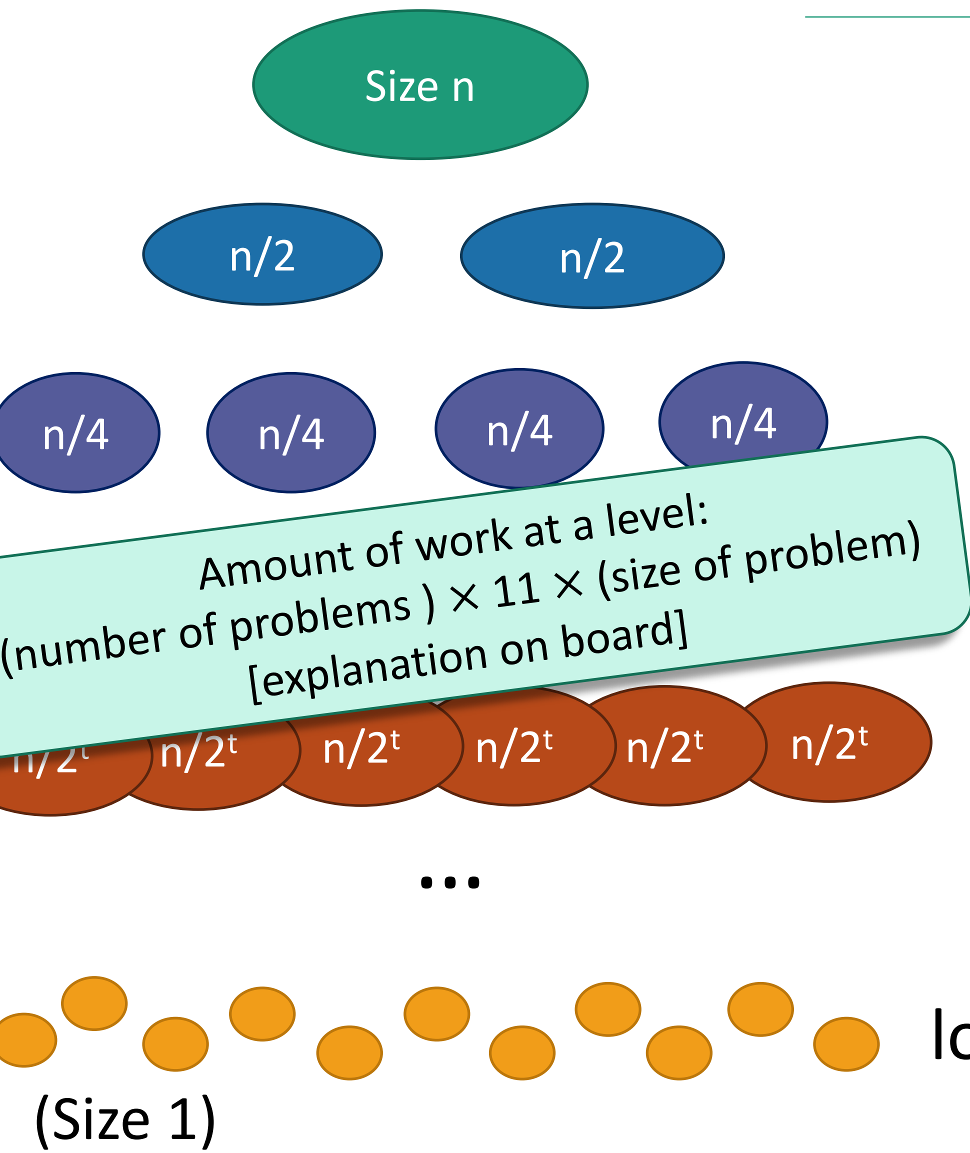
Code for the MERGE step is given in the Lecture2 notebook.

- Time to initialize an array of size k
- Plus the time to initialize three counters
- Plus the time to increment two of those counters k/2 times each
- Plus the time to compare two values at least k times
- Plus the time to copy k values from the existing array to the big array.
- Plus…

Let's say no more than 11k operations.

There's some justification for this number "11" in the lecture notes, but it's really pretty arbitrary.

Recursion tree



Level	# problems	Size of each problem	Amount of work at this level
0	1	n	$11n$
1	2	$n/2$	$11n$
2	4	$n/4$	$11n$
…			
t	2^t	$n/2^t$	$11n$
…			
log(n)	n	1	$11n$

Total runtime…

- 11n steps per level, at every level
- log(n) + 1 levels
- 11n (log(n) + 1) steps total

That was the claim!

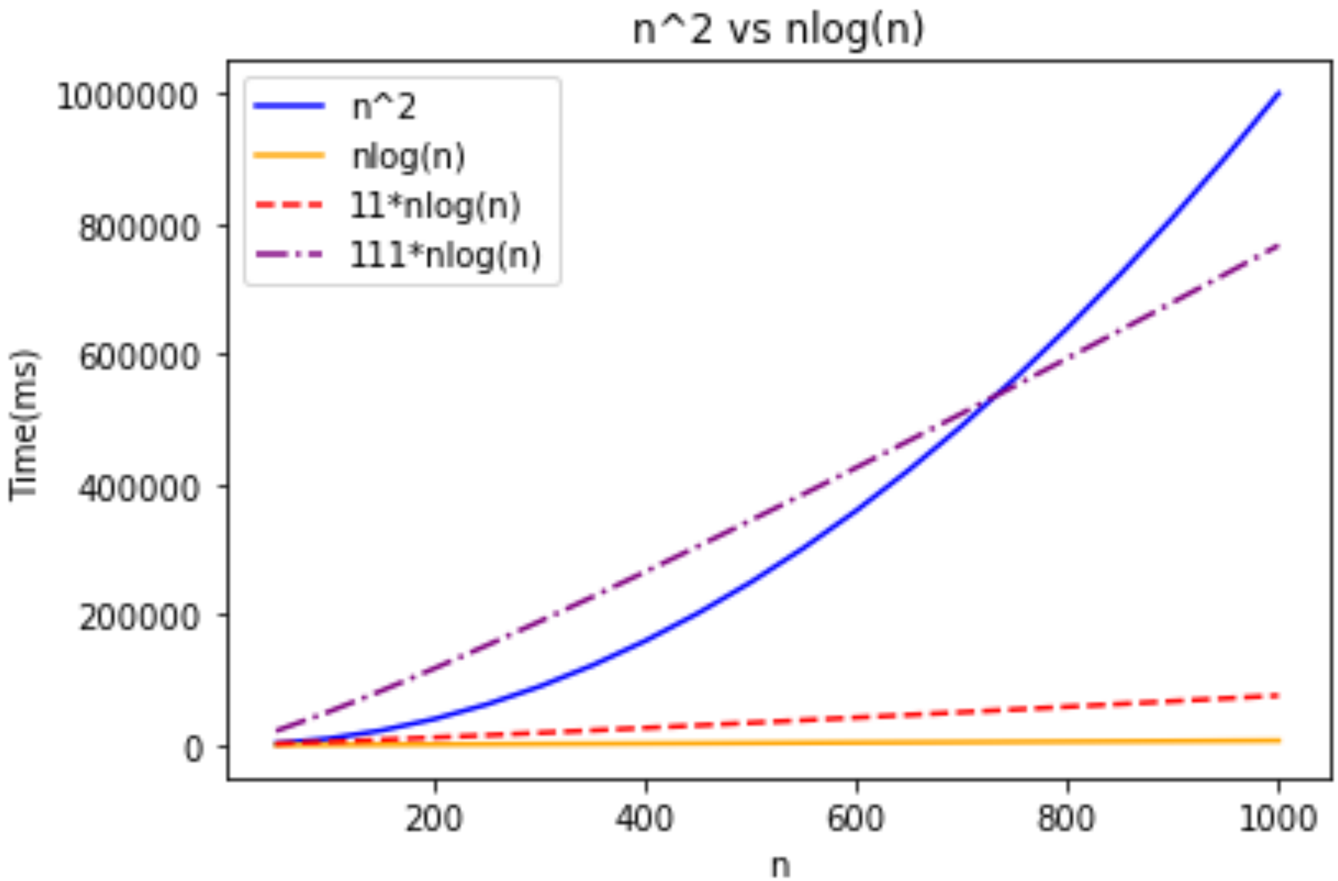
Big-O notation

How long does an operation take? Why are we being so sloppy about that “11”?

- What do we mean when we measure runtime?
  - We probably care about wall time: how long does it take to solve the problem, in seconds or minutes or hours?
- This is heavily dependent on the programming language, architecture, etc.
- These things are very important, but are not the point of this class.
- We want a way to talk about the running time of an algorithm, independent of these considerations.

Main idea:

Focus on how the runtime **scales** with n (the input size).



Asymptotic Analysis

How does the running time scale as n gets large?

One algorithm is "faster" than another if its runtime scales better with the size of the input.

Pros:

- Abstracts away from hardware- and language-specific issues.
- Makes algorithm analysis much more tractable.

Without making Plucky grumpy!

Cons:

- Only makes sense if n is large (compared to the constant factors).

$2^{100000000000000}\, n$ is "better" than n^2 ?!?!

pronounced “big-oh of …” or sometimes “oh of …”

O(…) means an upper bound

- Let T(n), g(n) be functions of positive integers.
  - Think of T(n) as being a runtime: positive and increasing in n.

- We say “T(n) is O(g(n))” if g(n) grows at least as fast as T(n) as n gets large.

- Formally,

$$T(n) = O\big(g(n)\big)$$

$$\Leftrightarrow$$

$$\exists c, n_0 > 0 \;\; s.t. \;\; \forall n \geq n_0,$$

$$0 \leq T(n) \leq c \cdot g(n)$$

Example

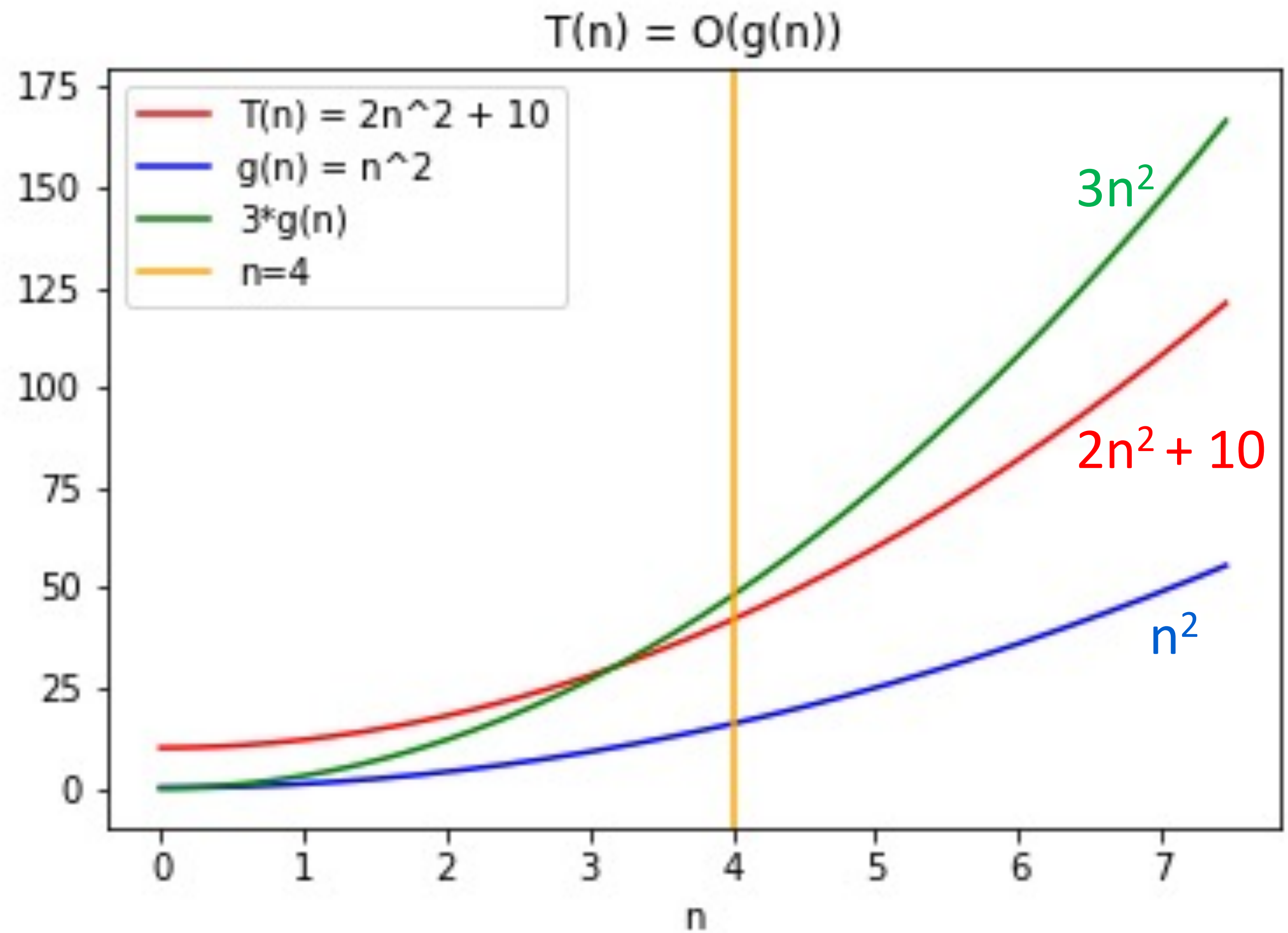
$2n^2 + 10 = O(n^2)$

$T(n) = O\big(g(n)\big)$

$\Leftrightarrow$

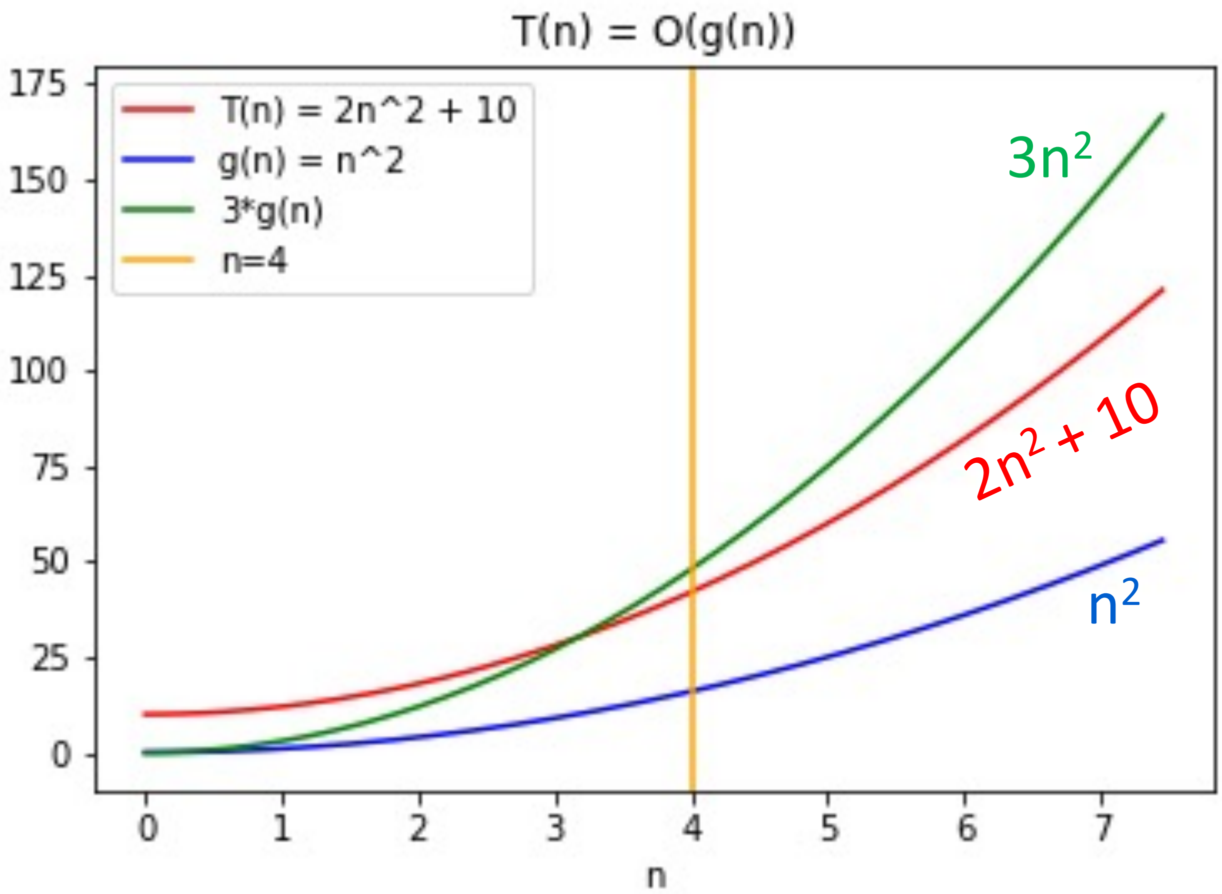
$\exists c, n_0 > 0 \;\; s.t. \;\; \forall n \geq n_0,$

$0 \leq T(n) \leq c \cdot g(n)$



Example
$2n^2 + 10 = O(n^2)$

$$T(n) = O(g(n))$$
$$\Leftrightarrow$$
$$\exists c, n_0 > 0 \;\; s.t. \;\; \forall n \geq n_0,$$
$$0 \leq T(n) \leq c \cdot g(n)$$



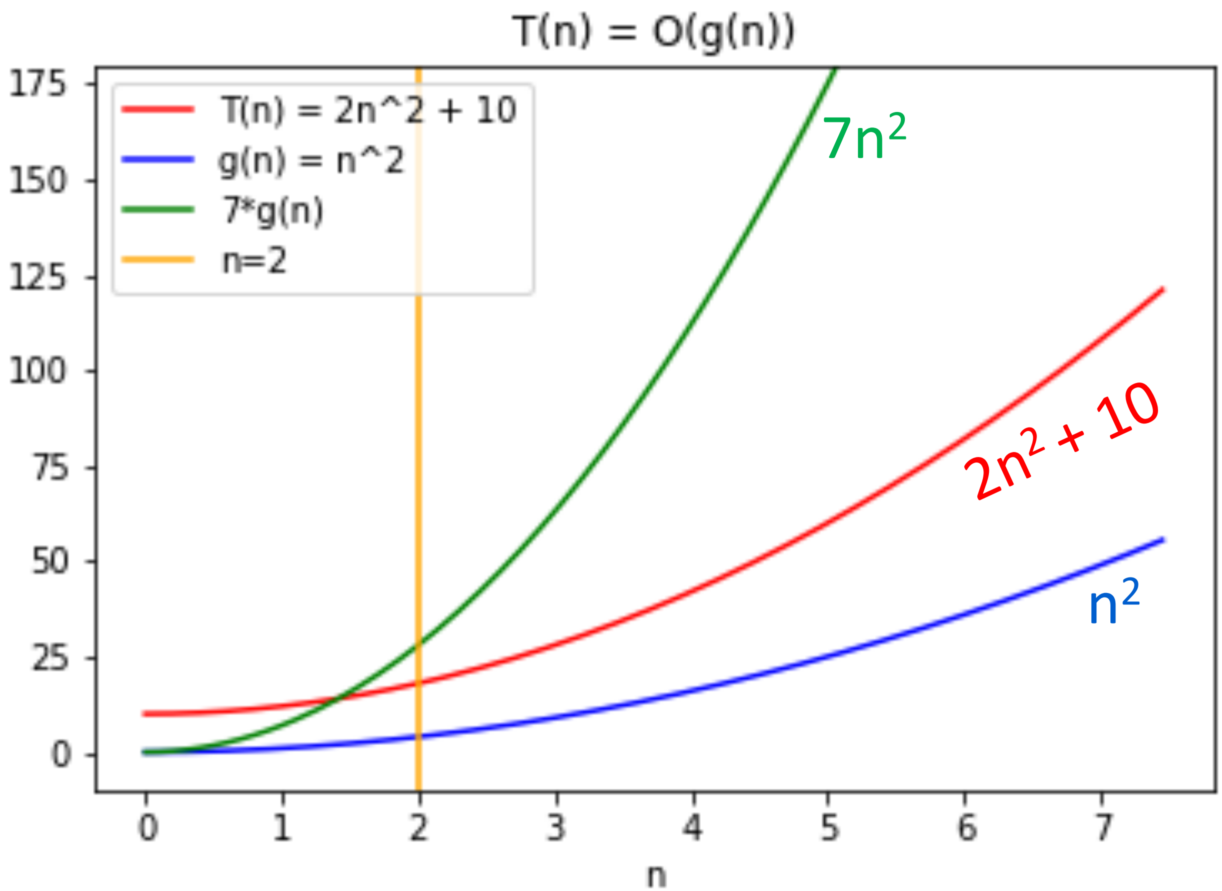
Formally:
- Choose c = 3
- Choose $n_0 = 4$
- Then:

$$\forall n \geq 4,$$
$$0 \leq 2n^2 + 10 \leq 3 \cdot n^2$$

same Example

$2n^2 + 10 = O(n^2)$

$$T(n) = O(g(n))$$
$$\Leftrightarrow$$
$$\exists c, n_0 > 0 \;\; s.t. \;\; \forall n \geq n_0,$$
$$0 \leq T(n) \leq c \cdot g(n)$$



Formally:

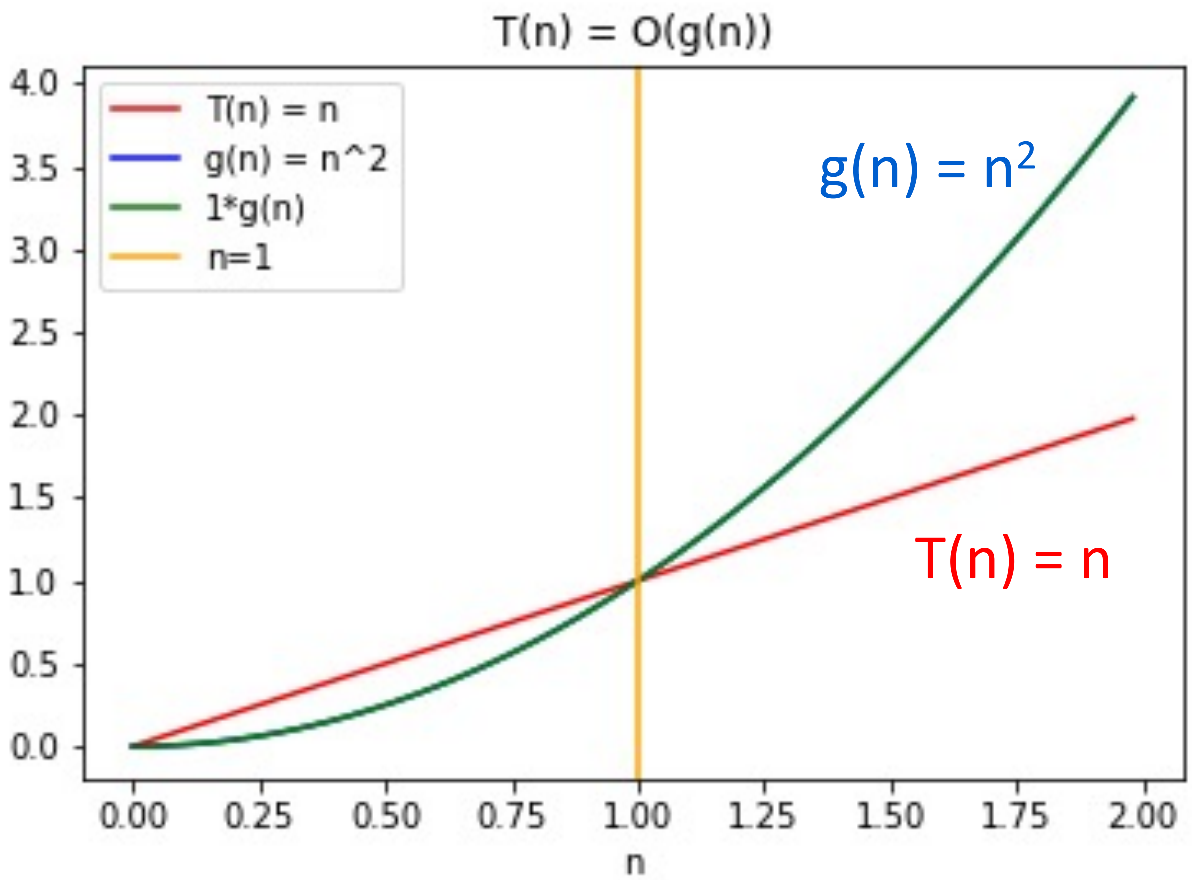
- Choose c = 7
- Choose n_0 = 2
- Then:

$$\forall n \geq 2,$$
$$0 \leq 2n^2 + 10 \leq 7 \cdot n^2$$

There is not a "correct" choice of c and n_0

Another example:
$n = O(n^2)$

$$T(n) = O\big(g(n)\big)$$
$$\Leftrightarrow$$
$$\exists c, n_0 > 0 \;\; s.t. \;\; \forall n \geq n_0,$$
$$0 \leq T(n) \leq c \cdot g(n)$$



- Choose c = 1
- Choose n_0 = 1
- Then

$$\forall n \geq 1,$$
$$0 \leq n \leq n^2$$

Ω(…) means a lower bound

- We say "T(n) is Ω(g(n))" if g(n) grows at most as fast as T(n) as n gets large.

- Formally,

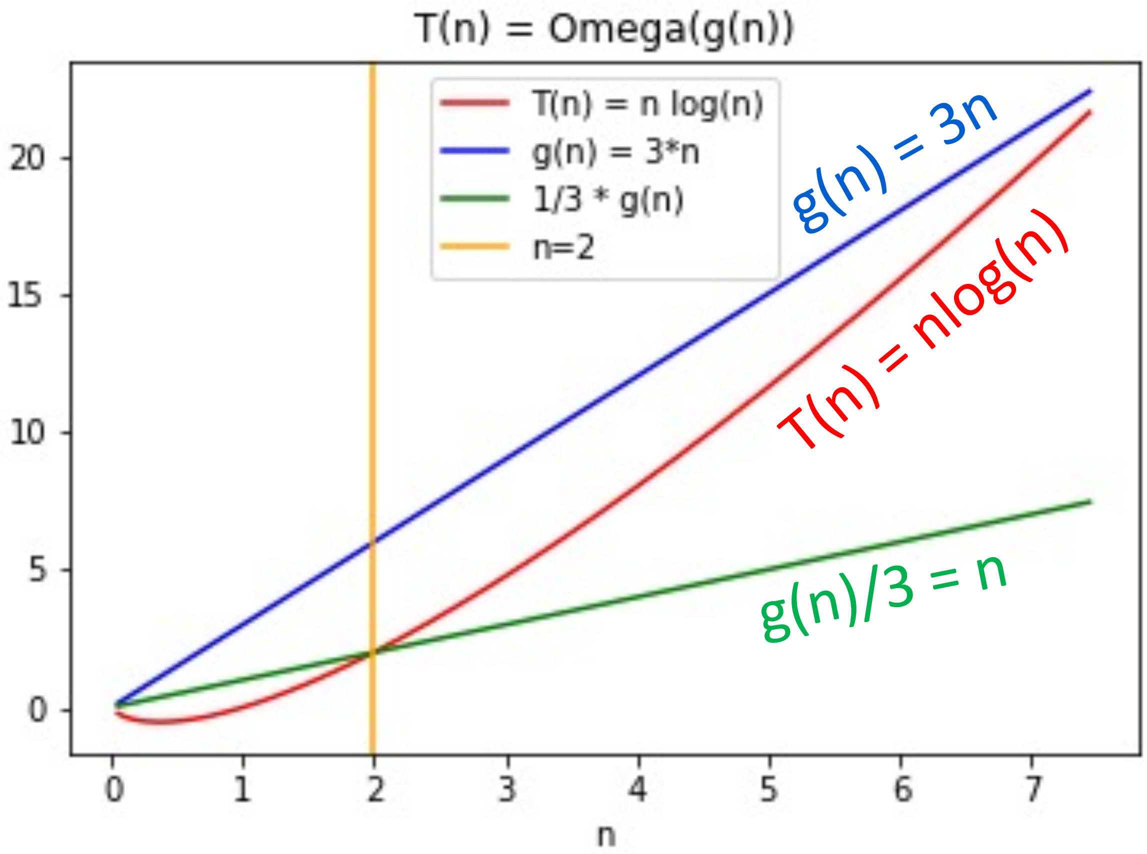
$$T(n) = \Omega\big(g(n)\big)$$
$$\Longleftrightarrow$$
$$\exists c, n_0 > 0 \;\; s.t. \;\; \forall n \geq n_0,$$
$$0 \leq c \cdot g(n) \leq T(n)$$

Switched these!!

Example

$n\log_2(n) = \Omega(3n)$

$$T(n) = \Omega\big(g(n)\big)$$
$$\Leftrightarrow$$
$$\exists c, n_0 > 0 \ \ s.t.\ \ \forall n \geq n_0,$$
$$0 \leq c \cdot g(n) \leq T(n)$$



- Choose c = 1/3
- Choose n_0 = 3
- Then

$$\forall n \geq 3,$$

$$0 \leq \frac{3n}{3} \leq n\log_2(n)$$

$\Theta(\ldots)$ means both!

- We say "$T(n)$ is $\Theta(g(n))$" if:

$$T(n) = O(g(n))$$

-AND-

$$T(n) = \Omega(g(n))$$

Take-away from examples

- To prove $T(n) = O(g(n))$, you have to come up with c and n_0 so that the definition is satisfied.

- To prove $T(n)$ is NOT $O(g(n))$, one way is **proof by contradiction**:
  - Suppose (to get a contradiction) that someone gives you a c and an n_0 so that the definition ***is*** satisfied.
  - Show that this someone must by lying to you by deriving a contradiction.

Part 2

- Recurrence Relations!
  - How do we measure the runtime a recursive algorithm?
  - Like **Integer Multiplication** and **MergeSort**?

- The ***Master Method***
  - A useful theorem so we don't have to answer this question from scratch each time.

Running time of MergeSort

- Let's call this running time T(n).
  - when the input has length n.

- We know that T(n) = O(nlog(n)).

- But if we didn't know that…

$$T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + 11 \cdot n$$

From last time

```
MERGESORT(A):
    n = length(A)
    if n ≤ 1:
        return A
    L = MERGESORT(A[:n/2])
    R = MERGESORT(A[n/2:])
    return MERGE(L,R)
```

Recurrence Relations

- $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + 11 \cdot n$ is a **recurrence relation.**
- It gives us a formula for T(n) in terms of T(less than n)

- The challenge:

Given a recurrence relation for T(n), find a closed form expression for T(n).

- For example, T(n) = O(nlog(n))

Technicalities I
Base Cases

- Formally, we should always have base cases with recurrence relations.
- $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + 11 \cdot n$ with $T(1) = 1$

is not the same as

- $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + 11 \cdot n$ with $T(1) = 1000000000$
- However, T(1) = O(1), so sometimes we'll just omit it.

Why does T(1) = O(1)?

Examples

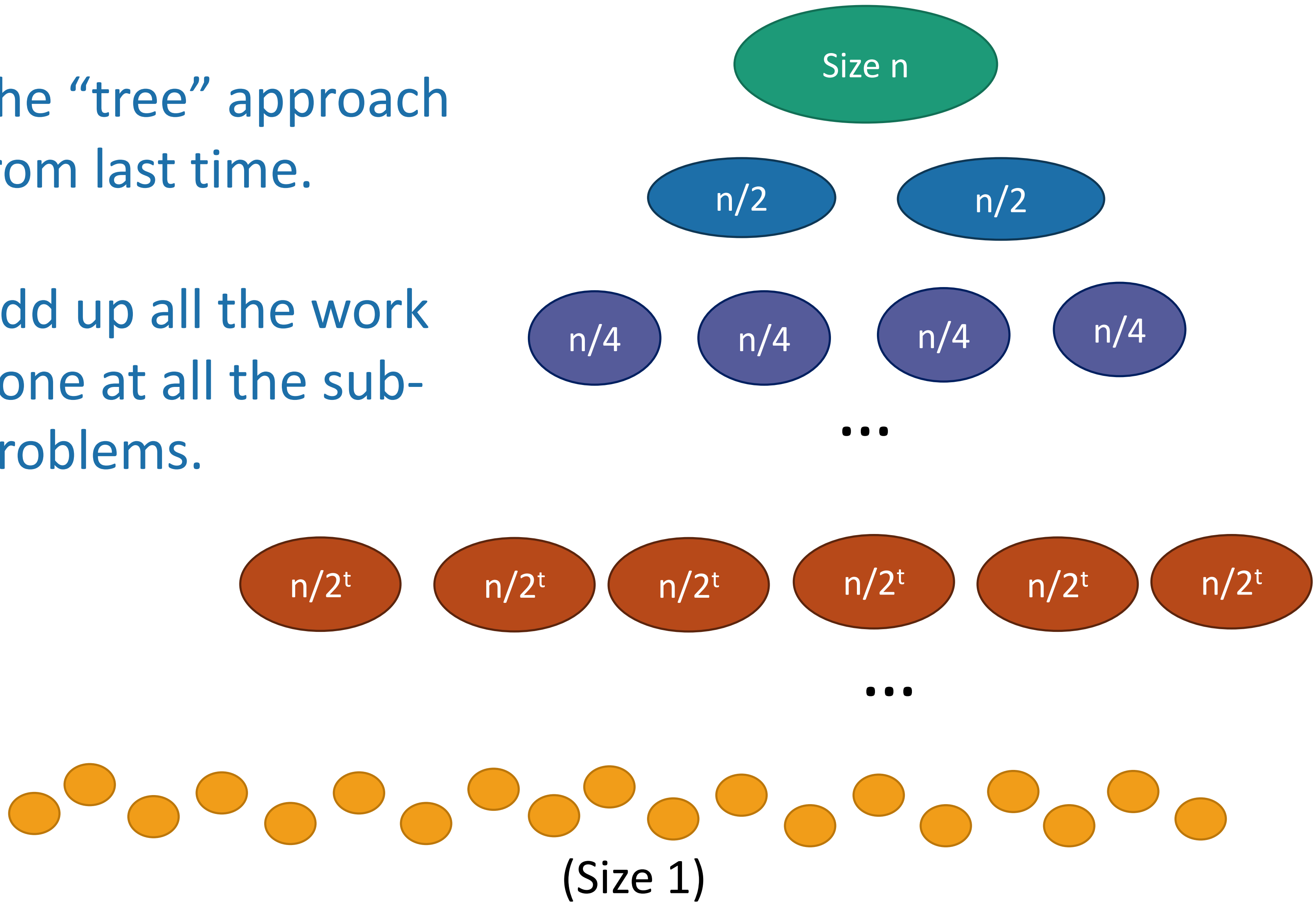
- You played around with these examples (when n is a power of 2):
  1. $T(n) = T\left(\frac{n}{2}\right) + n, \qquad T(1) = 1$
  2. $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n, \qquad T(1) = 1$
  3. $T(n) = 4 \cdot T\left(\frac{n}{2}\right) + n, \qquad T(1) = 1$

- What are the closed forms?

  [Cloud Deakin Module Page]

One approach for all of these

- The "tree" approach from last time.
- Add up all the work done at all the sub-problems.



Another approach:

Recursively apply the relationship a bunch until you see a pattern.

Formally, this should be accompanied with a proof that the pattern holds!

More next time.

- $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + 11 \cdot n$
- $T(n) = 2 \cdot \left(2 \cdot T\left(\frac{n}{4}\right) + \frac{11 \cdot n}{2}\right) + 11 \cdot n$
- $T(n) = 4 \cdot T\left(\frac{n}{4}\right) + 22 \cdot n$
- $T(n) = 4 \cdot \left(2 \cdot T\left(\frac{n}{8}\right) + \frac{11 \cdot n}{4}\right) + 22 \cdot n$
- $T(n) = 8 \cdot T\left(\frac{n}{8}\right) + 33 \cdot n$
- Following the pattern…
- $\boldsymbol{T(n) = n \cdot T(1) + 11 \cdot log(n) \cdot n = O(n \cdot \log(n))}$

More examples

T(n) = time to solve a problem of size n.

- Needlessly recursive integer multiplication
- $T(n) = 4\,T(n/2) + O(n)$
- $T(n) = O(\,n^2\,)$

- Karatsuba integer multiplication
- $T(n) = 3\,T(n/2) + O(n)$
- $T(n) = O(\,n^{\log_2(3)} \approx n^{1.6}\,)$

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

These two are the same as the ones on your pre-lecture exercise.

What's the pattern?!?!?!?!

The master theorem

- A **formula** that solves recurrences when all of the sub-problems are the same size.
  - We'll see an example Wednesday when not all problems are the same size.

- "Generalized" tree method.

A useful formula it is.
Know why it works you should.

Jedi master Yoda

The master theorem

We can also take n/b to mean either $\left\lfloor \frac{n}{b} \right\rfloor$ or $\left\lceil \frac{n}{b} \right\rceil$ and the theorem is still true.

- Suppose $T(n) = a \cdot T\left(\frac{n}{b}\right) + O(n^d)$. Then

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

Three parameters:

a : number of subproblems

b : factor by which input size shrinks

d : need to do n^d work to create all the subproblems and combine their solutions.

Many symbols those are….

Technicalities II

Integer division

Plucky the
Pedantic Penguin

- If n is odd, I can't break it up into two problems of size n/2.

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + O(n)$$

- However (see CLRS, Section 4.6.2), one can show that the Master theorem works fine if you pretend that what you have is:

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + \ O(n)$$

- From now on we'll mostly ignore floors and ceilings in recurrence relations.

Examples

(details on board)

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + O(n^d).$$

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

- Needlessly recursive integer mult.
  - T(n) = 4 T(n/2) + O(n)
  - T(n) = O(n^2)

a = 4
b = 2
d = 1

$a > b^d$ ✓

- Karatsuba integer multiplication
  - T(n) = 3 T(n/2) + O(n)
  - T(n) = O($n^{\text{log\_2(3)}} \approx n^{1.6}$)

a = 3
b = 2
d = 1

$a > b^d$ ✓

- MergeSort
  - T(n) = 2T(n/2) + O(n)
  - T(n) = O(nlog(n))

a = 2
b = 2
d = 1

$a = b^d$ ✓

- That other one
  - T(n) = T(n/2) + O(n)
  - T(n) = O(n)

a = 1
b = 2
d = 1

$a < b^d$ ✓

Proof of the master theorem

- We'll do the same recursion tree thing we did for MergeSort, but be more careful.
- Suppose that $T(n) = a \cdot T\left(\frac{n}{b}\right) + c \cdot n^d$.

Hang on! The hypothesis of the Master Theorem was the the extra work at each level was $O(n^d)$. That's NOT the same as work <= cn^d for some constant c.

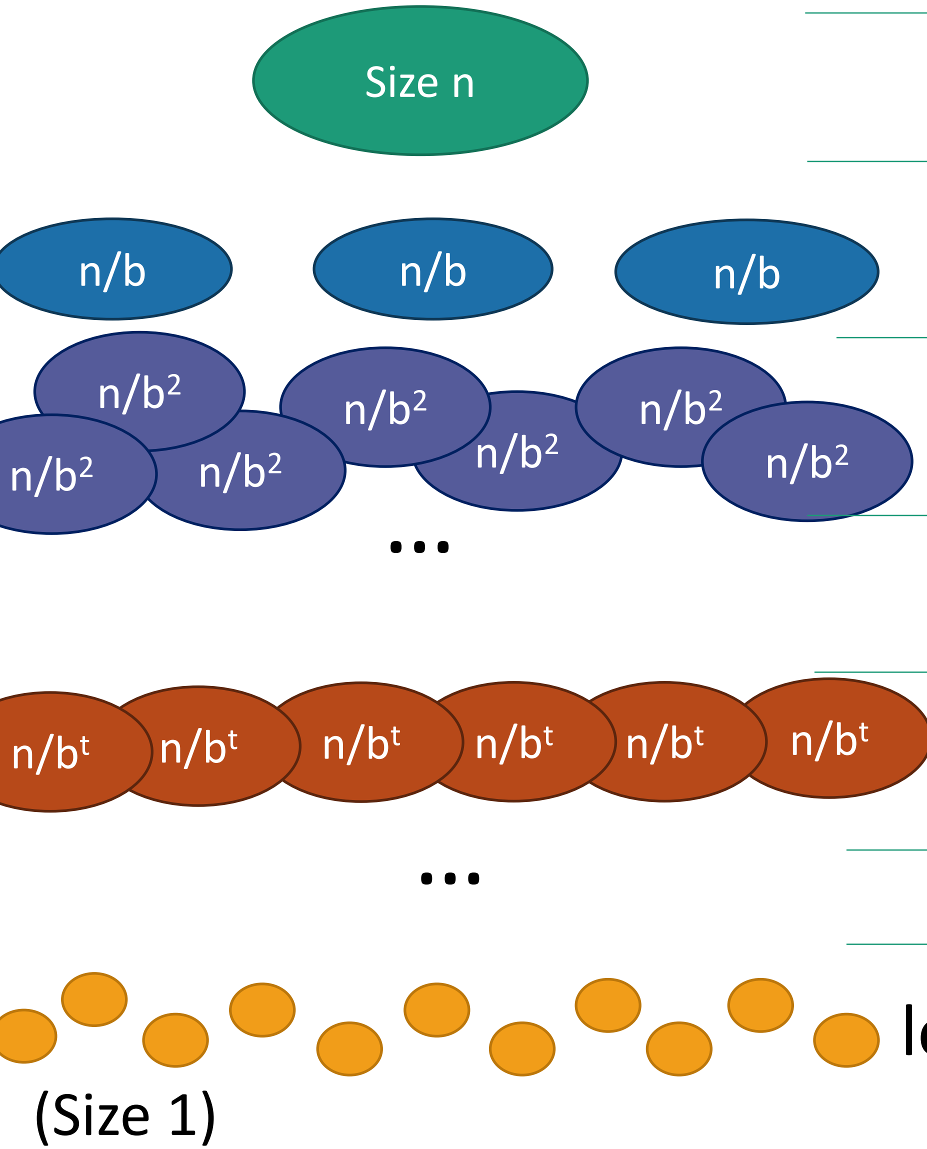
Plucky the Pedantic Penguin

That's true … we'll actually prove a weaker statement that uses this hypothesis instead of the hypothesis that $T(n) = a \cdot T\left(\frac{n}{b}\right) + O(n^d)$. It's a good exercise to make this proof work rigorously with the O() notation.

Siggi the Studious Stork

Recursion tree

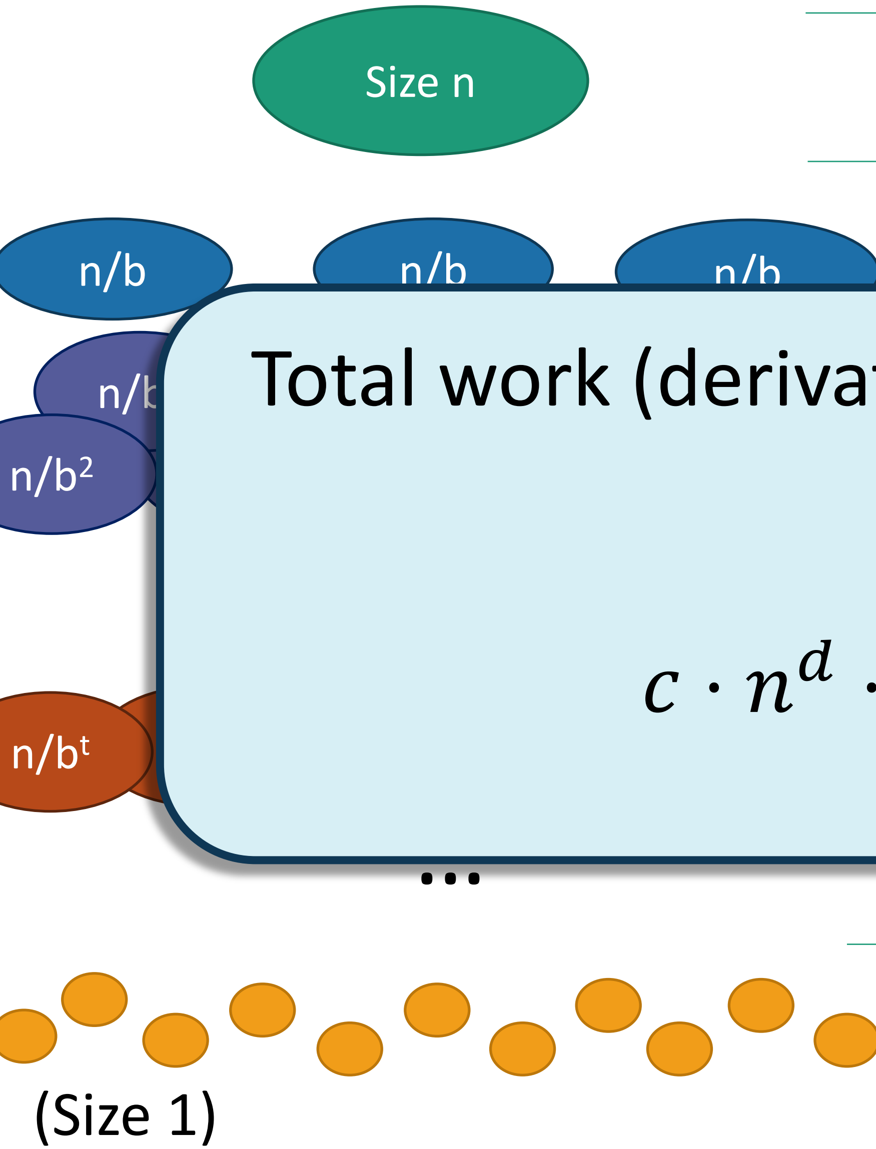
$$T(n) = a \cdot T\left(\frac{n}{b}\right) + c \cdot n^d$$



Level	# problems	Size of each problem	Amount of work at this level
0	1	n	$c \cdot n^d$
1	a	n/b	$ac\left(\frac{n}{b}\right)^d$
2	a^2	n/b^2	$a^2 c\left(\frac{n}{b^2}\right)^d$
t	a^t	n/b^t	$a^t c\left(\frac{n}{b^t}\right)^d$
…			
$\log_b(n)$	$a^{\log_b(n)}$	1	$a^{\log_b(n)} c$

Recursion tree

$T(n) = a \cdot T\left(\frac{n}{b}\right) + c \cdot n^d$



Level	# problems	Size of each problem	Amount of work at this level
0	1	n	$c \cdot n^d$
1	a	n/b	$ac\left(\frac{n}{b}\right)^d$
…			
$\log_b(n)$	$a^{\log_b(n)}$	1	$a^{\log_b(n)}c$

Total work (derivation on board) is at most:

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

Now let's check all the cases

(on board)

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

Even more generally, for T(n) = aT(n/b) + f(n)…

Theorem 3.2 (Master Theorem). *Let* $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$ *be a recurrence where* $a \geq 1$, $b > 1$. *Then,*

- *If* $f(n) = O\left(n^{\log_b a - \epsilon}\right)$ *for some constant* $\epsilon > 0$, $T(n) = \Theta\left(n^{\log_b a}\right)$.
- *If* $f(n) = \Theta\left(n^{\log_b a}\right)$, $T(n) = \Theta\left(n^{\log_b a} \log n\right)$.
- *If* $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$ *for some constant* $\epsilon > 0$ *and if* $af(n/b) \leq cf(n)$ *for* $c < 1$ *and all sufficiently large* n, *then* $T(n) = \Theta(f(n))$.

[From CLRS]

Understanding the Master Theorem

- Suppose $T(n) = a \cdot T\left(\frac{n}{b}\right) + O\left(n^d\right)$. Then

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

- What do these three cases mean?

The eternal struggle

Branching causes the number of problems to explode!
The most work is at the bottom of the tree!

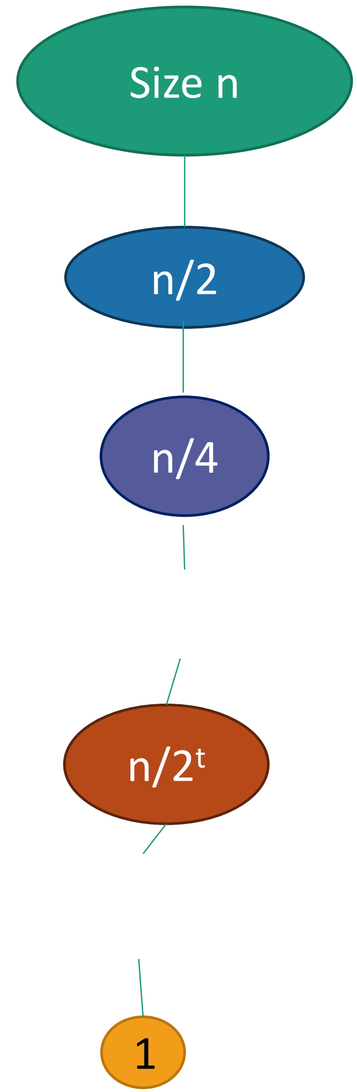
The problems lower in the tree are smaller!
The most work is at the top of the tree!

Consider our three warm-ups

1. $T(n) = T\left(\frac{n}{2}\right) + n$
2. $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n$
3. $T(n) = 4 \cdot T\left(\frac{n}{2}\right) + n$

First example: tall and skinny tree

1. $T(n) = T\left(\frac{n}{2}\right) + n, \qquad \left(a < b^d\right)$



- The amount of work done at the top (the biggest problem) swamps the amount of work done anywhere else.

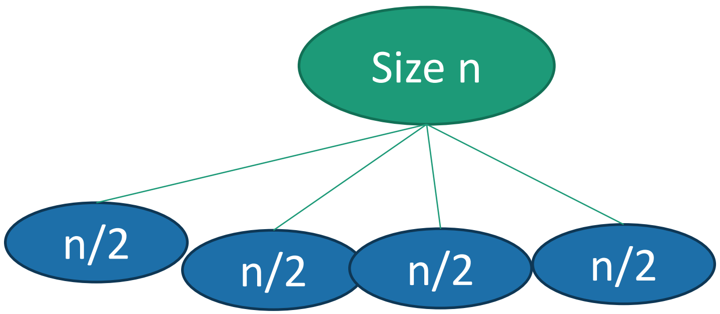
- T(n) = O(work at top) = O(n)

WINNER

Most work at the top of the tree!

Third example: bushy tree

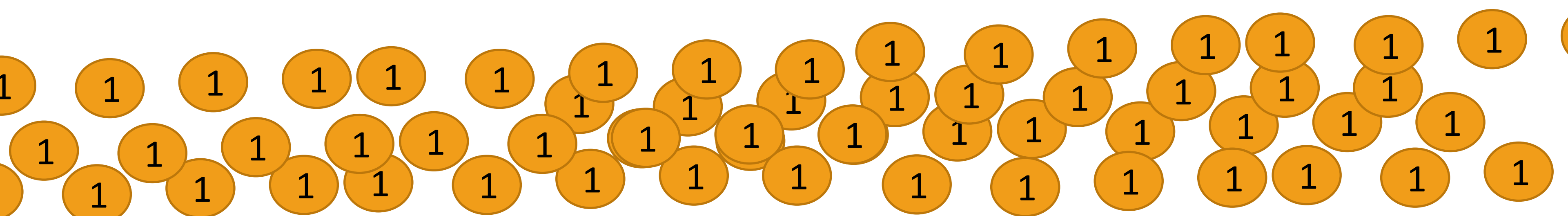
3. $T(n) = 4 \cdot T\left(\frac{n}{2}\right) + n, \qquad (a > b^d)$



WINNER

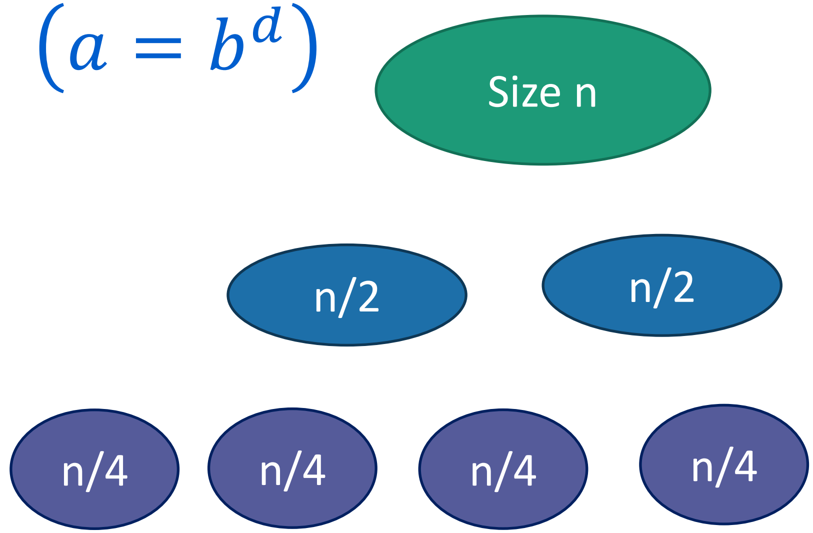
Most work at the bottom of the tree!

- There are a HUGE number of leaves, and the total work is dominated by the time to do work at these leaves.
- T(n) = O(work at bottom) = O($4^{\text{depth of tree}}$) = O(n^2)



Second example: just right

2. $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n, \qquad \left(a = b^d\right)$



- The branching **just** balances out the amount of work.
- The same amount of work is done at every level.
- T(n) = (number of levels) * (work per level)
- = log(n) * O(n) = O(nlog(n))

Recap

- The "Master Method" makes our lives easier.
- But it's basically just codifying a calculation we could do from scratch if we wanted to.

- What if the sub-problems are different sizes?
- And when might that happen?

- The Master Theorem only works when all sub-problems are the same size.
- **That's not always the case.**

- We'll use something called the **substitution method** instead.

The Plan

1. The **Substitution Method**
    - You got a sneak peak on your pre-lecture exercise
2. The **SELECT** problem.
3. The **SELECT** solution.
4. Return of the **Substitution Method.**

A non-tree method

For most of this lecture, division is integer division: $\frac{n}{2}$ means $\left\lfloor \frac{n}{2} \right\rfloor$.
As we noted last time we'll be pretty sloppy about the difference.

- Here's another way to solve:
  - $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n$
  - $T(0) = 0,\ T(1) = 1$

1. Guess what the answer is.
2. Formally prove that that's what the answer is.

1. Guess what the answer is.

- $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n$

- $T(n) = 2 \cdot \left(2 \cdot T\left(\frac{n}{4}\right) + \frac{n}{2}\right) + n$
- $T(n) = 4 \cdot T\left(\frac{n}{4}\right) + 2 \cdot n$

- $T(n) = 4 \cdot \left(2 \cdot T\left(\frac{n}{8}\right) + \frac{n}{4}\right) + 2 \cdot n$
- $T(n) = 8 \cdot T\left(\frac{n}{8}\right) + 3 \cdot n$

- Following the pattern…
- $\boldsymbol{T(n) = n \cdot T(1) + log(n) \cdot n = n(log(n) + 1\ )}$

So that is our guess!

We'll go fast through these computations because you all did it on your pre-lecture exercise!

- Inductive hypothesis:
  - $\boldsymbol{T(k) \leq k(log(k) + 1\,)}$ for all $1 \leq \mathrm{k} \leq \mathrm{n}$
- Base case:
  - $\boldsymbol{T(1) = 1 = 1(log(1) + 1\,)}$
- Inductive step:
  - $\boldsymbol{T(n)} = 2 \cdot T\left(\frac{n}{2}\right) + n$

$$\leq 2\left(\frac{n}{2}\left(\log\left(\frac{n}{2}\right) + 1\,\right)\right) + n$$

$$= 2\left(\frac{n}{2}(\log(n) - 1 + 1\,)\right) + n$$

$$= 2\left(\frac{n}{2}\log(n)\right) + n$$

$$= \boldsymbol{n(\log(n) + 1\,)}$$

What happened between these two lines?

- Conclusion:
  - By induction, $\boldsymbol{T(n) = n(log(n) + 1\,)}$ for all n > 0.

That's called the
substitution method

- So far, just seems like a different way of doing the same thing.

- But consider this!

$$T(n) = 3n + T\left(\frac{n}{5}\right) + T\left(\frac{n}{2}\right)$$

$$T(n) = 10n \text{ when } 1 \leq n \leq 10$$

Gross!

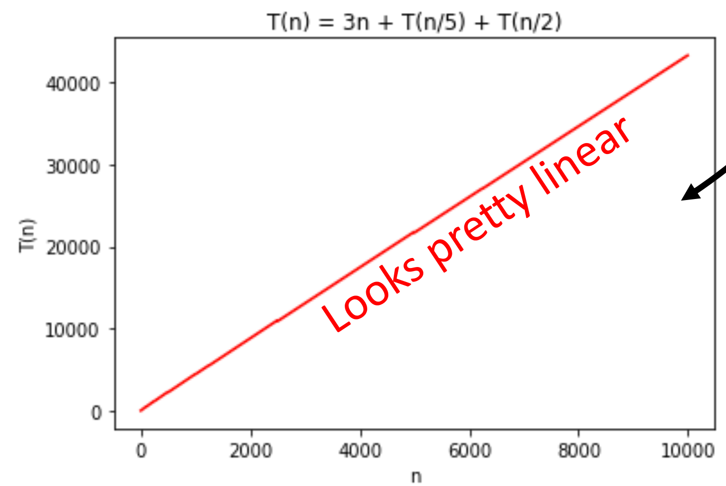
Step 1: guess what the answer is

$$T(n) = 3n + T\left(\frac{n}{5}\right) + T\left(\frac{n}{2}\right)$$

$$T(n) = 10n \text{ when } 1 \leq n \leq 10$$

- Let's try the same unwinding thing to get a feel for it.

- Okay, that gets gross fast. We can also just try it out.



- What else do we know?:
  - $T(n) \leq 3n + T\left(\frac{n}{5}\right) + T\left(\frac{n}{2}\right)$
    $\leq 3n + 2 \cdot T\left(\frac{n}{2}\right)$
    $= O(n\log(n))$
  - $T(n) \geq 3n$
  - So the right answer is somewhere between O(n) and O(nlog(n))…

$$T(n) = 3n + T\left(\frac{n}{5}\right) + T\left(\frac{n}{2}\right)$$

$$T(n) = 10n \text{ when } 1 \le n \le 10$$

Let's guess O(n)

Step 2: prove our guess is right

- Inductive Hypothesis: $\boldsymbol{T(k) \le Ck}$ for all $1 \le k < n$.
- Base case: $\boldsymbol{T(k) \le Ck}$ for all $\mathrm{k} \le 10$
- Inductive step:
  - $T(n) = 3n + T\left(\frac{n}{5}\right) + T\left(\frac{n}{2}\right)$

$$\le 3n + \boldsymbol{C}\left(\frac{n}{5}\right) + \boldsymbol{C}\left(\frac{n}{2}\right)$$

$$= 3n + \frac{\boldsymbol{C}}{5}n + \frac{\boldsymbol{C}}{2}n$$

$$\le \boldsymbol{C}n \text{ ??}$$

- Conclusion:
  - There is some $\boldsymbol{C}$ so that for all $n \ge 1$, $\boldsymbol{T(n) \le Cn}$
  - Aka, T(n) = O(n).

C is some constant we'll have to fill in later!

Whatever we choose C to be, it should have C≥10

Let's solve for C and make this true!
C = 10 works.

Now pretend like we knew it all along.

$$T(n) = 3n + T\left(\frac{n}{5}\right) + T\left(\frac{n}{2}\right)$$

$$T(n) = 10n \text{ when } 1 \leq n \leq 10$$

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

Proof:

- Inductive Hypothesis: $\boldsymbol{T(k) \leq 10k}$ for all k < n.
- Base case: $\boldsymbol{T(k) \leq 10k}$ for all k $\leq$ 10
- Inductive step:
  - $T(n) = 3n + T\left(\frac{n}{5}\right) + T\left(\frac{n}{2}\right)$
  - $T(n) \leq 3n + \mathbf{10}\left(\frac{n}{5}\right) + \mathbf{10}\left(\frac{n}{2}\right)$
  - $T(n) \leq 3n + 2n + 5n = 10n.$
- Conclusion:
  - For all $n \geq 1$, $\boldsymbol{T(n) \leq 10n}$, aka T(n) = O(n).

What have we learned?

- The substitution method can work when the master theorem doesn’t.
  - For example with different-sized sub-problems.

- Step 1: generate a guess
  - Throw the kitchen sink at it.
- Step 2: try to prove that your guess is correct
  - You may have to leave some constants unspecified till the end – then see what they need to be for the proof to work!!
- Step 3: profit
  - Pretend you didn’t do Steps 1 and 2 and write down a nice proof.

The Plan

1. The **Substitution Method**
   - You got a sneak peak on your pre-lecture exercise
2. The **SELECT** problem.
3. The **SELECT** solution.
4. Return of the **Substitution Method.**

The problem we will solve

A is an array of size n, k is in {1,…,n}

- SELECT(A, k):
  - Return the k'th smallest element of A.

For today, assume all arrays have distinct elements.

7	4	3	8	1	5	9	14

- SELECT(A, 1) = 1
- SELECT(A, 2) = 3
- SELECT(A, 3) = 4
- SELECT(A, 8) = 14

- SELECT(A, 1) = MIN(A)
- SELECT(A, n/2) = MEDIAN(A)
- SELECT(A, n) = MAX(A)

Being sloppy about floors and ceilings!

Note that the definition of Select is 1-indexed…

We’re gonna do it in time O(n)

- Let’s start with MIN(A) aka SELECT(A, 1).
- MIN(A):
  - ret = ∞
  - **For** i=0, ..., n-1:
    - If A[i] < ret:
      - ret = A[i]
  - **Return** ret

This stuff is O(1)

This loop runs O(n) times

- Time O(n). Yay!

How about SELECT(A,2)?

- SELECT(A,2):
  - ret = ∞
  - minSoFar = ∞
  - **For** i=0, .., n-1:
    - **If** A[i] < ret and A[i] < minSoFar:
      - ret = minSoFar
      - minSoFar = A[i]
    - **Else** if A[i] < ret and A[i] >= minSoFar:
      - ret = A[i]
  - **Return** ret

(The actual algorithm here is not very important because this won't end up being a very good idea…)

Still O(n)

SO FAR SO GOOD.

SELECT(A, n/2) aka MEDIAN(A)?

- MEDIAN(A):
  - ret = ∞
  - minSoFar = ∞
  - secondMinSoFar = ∞
  - thirdMinSoFar = ∞
  - fourthMinSoFar = ∞
  - ….

- This is not a good idea for large k (like n/2 or n).
- Basically this is just going to turn into something like INSERTIONSORT…and that was $O(n^2)$.

A much better idea for large k

- SELECT(A, k):
  - A = MergeSort(A)
  - **return** A[k-1]

It's k-1 and not k since my pseudocode is 0-indexed and the problem is 1-indexed…

- Running time is O(n log(n)).
- So that's the benchmark….

Can we do better?

We're hoping to get O(n)

The Plan

1. The **Substitution Method**
   - You got a sneak peak on your pre-lecture exercise
2. The **SELECT** problem.
3. The **SELECT** solution.
4. Return of the **Substitution Method.**

Idea: divide and conquer!

Say we want to find SELECT(A, k)

9	8	3	6	1	4	2

How about this pivot?

First, pick a "pivot." We'll see how to do this later.

This PARTITION step takes time O(n). (Notice that we don't sort each half).

Next, partition the array into "bigger than 6" or "less than 6"

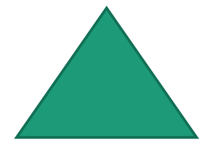
L = array with things smaller than A[pivot]

R = array with things larger than A[pivot]

Idea: divide and conquer!

Say we want to find SELECT(A, k)

6



How about this pivot?

First, pick a "pivot." We'll see how to do this later.

This PARTITION step takes time O(n). (Notice that we don't sort each half).

Next, partition the array into "bigger than 6" or "less than 6"

3	1	4	2

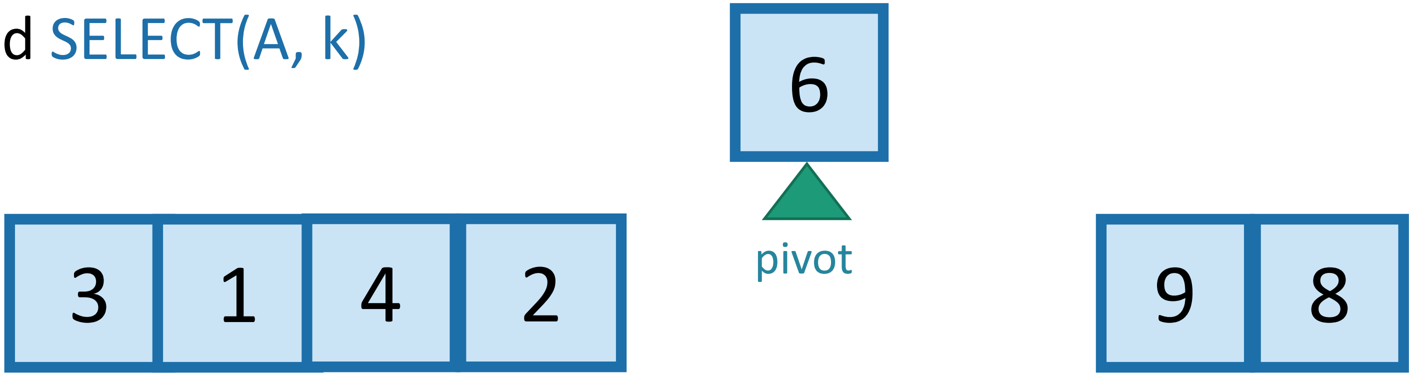
L = array with things smaller than A[pivot]

9	8

R = array with things larger than A[pivot]

Idea continued…

Say we want to find SELECT(A, k)



L = array with things smaller than A[pivot]

R = array with things larger than A[pivot]

- If k = 5 = len(L) + 1:
  - We should return A[pivot]
- If k < 5:
  - We should return SELECT(L, k)
- If k > 5:
  - We should return SELECT(R, k – 5)

This suggests a recursive algorithm

(still need to figure out how to pick the pivot…)

Pseudocode

- **getPivot**(A) returns some pivot for us.
  - How?? We'll see later…
- **Partition**(A,p) splits up A into L, A[p], R.

- **Select**(A,k):
  - **If** len(A) <= 50:
    - A = **MergeSort**(A)
    - **Return** A[k-1]
  - p = **getPivot**(A)
  - L, pivotVal, R = **Partition**(A,p)
  - **if** len(L) == k-1:
    - return pivotVal
  - **Else if** len(L) > k-1:
    - return **Select**(L, k)
  - **Else if** len(L) < k-1:
    - return **Select**(R, k – len(L) – 1)

Base Case: If the len(A) = O(1), then any sorting algorithm runs in time O(1).

Case 1: We got lucky and found exactly the k'th smallest value!

Case 2: The k'th smallest value is in the first part of the list

Case 3: The k'th smallest value is in the second part of the list

What is the running time?

- $T(n) = \begin{cases} T(\mathbf{len(L)}) + O(n) & \mathbf{len(L)} > k - 1 \\ T(\mathbf{len(R)}) + O(n) & \mathbf{len(L)} < k - 1 \\ O(n) & \mathbf{len(L)} = k - 1 \end{cases}$

- What are **len(L)** and **len(R)**?
  - That depends on how we pick the pivot…
  - What do we hope happens?
  - What do we hope doesn't happen?

In an ideal world*…

- We split the input in half:
  - **len(L) = len(R) = (n-1)/2**
- Let's use the **Master Theorem**!
  - $T(n) \leq T\left(\frac{n}{2}\right) + O(n)$
  - So a = 1, b = 2, d = 1
  - $T(n) \leq O\left(n^d\right) = O(n)$

Apply here, the Master Theorem does NOT. Making unsubstantiated assumptions about problem sizes, we are.

Jedi master Yoda

- Suppose $T(n) = a \cdot T\left(\frac{n}{b}\right) + O\left(n^d\right)$. Then

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

*Okay, really ideal would be that we always pick the pivot so that len(L) = k-1. But say we don't have control over k, just over how we pick the pivot.

Question

- ***How do we pick a good pivot?***

- Randomly?
  - That works well if there's no bad guy.
  - But if there is a bad guy who gets to see our pivot choices, that's just as bad as the worst-case pivot.

Aside:

- In practice, there is often no bad guy. In that case, just pick a random pivot and it works really well!

But for today

- Let’s assume there’s this bad guy.
- We’ll get a **stronger guarantee**
- We’ll get to see a **really clever algorithm**
- And we’ll get more practice with the **substitution method.**

The Plan

1. The **Substitution Method**
   - You got a sneak peak on your pre-lecture exercise
2. The **SELECT** problem.
3. The **SELECT** solution.
   a) The outline of the algorithm.
   b) How to pick the pivot.
4. Return of the **Substitution Method.**

How should we pick the pivot?

- We’d like to live in the ideal world.

- Pick the pivot to divide the input in half!
- Aka, pick the median!
- Aka, pick **`Select`**`(A,n/2)`

How should we pick the pivot?

- We'd like to **approximate** the ideal world.

- Pick the pivot to divide the input **about** in half!
- Maybe this is easier!

In an ~~ideal~~ okay world…

Apply here, the Master Theorem STILL does NOT. (Since we don't know that we can do this – and if we could how long would it take?).

Jedi master Yoda

But at least it gives us a goal!

Lucky the lackadaisical lemur

- We split the input not quite in half:
  - **3n/10 < len(L) < 7n/10**
  - **3n/10 < len(R) < 7n/10**

- If we could do that, the **Master Theorem** would say:
  - $T(n) \leq T\left(\frac{7n}{10}\right) + O(n)$
  - So a = 1, b = 10/7, d = 1
  - $T(n) \leq O\left(n^d\right) = O(n)$

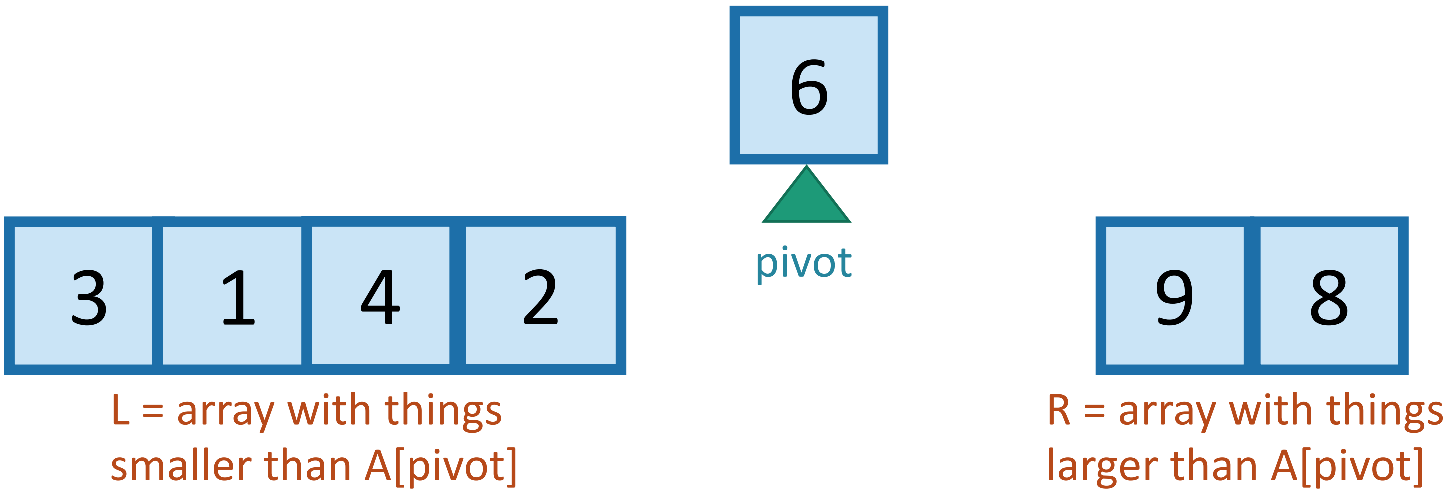
STILL GOOD!

- Suppose $T(n) = a \cdot T\left(\frac{n}{b}\right) + O\left(n^d\right)$. Then

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

Goal

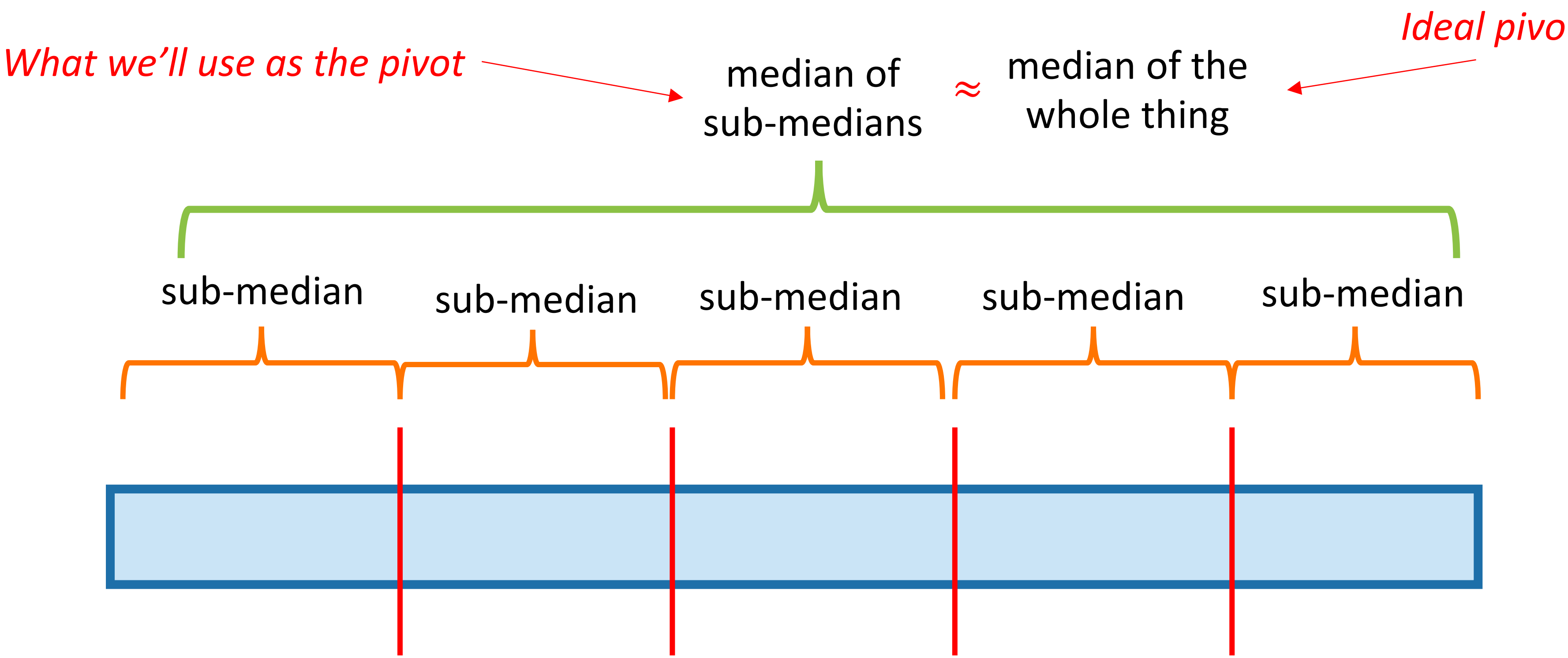
- Pick the pivot so that



$$\frac{3n}{10} < \text{len}(L) < \frac{7n}{10} \qquad \frac{3n}{10} < \text{len}(R) < \frac{7n}{10}$$

Another divide-and-conquer alg!

- We can't solve `Select(A,n/2)` (yet)
- But we can **divide and conquer** and solve `Select(B,m/2)` for smaller values of m (where len(B) = m).
- **Lemma*:** The median of sub-medians is close to the median.



How to pick the pivot

- CHOOSEPIVOT(A):
  - Split A into m = $\left\lceil \frac{n}{5} \right\rceil$ groups, of size <=5 each.
  - **For** i=1, .., m:
    - Find the median within the i'th group, call it p_i
  - p = SELECT([p_1, p_2, p_3, …, p_m] , m/2)
  - **return** p

This takes time O(1), for each group, since each group has size 5. So that's O(m) total in the for loop.

8

4

1	8	9	3	15	5	9	1	3	4	12	2	1	5	20	15	13	2	4	6	12	1	15	22	3

6

Pivot is SELECT(8 4 5 6 12 , 3) = 6:

12

6

1	8	9	3	15	5	9	1	3	4	12	2	1	5	20	15	13	2	4	6	12	1	15	22	3

PARTITION around that 6:

1	3	5	1	3	4	2	1	2	4	1	3	5	6	8	9	15	9	12	20	15	13	12	15	22

This part is L

This part is R: it's almost the same size as L.

CLAIM: this works

divides the array *approximately* in half

Lemma: If we choose the pivots like this, then

$$|L| \leq \frac{7n}{10} + 5$$

and

$$|R| \leq \frac{7n}{10} + 5$$

How about the running time?

- Suppose the Lemma is true. (It is).
  - $|L| \leq \frac{7n}{10} + 5$ and $|R| \leq \frac{7n}{10} + 5$
- Recurrence relation:

$$T(n) \leq ?$$

Pseudocode

- **getPivot**(A) returns some pivot for us.
  - How?? We'll see later…
- **Partition**(A,p) splits up A into L, A[p], R.

- **Select**(A,k):
  - **If** len(A) <= 50:
    - A = **MergeSort**(A)
    - **Return** A[k-1]
  - p = **getPivot**(A)
  - L, pivotVal, R = **Partition**(A,p)
  - **if** len(L) == k-1:
    - return pivotVal
  - **Else if** len(L) > k-1:
    - return **Select**(L, k)
  - **Else if** len(L) < k-1:
    - return **Select**(R, k – len(L) – 1)

Base Case: If the len(A) = O(1), then any sorting algorithm runs in time O(1).

Case 1: We got lucky and found exactly the k'th smallest value!

Case 2: The k'th smallest value is in the first part of the list

Case 3: The k'th smallest value is in the second part of the list

How about the running time?

- Suppose the Lemma is true. (It is).
  - $|L| \leq \frac{7n}{10} + 5$ and $|R| \leq \frac{7n}{10} + 5$
- Recurrence relation:

$$T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + O(n)$$

The Plan

1. The **Substitution Method**
   - You got a sneak peak on your pre-lecture exercise
2. The **SELECT** problem.
3. The **SELECT** solution.
   a) The outline of the algorithm.
   b) How to pick the pivot.
4. Return of the **Substitution Method.**

This sounds like a job for…

The Substitution Method!

Step 1: generate a guess
Step 2: try to prove that your guess is correct
Step 3: profit

$$T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + O(n)$$

Like we did last time, treat this O(n) as cn for our analysis.
(For simplicity in class – to be rigorous we should use the formal definition!)

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

Recap

- The substitution method is another way to solve recurrence relations.
  - Can work when the master theorem doesn’t!
- One place we needed it was for SELECT.
  - Which we can do in time O(n)!