6/22 Lecture Agenda

- Announcements
- Part 1-2: Divide (and conquer) to multiply
- Part 1-3: MergeSort

Announcements!

The power outage that forced Stanford University to cancel classes for a day had no end in sight Wednesday afternoon, after Pacific Gas & Electric Co. said it could not access an area where repairs are needed to fix an equipment failure due to a nearby grass fire.

SF Chronicle

- The power is out! (But you knew that)
  - We'll do Friday's normally-live Problem Session (1:30-2:30) online (in addition to the 7:30-8:30 online version)
  - If power is *still* going to be out on Monday, we'll switch to 2020-2021 style Zoom until it's better
  - If this drags on, we may bump some deadlines a bit, accordingly

- HW1 coming tonight (perhaps without autograders for coding problem)

More announcements!

- The course site (cs161.stanford.edu) is mostly complete, yay!

- Office hours start Thursday
  - If any CAs need to cancel / reschedule / relocate because of power issues, we'll let you know
  - You can find our Nooks via the link in the upper right of the course site

6/22 Lecture Agenda

- Announcements
- Part 1-2: Divide (and conquer) to multiply
- Part 1-3: MergeSort

WORLD 1-2

Divide (and Conquer) to Multiply

Divide and Conquer

Sorting & Randomization

Data Structures

Graph Search

Dynamic Programming

Greed & Flow

Special Topics

Multiplication

"What is this?" you shout. "The third grade?"

This is a perfectly good algorithm...

```
  1   1
  1   1
    1 2 3
x     4 5
---------
    6 1 5
  4 9 2 0
---------
  5 5 3 5
```

But this is CS161! Can we do better?

Integer Multiplication

n

  1233925720752752384623764283568364918374523856298
x 4562323582342395285623467235019130750135350013753

???

How fast is the grade-school multiplication algorithm?

(How many one-digit operations?)

Integer Multiplication

n

1233925720752752384623764283568364918374523856298

x 4562323582342395285623467235019130750135350013753

How fast is the grade-school multiplication algorithm?

(How many one-digit operations?)

- At most n^2 multiplications
- At most n^2 additions (for carries)
- Finally, add n different numbers of at most $2n$ digits

so: $O(n^2)$.

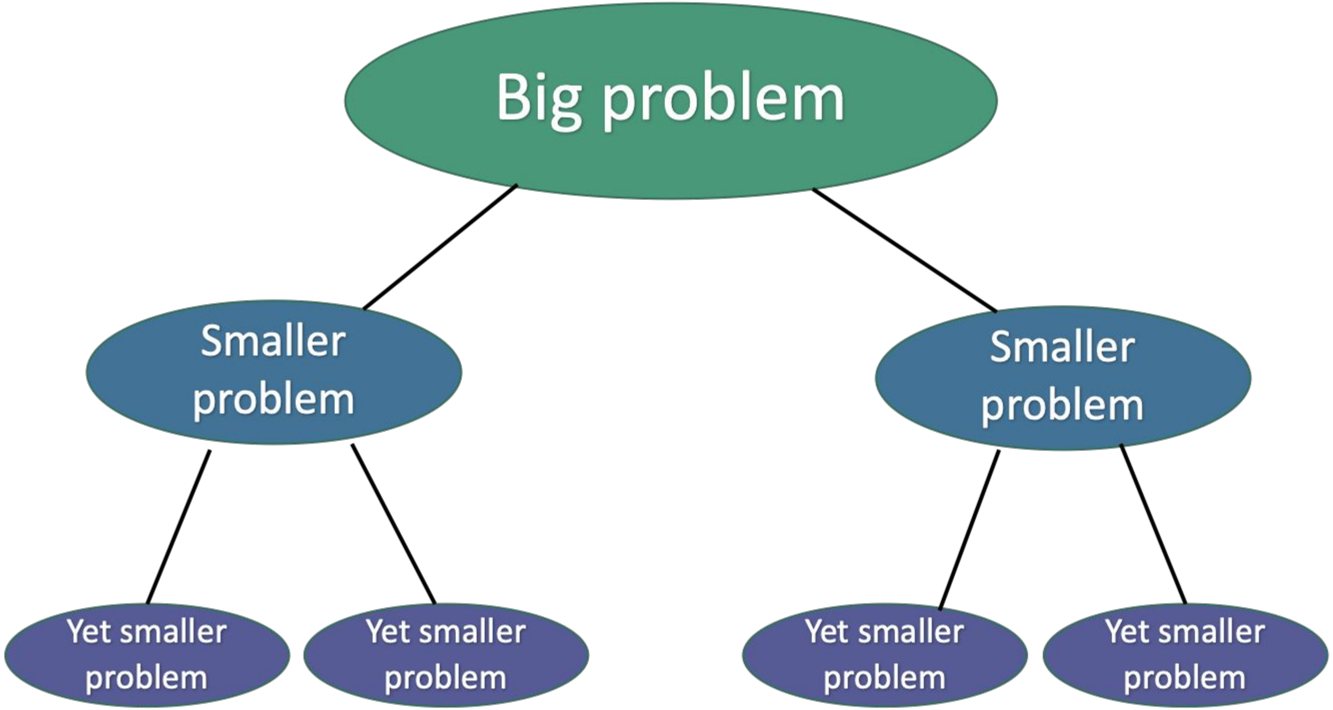
note: we reasonably treated single-digit math operations as $O(1)$

Can we do better?

- It's not obvious that we should expect to be able to!
- After all, don't we have to pair up each digit in the first number with each digit in the second number? And isn't that inherently $O(n^2)$?
- (Is $O(n^2)$ really so bad?
  - Think of trying to get a group of *n* people to all get along...
  - So, yes. Yes it is. At least for big enough *n*.)

Divide and conquer

Break problem up into smaller (easier) sub-problems



Divide and conquer for multiplication

Break up an integer:

$$1234 = 12 \times 100 + 34$$

$$1234 \times 5678$$

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

$$= (12 \times 56)10000 + (34 \times 56 + 12 \times 78)100 + (34 \times 78)$$

(1) 12×56 (2) 34×56 (3) 12×78 (4) 34×78

One 4-digit multiply → Four 2-digit multiplies

More generally

(Suppose n is even!)

Break up an n-digit integer:

$$[x_1 x_2 \cdots x_n] = [x_1 x_2 \cdots x_{n/2}] \times 10^{n/2} + [x_{n/2+1} x_{n/2+2} \cdots x_n]$$

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

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

One n-digit multiply → Four (n/2)-digit multiplies

Divide and conquer algorithm

not very precisely…

(Assume n is a power of 2…)

x,y are n-digit numbers

Multiply(x, y):

- **If** n=1:
  - **Return** xy

Base case: I've memorized my 1-digit multiplication tables…

- Write $x = a\,10^{\frac{n}{2}} + b$
- Write $y = c\,10^{\frac{n}{2}} + d$

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

- Recursively compute $ac,\ ad, bc, bd$:
  - ac = **Multiply**(a, c), etc..
- Add them up to get xy:
  - xy = ac 10^{n} + (ad + bc) $10^{n/2}$ + bd

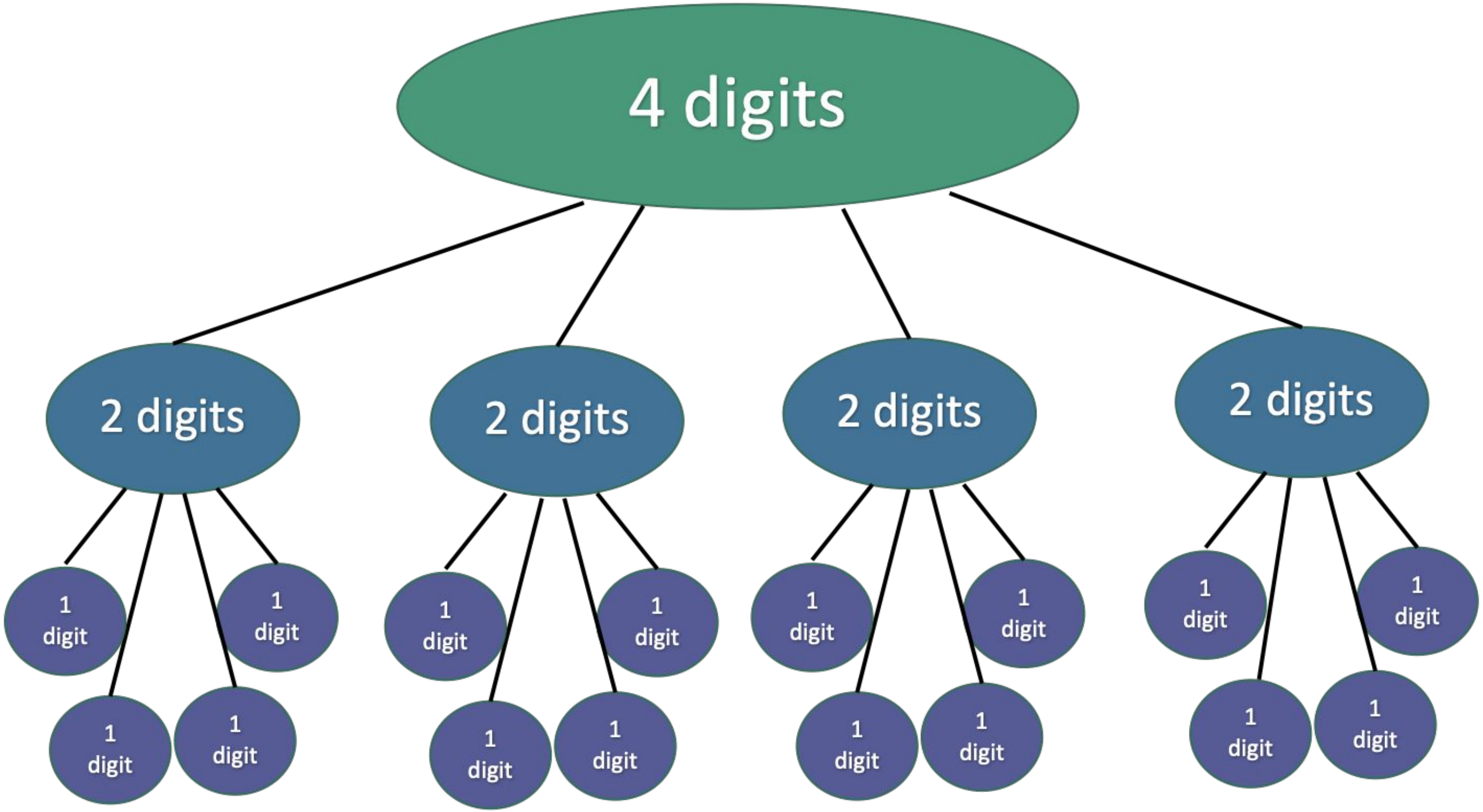
- We saw that this 4-digit multiplication problem broke up into four 2-digit multiplication problems

$$1234 \times 5678$$

- If you recurse on those 2-digit multiplication problems, how many 1-digit multiplications do you end up with total?

Recursion Tree

16 one-digit multiplies!

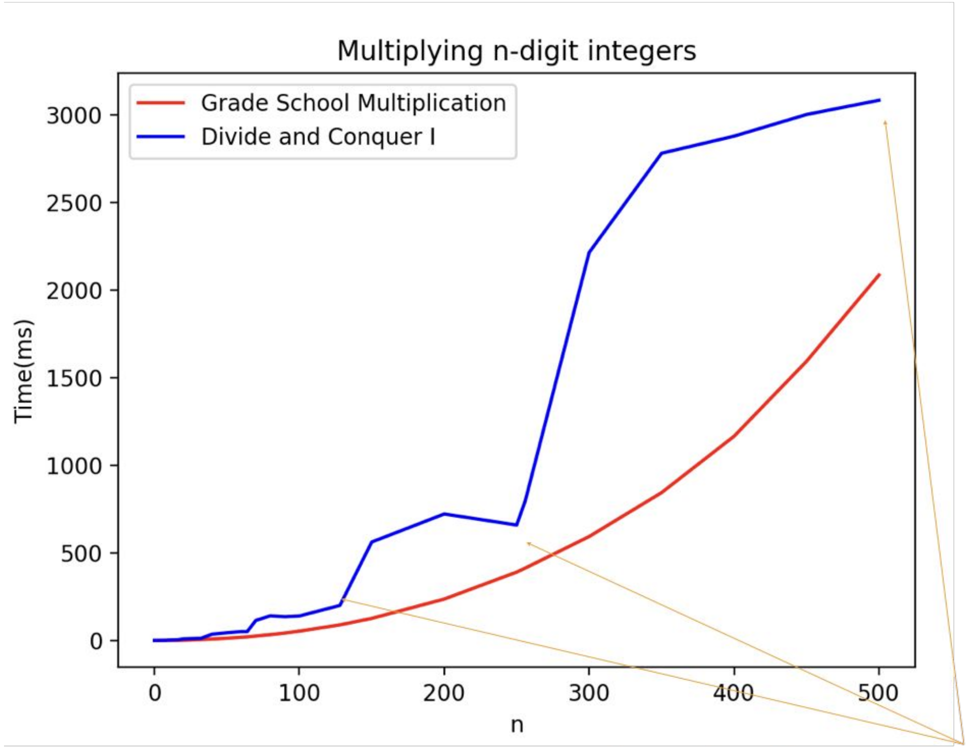


What is the running time?

- Better or worse than the grade school algorithm?

- How do we answer this question?
  1. Try it.
  2. Try to understand it analytically.

1. Try it.



Conjectures about running time?

Doesn't look too good but hard to tell…

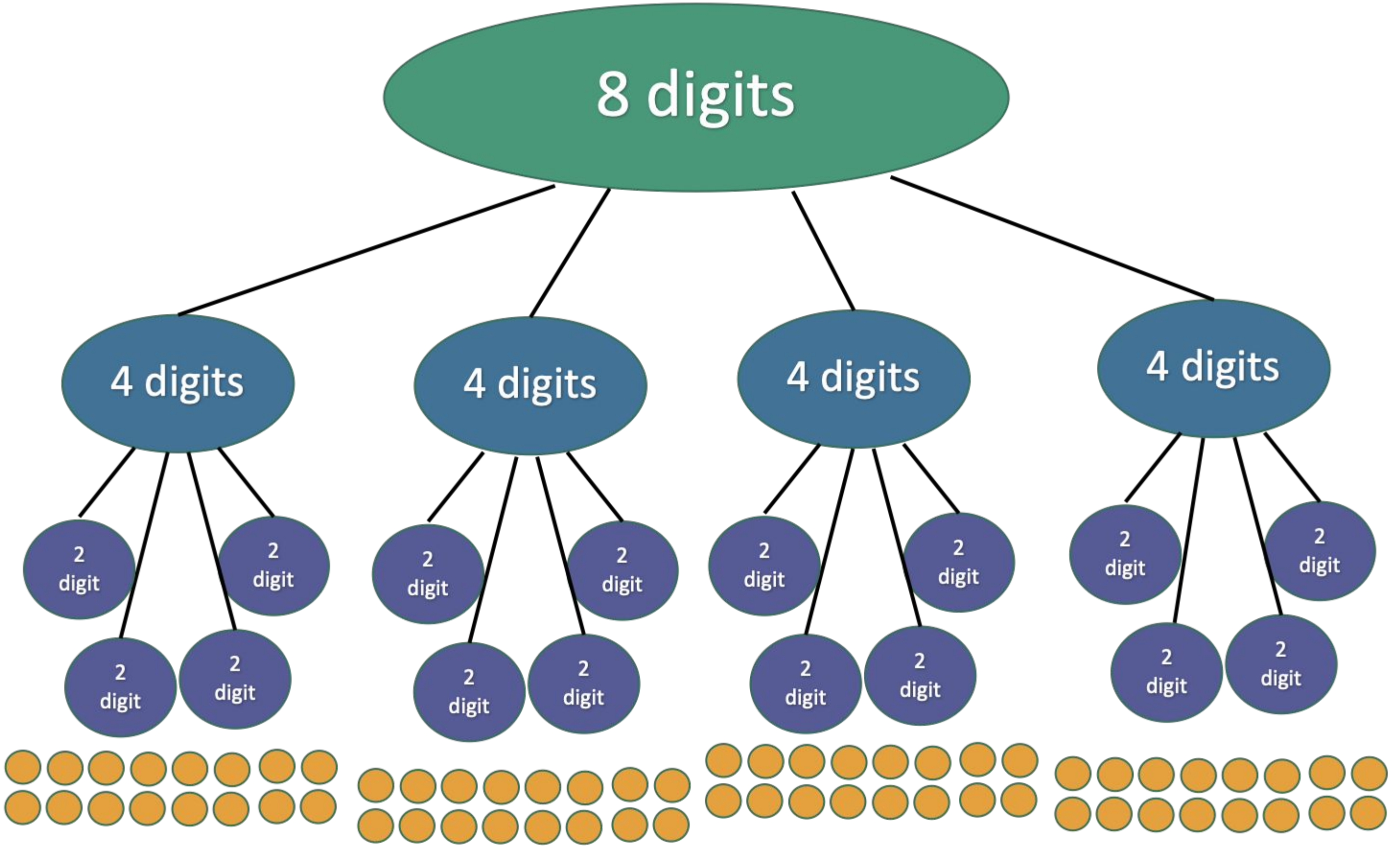
Maybe one implementation is slicker than the other?

Maybe if we were to run it to n=10000, things would look different.

Something funny is happening at powers of 2…

Recursion Tree

64 one-digit multiplies!



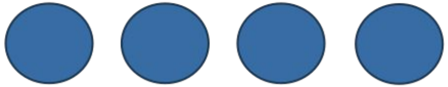
2. Try to understand the running time analytically

Claim:

The running time of this algorithm is AT LEAST n^2 operations.

There are n^2 1-digit problems

1 problem
of size n



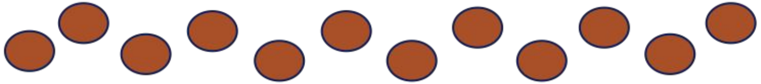
4 problems
of size n/2

…

4^t problems
of size $n/2^t$

Note: this is just a
cartoon – not going
to draw all 4^t circles!

…



n^2 problems
of size 1

- If you cut n in half $\log 2(n)$ times, you get down to 1.
- So at level
  $t = \log 2(n)$
  we get…

$4^{\log_2 n} = n^{\log_2 4} = n^2$
problems of size 1.

Well SHUCKS

- We tried out an awesome new strategy and it didn't do asymptotically better!
  - It didn't even seem to run faster than grade school multiplication!

- Guess we can pack up and go home. Thanks for coming to CS161!

- Or...

Divide and conquer can actually make progress

- Karatsuba figured out how to do this better!

$$xy = (a \cdot 10^{n/2} + b)(c \cdot 10^{n/2} + d)$$
$$= ac \cdot 10^n + (ad + bc)10^{n/2} + bd$$

Need these three things

- If only we could recurse on three things instead of four…

Karatsuba integer multiplication

- Recursively compute these THREE things:
  - ac
  - bd
  - (a+b)(c+d)

Subtract these off

get this

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

- Assemble the product:

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

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

How would this work?

(Still not super precise; still assume n is a power of 2.)

x,y are n-digit numbers

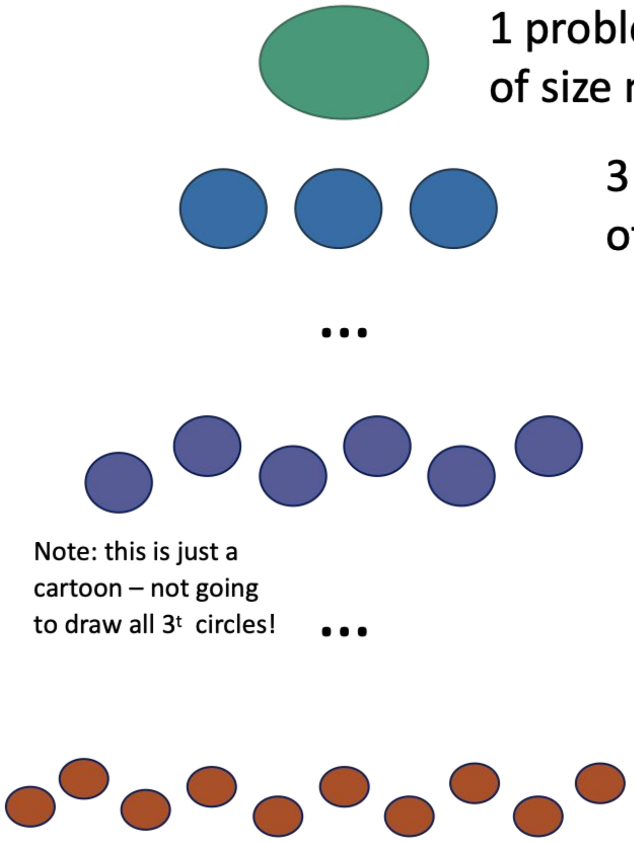
Multiply(x, y):

- **If** n=1:
  - **Return** xy
- Write $x = a\,10^{\frac{n}{2}} + b$ and $y = c\,10^{\frac{n}{2}} + d$

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

- ac = **Multiply**(a, c)
- bd = **Multiply**(b, d)
- z = **Multiply**(a+b, c+d)
- xy = ac 10^{n} + (z – ac - bd) $10^{n/2}$ + bd
- Return xy

What's the running time?



1 problem of size n

3 problems of size n/2

...

3^t problems of size $n/2^t$

...

$n^{1.6}$ problems of size 1

- If you cut n in half $\log 2(n)$ times, you get down to 1.
- So at level $t = \log 2(n)$ we get...

  $3^{\log_2 n} = n^{\log_2 3} \approx n^{1.6}$ problems of size 1.

We aren't accounting for the work at the higher levels! But we'll see later that this turns out to be okay.

Can we do better?

- **Toom-Cook** (1963): instead of breaking into three n/2-sized problems, break into five n/3-sized problems.
  - Runs in time $O\left(n^{1.465}\right)$

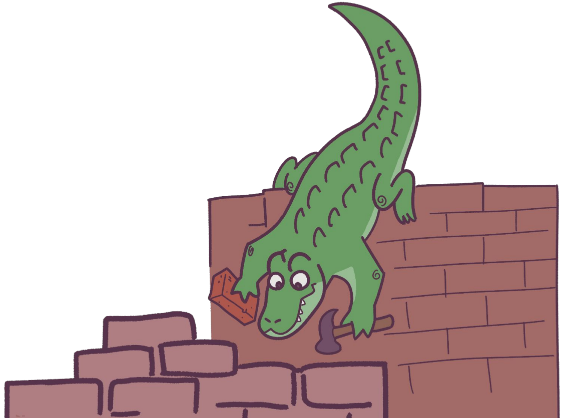
Optional: Try to figure out how to break up an n-sized problem into five n/3-sized problems! **(Hint: start with nine n/3-sized problems).**

Optional: Given that you can break an n-sized problem into five n/3-sized problems, where does the 1.465 come from?

- **Schönhage–Strassen** (1971):
  - Runs in time $O(n\log(n)\log\log(n))$
- **Furer** (2007)
  - Runs in time $n\log(n) \cdot 2^{O(\log^*(n))}$
- **Harvey and van der Hoeven** (2019)
  - Runs in time $O(n\log(n))$

[This is just for fun, you don't need to know these algorithms!]

But are these practical?



- From a talk by David Harvey, regarding their $O(n \log n)$ multiplication algorithm based on Discrete Fourier Transforms: (https://www.youtube.com/watch?v=FKGRc867j10)

For what n do we win?

The argument in our paper only kicks in for

$$n \geqslant 2^{1729^{12}} \approx 10^{214857091104455251940635045059417341952}.$$

This can probably be improved (a lot).

Isn't O(n log n) supposed to be better than O(n^2)?

- Remember that the definition only guarantees that there is *some* constant past which *some* multiple is an upper bound.

- **The very constant factors and multipliers that big-O ignores might be huge in practice!**

So are these algorithms useless then?

- Karatsuba isn't going to take the third grade by storm...
  - but it *is* useful in cryptography!
    - and cryptography lets you buy things on your phone!
  - Also, CPython (the most common implementation) uses it to multiply sufficiently large numbers!

- Even the $O(n \log n \log \log n)$ algorithm can be used in practice...
  - and remember that even it may have seemed useless once!

6/22 Lecture Agenda

- Announcements
- Part 1-2: Divide (and conquer) to multiply
- Part 1-3: MergeSort

WORLD 1-3

MergeSort

Divide and Conquer

Sorting & Randomization

Data Structures

Graph Search

Dynamic Programming

Greed & Flow

Special Topics

The sorting problem

Input: A list of comparable objects

- i.e., each object has a (not necessarily unique) value, and there is a way to compare any two values.

Output: A list of the same objects, but arranged such that their values are in nondecreasing order.

because of ties

Some flavors:

- Is the sort **stable**? (i.e. is it guaranteed that any two elements with the same value stay in the same relative order even after sorting?)
- Is the implementation in-place or does it make a new copy?

So many sorts of sorts!

We're not going to cover some, like bubble and selection, because IMO they're not really important. (Insertion will come up on a pre-HW or HW, maybe)

Each one we cover in CS161 will illustrate a different idea (kinda like how it's good to know different *types* of programming languages)

What do major languages actually use to sort?

C++: Introsort (a hybrid of Quicksort, Heapsort, and Insertion Sort)

Java: Quicksort (for primitives), a modified MergeSort (for objects)

JavaScript: Implementation-dependent

Python: Timsort (a hybrid of Insertion Sort and MergeSort)

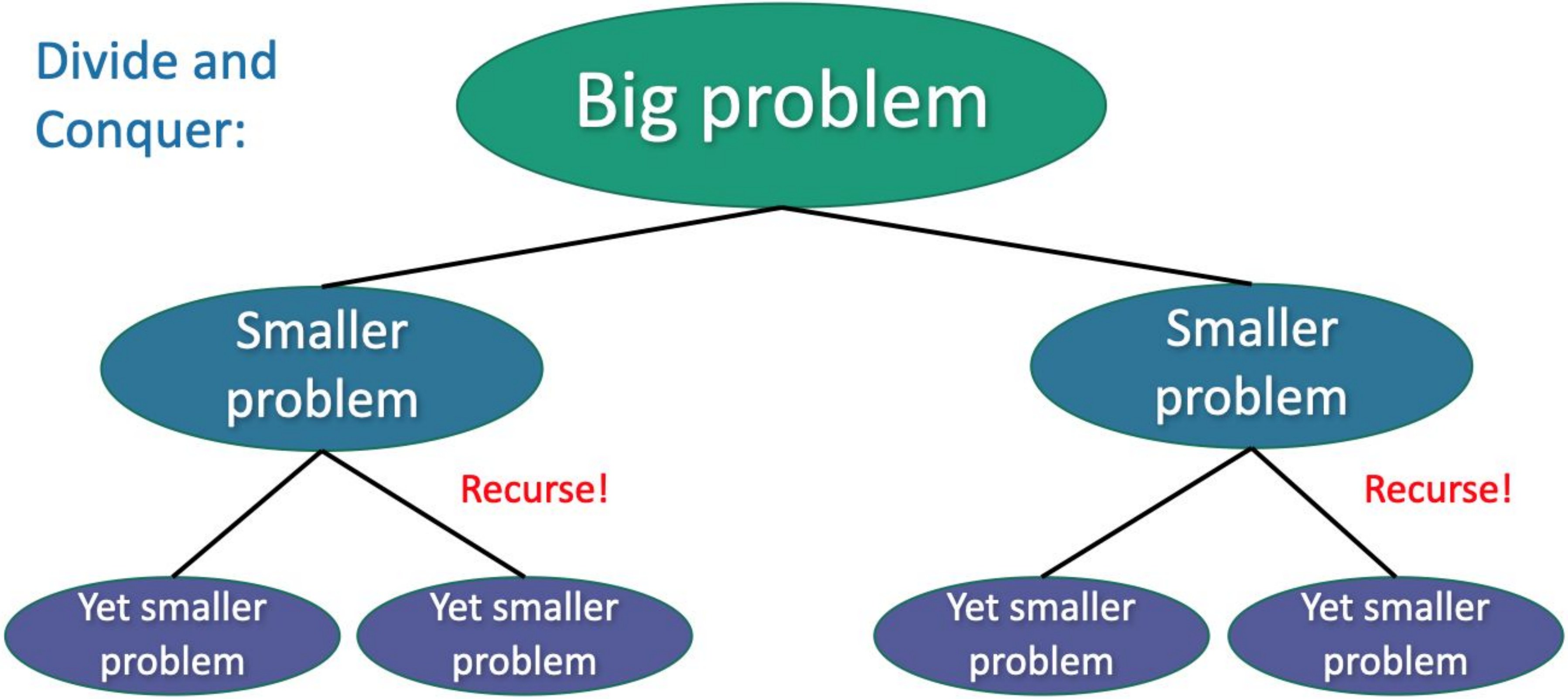
One takeaway from this: maybe there is no universally best sort?

Why MergeSort?

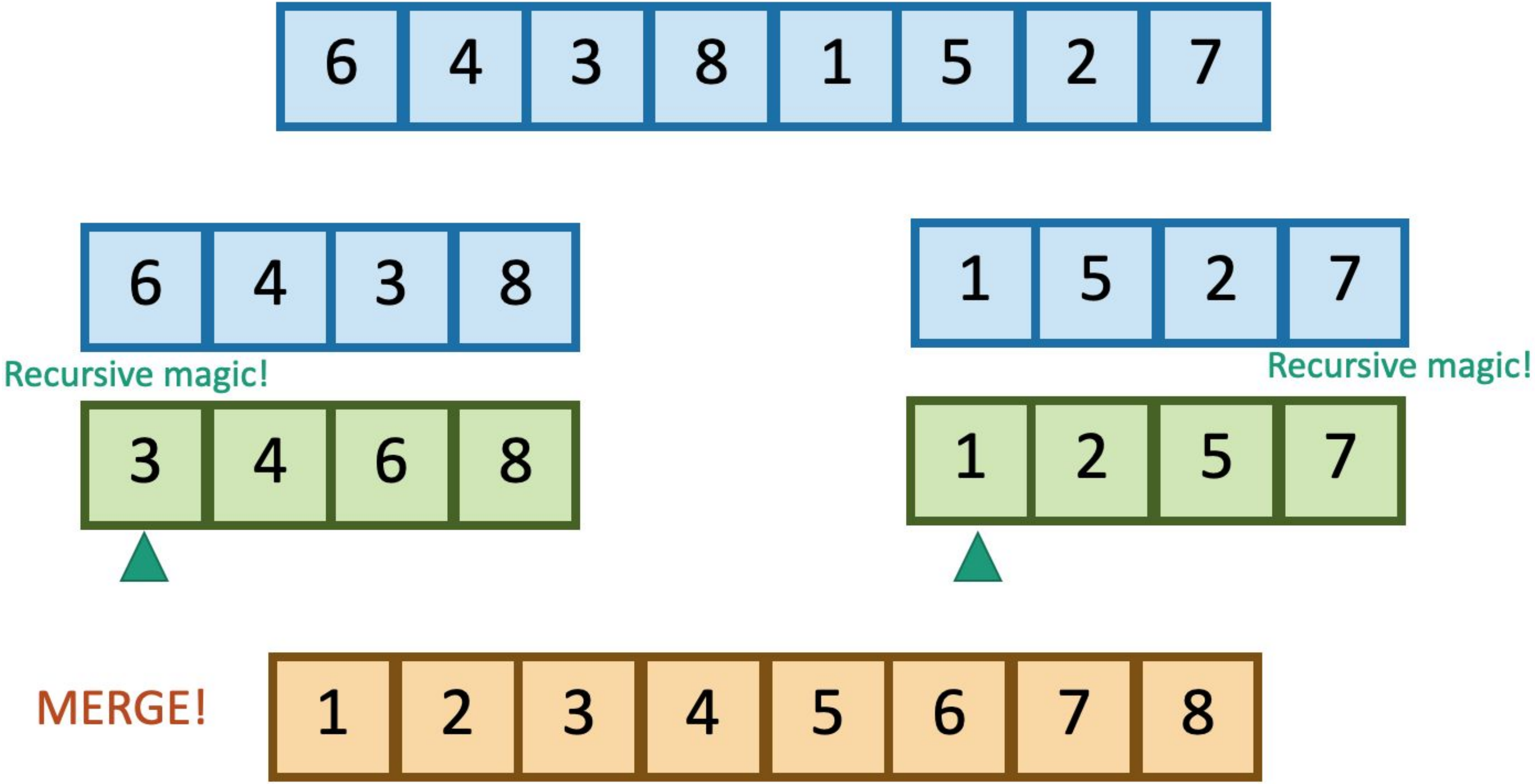
- It's asymptotically fast
- It's (IMO) one of the more beautiful sorts
- It includes a Merge step that is a powerful idea worth knowing about
- It illustrates "divide and conquer" well

- Recall from last time:

Divide and Conquer:



MergeSort

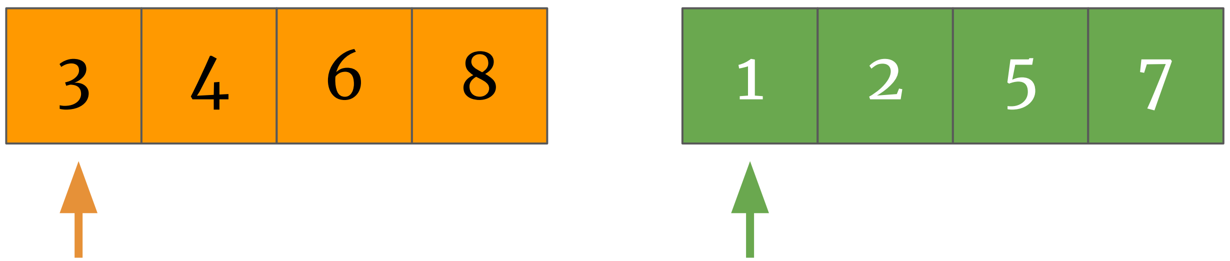

6 4 3 8 1 5 2 7
6 4 3 8
1 5 2 7
Recursive magic!
Recursive magic!
3 4 6 8
1 2 5 7
MERGE!
1 2 3 4 5 6 7 8

The Merge Step

- Input: Two sorted lists.

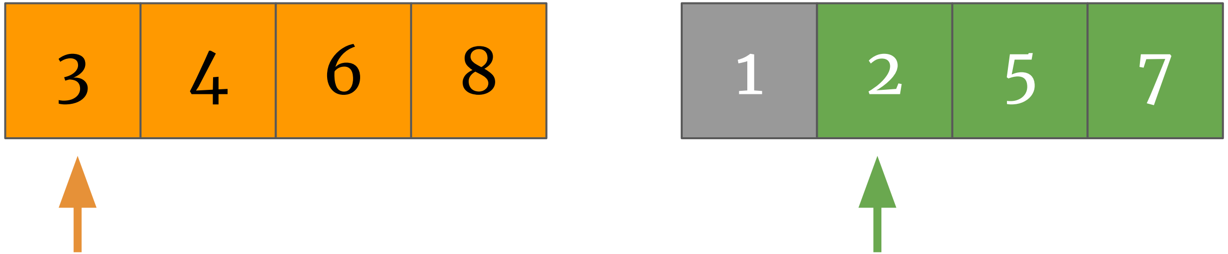
- Output: A single sorted list containing all the elements of the input lists.

Ex: Input [3, 4, 6, 8], [1, 2, 5, 7]
Output [1, 2, 3, 4, 5, 6, 7, 8]



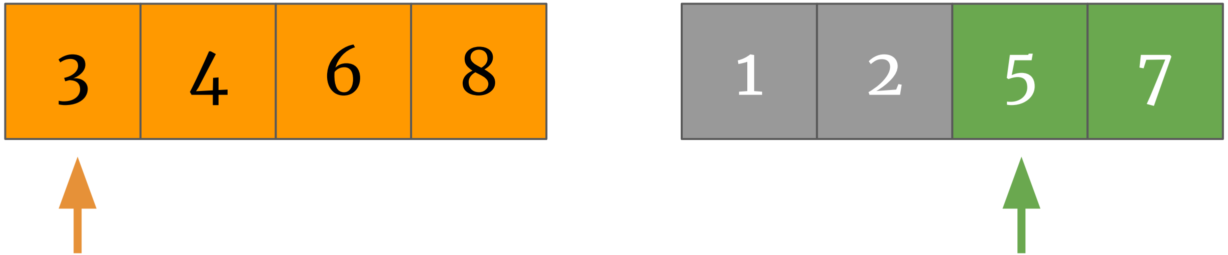
Repeatedly do the following:

- Check the elements pointed to by the two pointers.
- Add the smallest one to the new list. Advance that pointer.



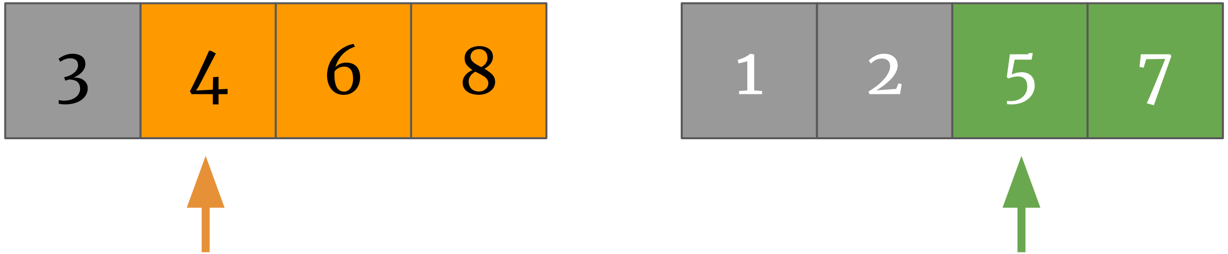
Repeatedly do the following:

- Check the elements pointed to by the two pointers.
- Add the smallest one to the new list. Advance that pointer.



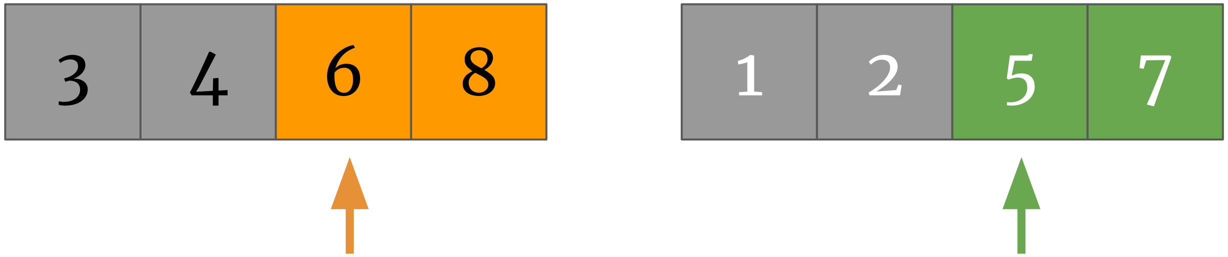
Repeatedly do the following:

- Check the elements pointed to by the two pointers.
- Add the smallest one to the new list. Advance that pointer.



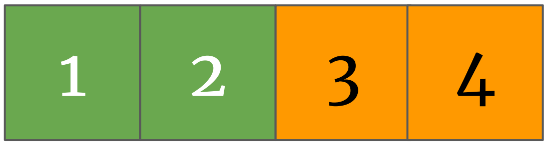
Repeatedly do the following:

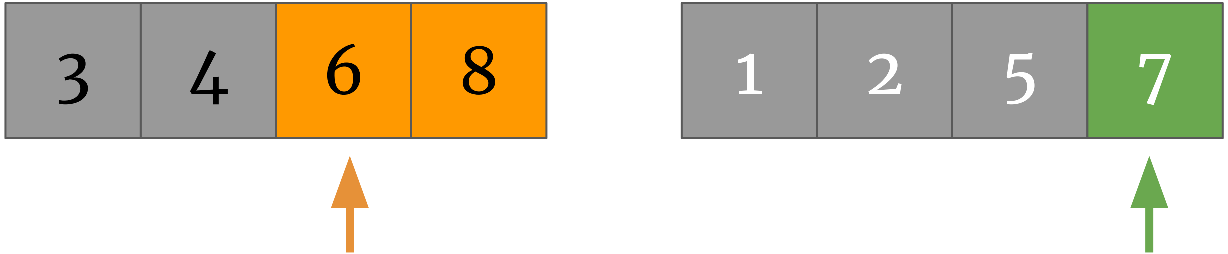
- Check the elements pointed to by the two pointers.
- Add the smallest one to the new list. Advance that pointer.



Repeatedly do the following:

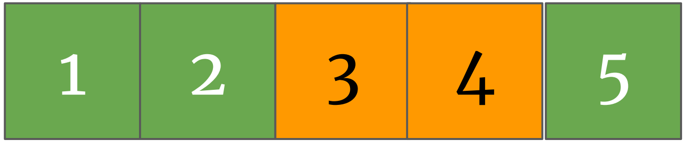
- Check the elements pointed to by the two pointers.
- Add the smallest one to the new list. Advance that pointer.

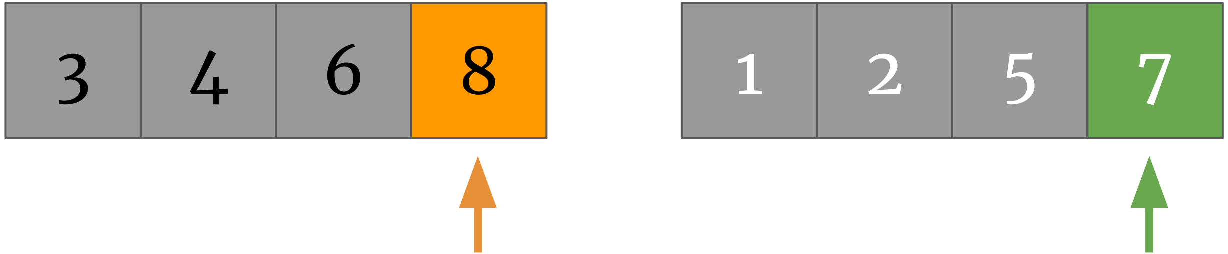




Repeatedly do the following:

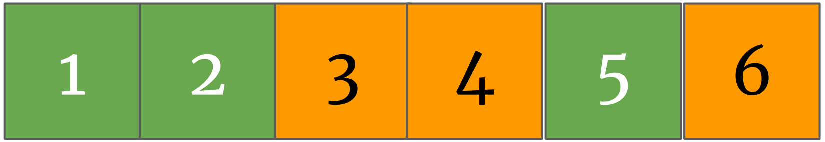
- Check the elements pointed to by the two pointers.
- Add the smallest one to the new list. Advance that pointer.

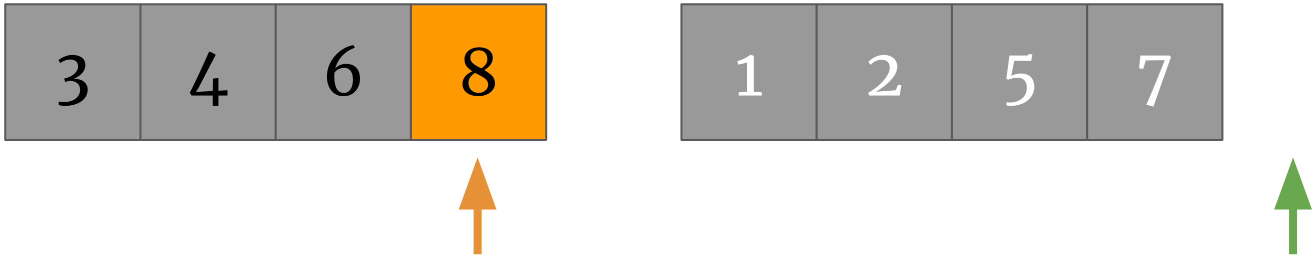




Repeatedly do the following:

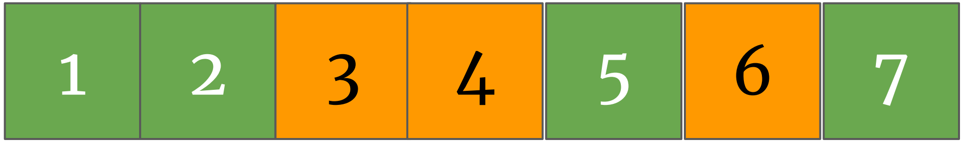
- Check the elements pointed to by the two pointers.
- Add the smallest one to the new list. Advance that pointer.

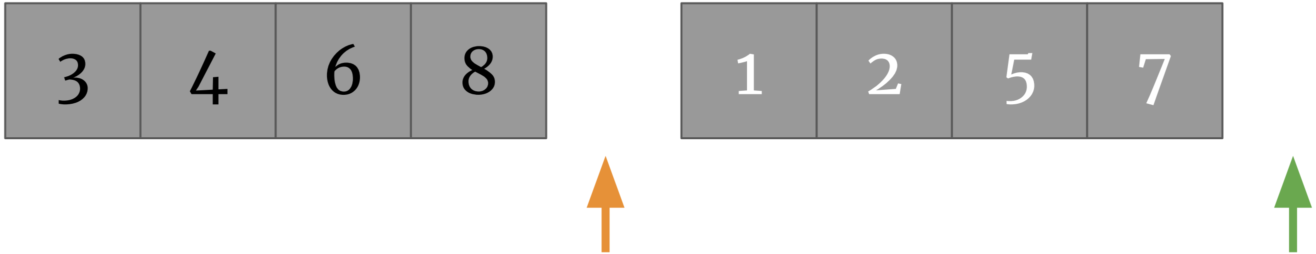




Repeatedly do the following:

- Check the elements pointed to by the two pointers.
- Add the smallest one to the new list. Advance that pointer.





Repeatedly do the following:

- Check the elements pointed to by the two pointers.
- Add the smallest one to the new list. Advance that pointer.

Key Ideas

- Only works because the two input lists are already sorted.
  - But what sorted them? A deeper call to MergeSort!

- Runs in O(n) time.
  - Intuitively, this is because each step moves one of the pointers ahead, and they can collectively move only $2n$ steps. And comparing two values takes constant time.

- Works fine with ties, but think of what you would need to do to ensure that the sort remains stable…

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])

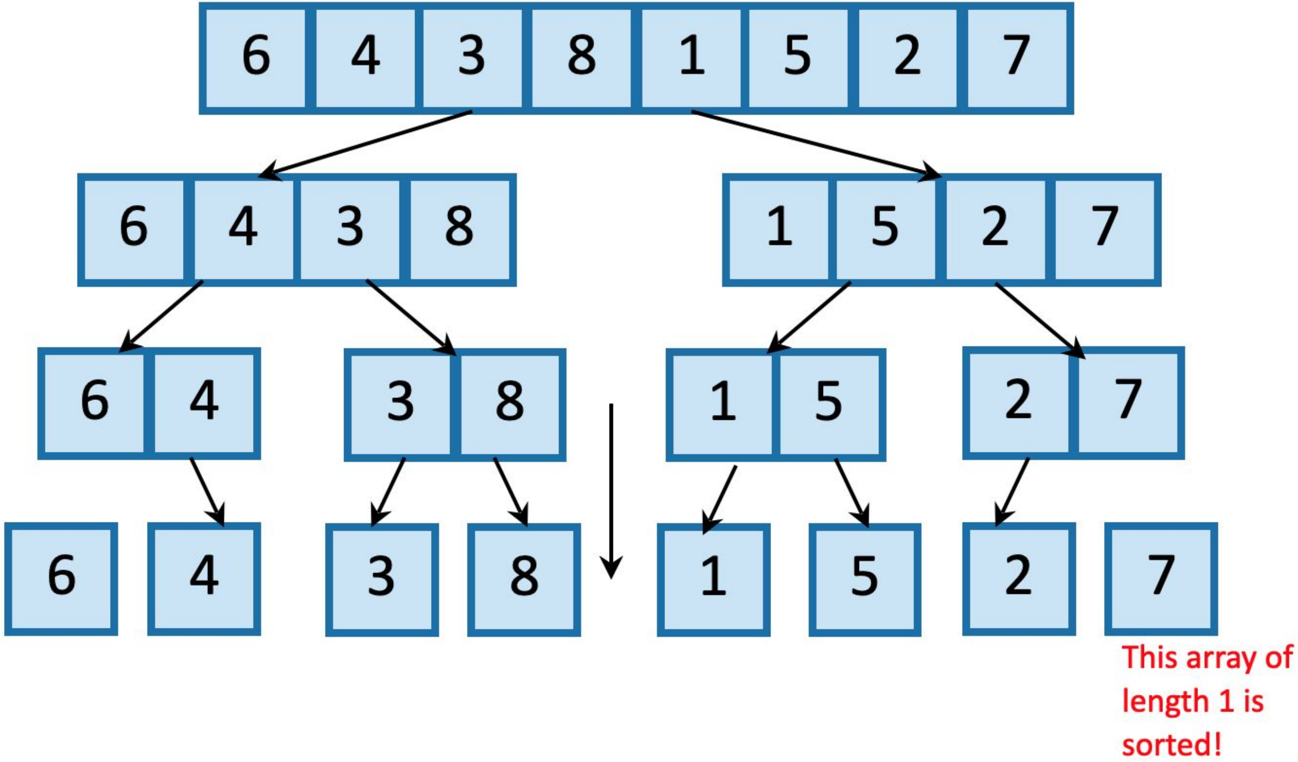
Sort the right half

- **return** MERGE(L,R)

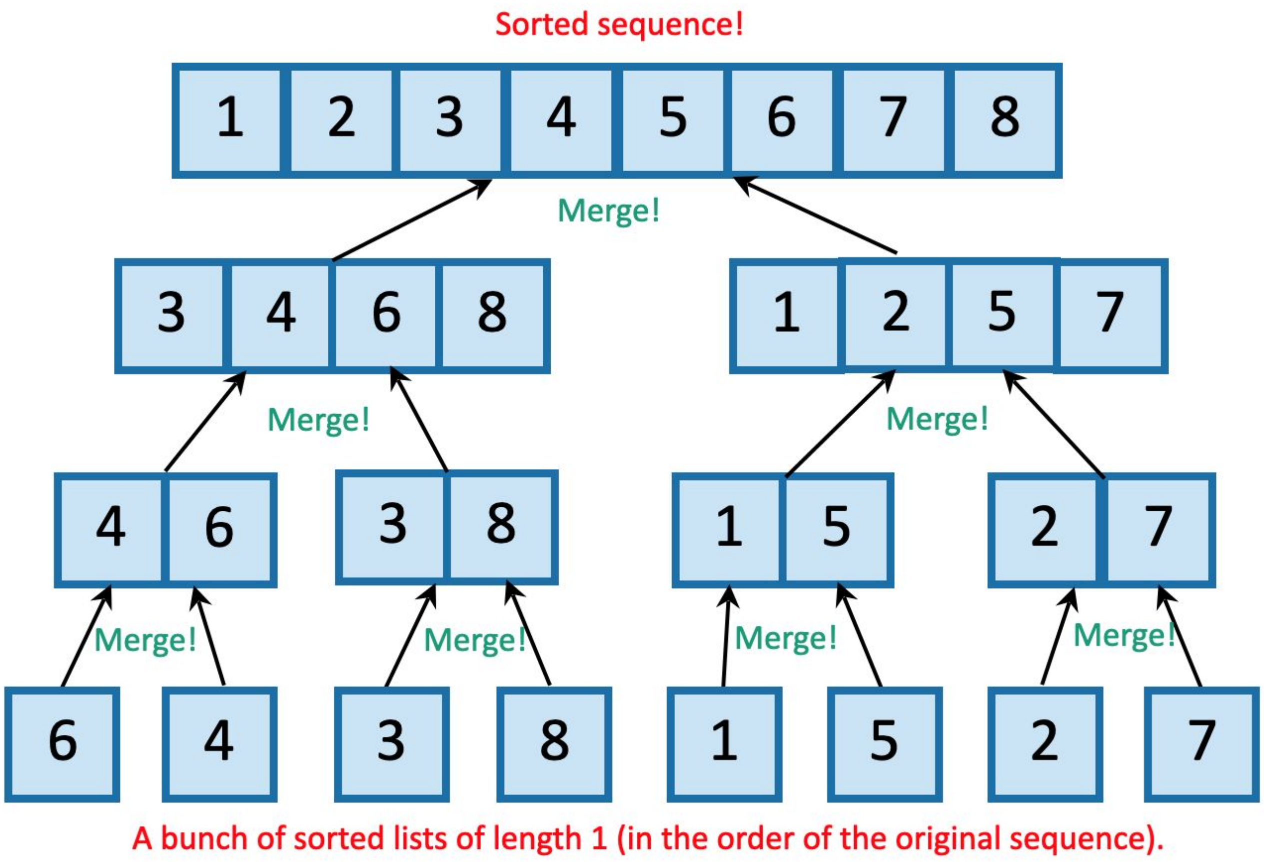
Merge the two halves

What actually happens?

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



Then, merge them all back up!



But

- Is it correct?
  - Yes! We can prove it!

- Is it fast?
  - Yes! Runs in O(n log n) time. We can prove it!
  - Next time, we'll see that this is actually the best we can do for sorts that operate by comparing elements (as MergeSort does).

Induction on recursion levels

We assume here that n is an integer power of 2. It's not hard to adapt the idea, though.

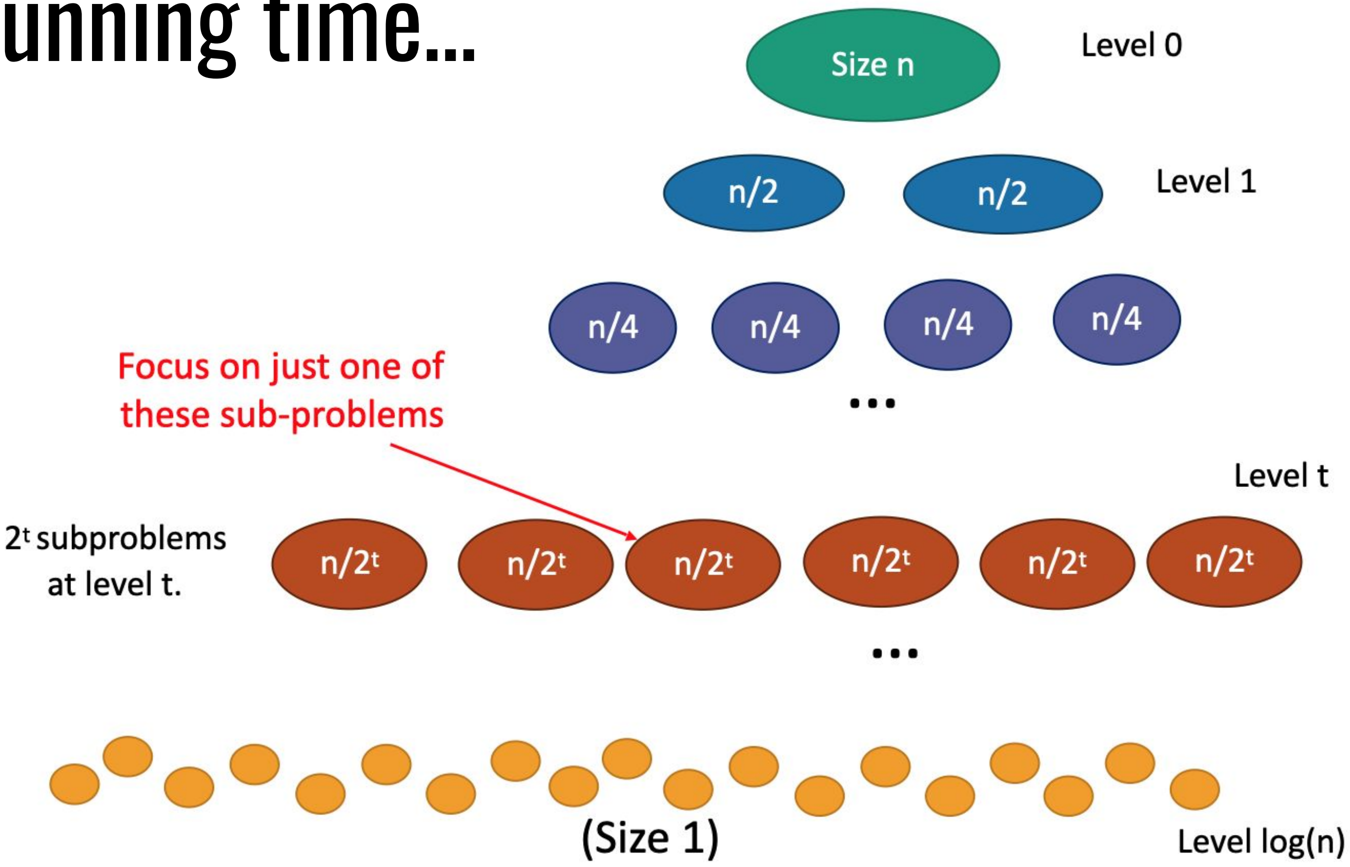
- **Claim:** When we are i levels up from the bottom, every chunk of size 2^i is sorted. (Note: we break the list into $n / 2^i$ chunks of size 2^i each. The claim is not about *arbitrary* blocks of size 2^i.)

- **Base case:** 0 levels up from the bottom, each chunk of size 1 is trivially sorted.

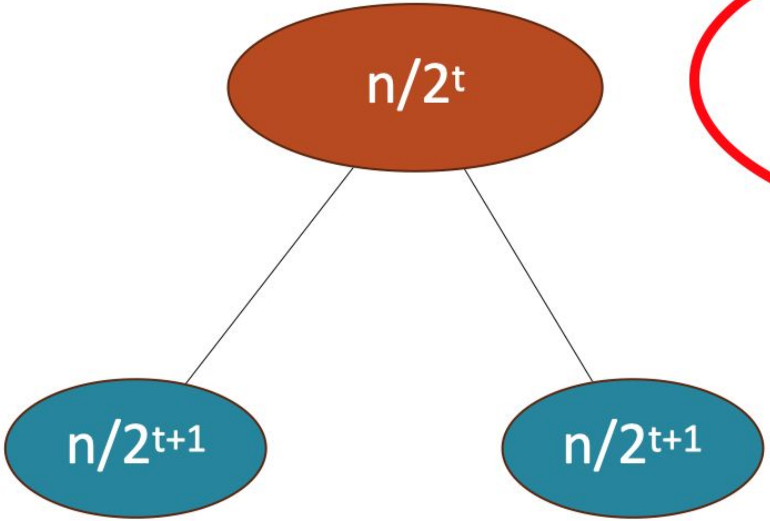
- **Inductive step:** Suppose the claim holds for all $0 \leq n < k$. We will show that it holds for $n = k$.
  - Consider any chunk of size 2^k. It is formed from merging the two sorted lists of size 2^{k-1} in the level below it.
  - Inductively, we know these are sorted.
  - As long as Merge correctly merges sorted lists (which we could also prove if we wanted), then the chunk of size 2^k is sorted.

Then at the end of the procedure, in particular, the overall list is sorted!

Now for the running time...



How much work in this sub-problem?



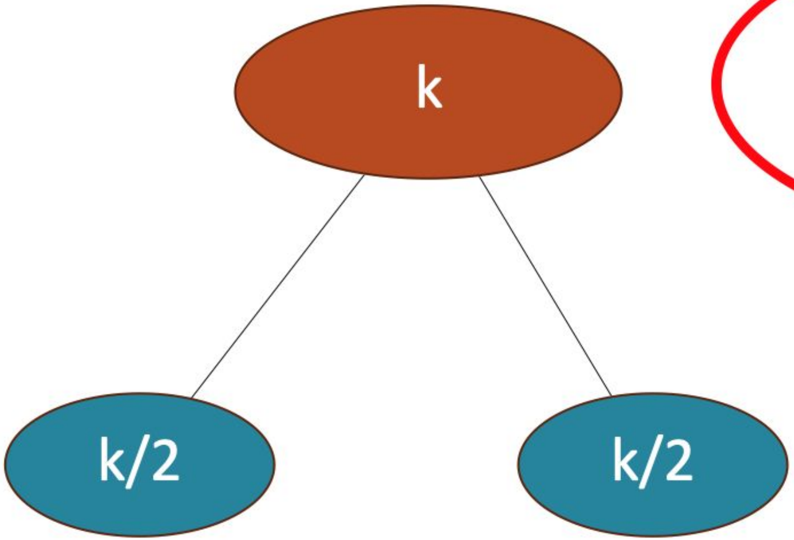
Time spent MERGE-ing the two subproblems

+

Time spent within the two sub-problems

How much work in this sub-problem?

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

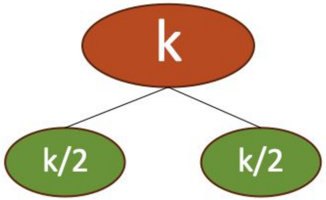


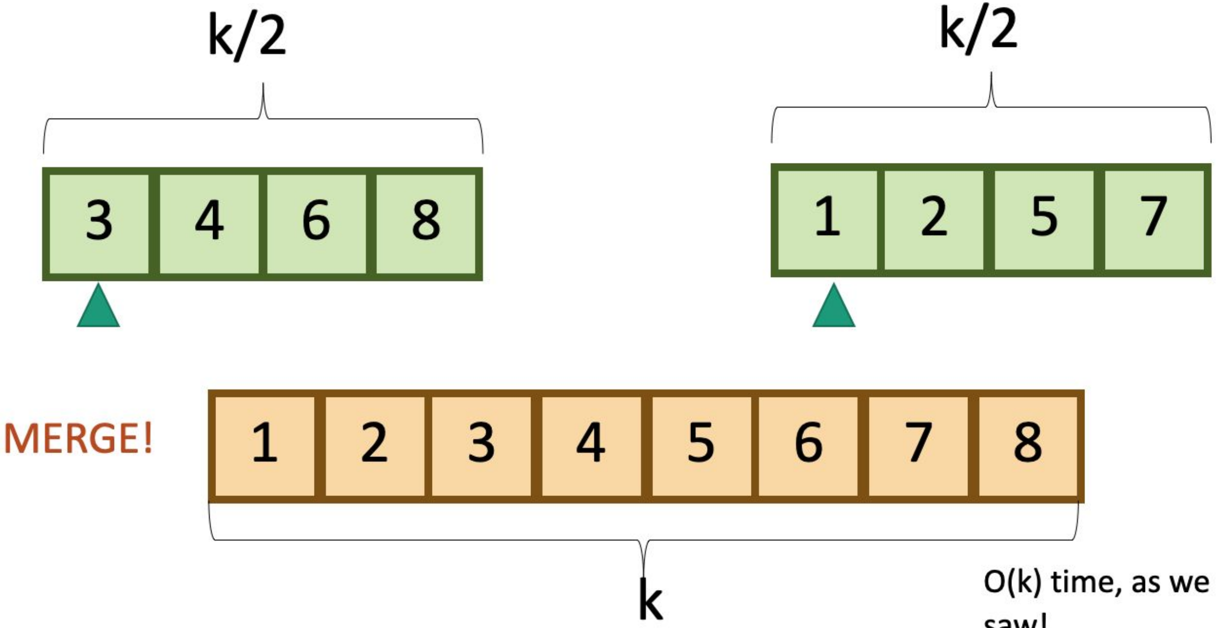
Time spent MERGE-ing the two subproblems

+

Time spent within the two sub-problems

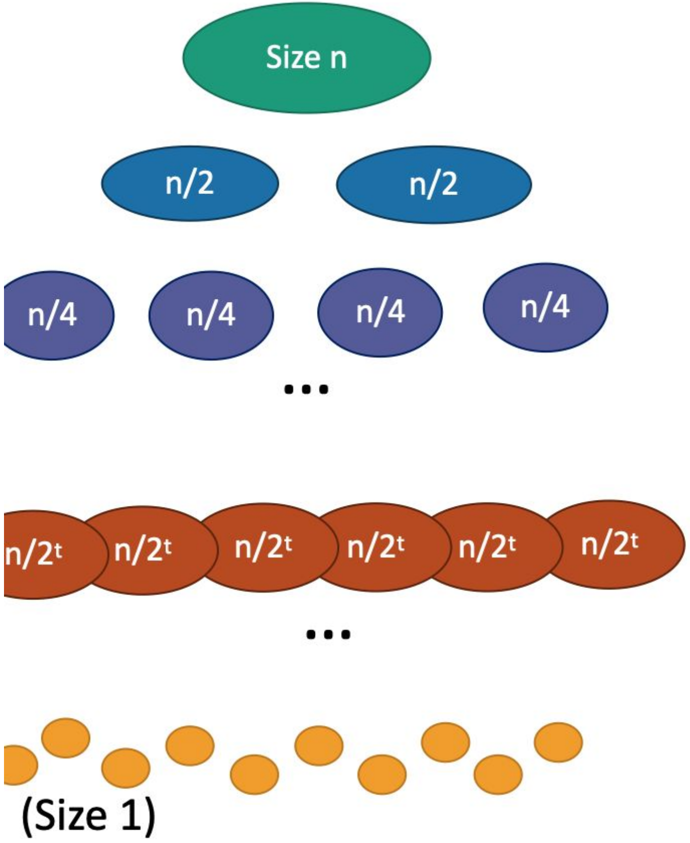
How long does it take to MERGE?



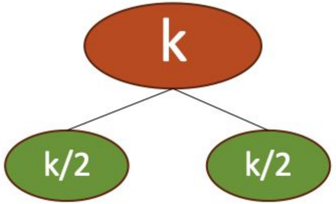


O(k) time, as we saw!

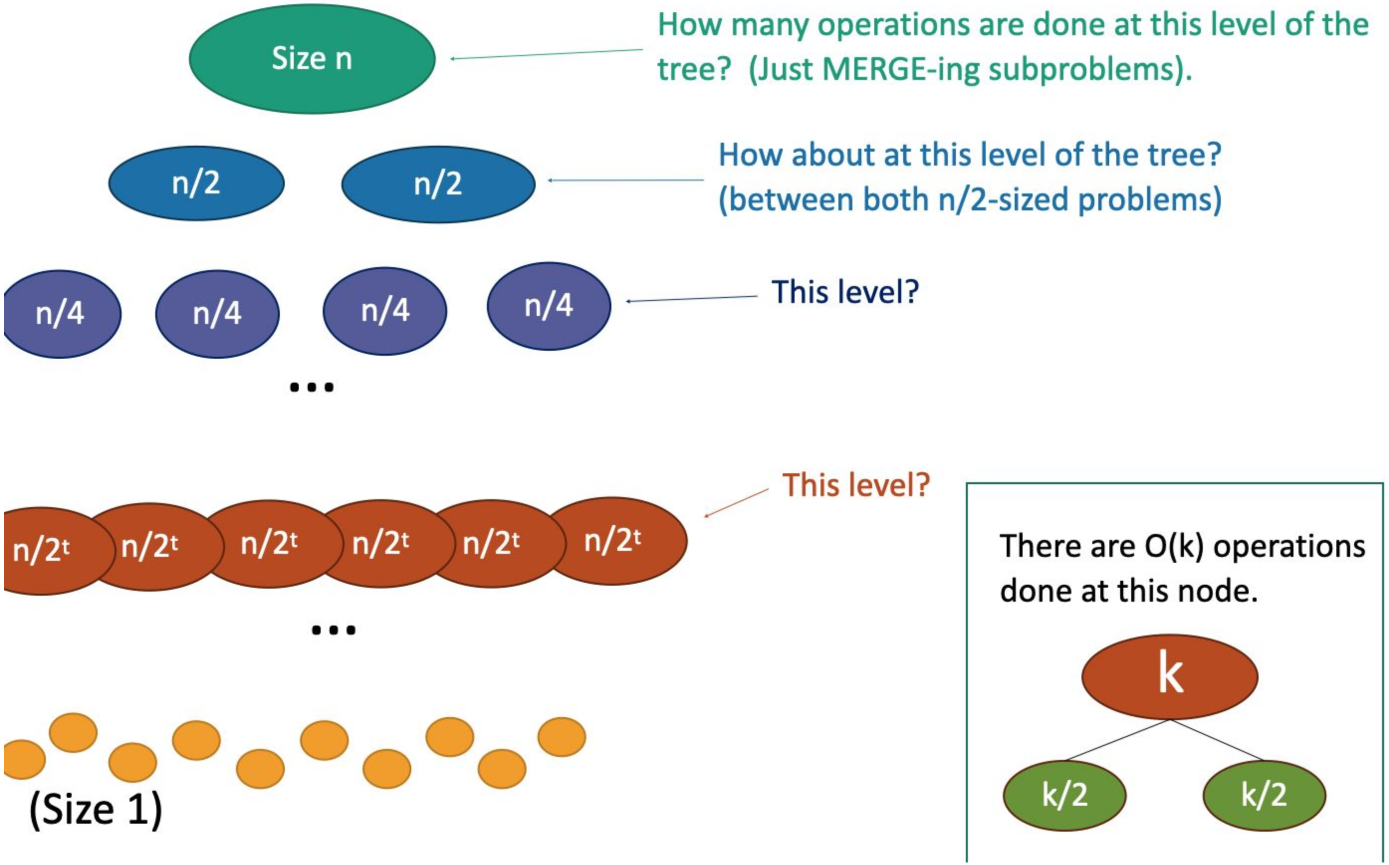
Recursion tree



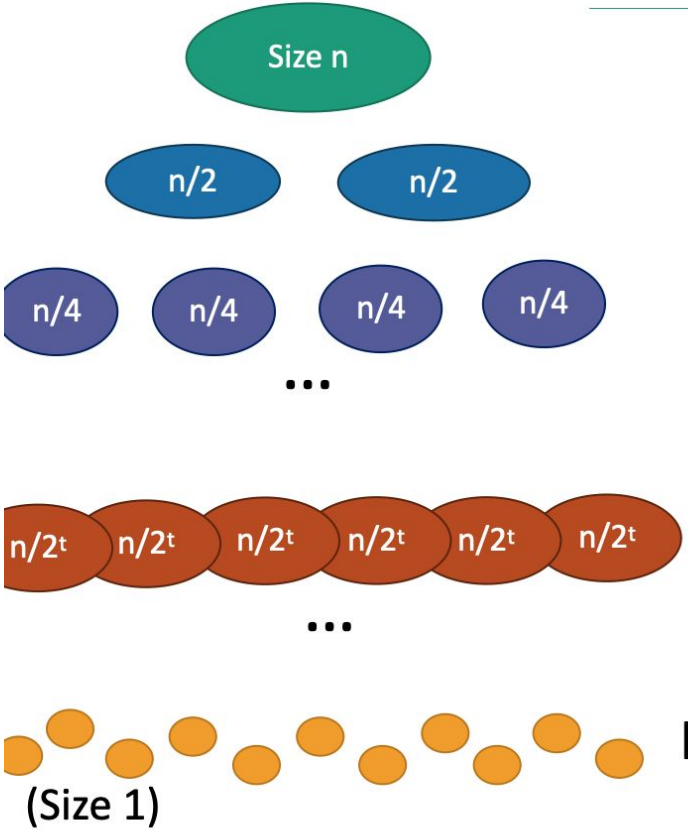
There are O(k) operations done at this node.



Recursion tree



Recursion tree



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

Total runtime…

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

The running time, viewed as a recurrence...

Let $T(n)$ denote the running time of a procedure (in this case, MergeSort).

For MergeSort, we can define $T(n)$ recursively as:

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

work of two subproblems of half size

work of merging

This depends on itself! It's a **recurrence**.

Solving divide-and-conquer recurrences

We argued that this recurrence $T(n) = 2T(n/2) + O(n)$ solves to $T(n) = O(n \log n)$.

Is there a more general way of solving this kind of recurrence without doing all that work again?

- See Homework 1, Problem 3!

Solving other recurrences

- What if the recurrence has some other form? Or what if we want an exact solution?

- Unfortunately, there is no general all-purpose technique for this. We'll look at some possibilities here and in the next lecture.

Some conventions about T(n)

- $T(n)$ represents a running time, so it is always nonnegative.
- n represents something like the size of the input, so it is always nonnegative.

$T(n) = T(n-1) + 1, T(1) = 1$

- One way: start at the top, unroll, find a pattern.

$T(n)$

$= T(n-1) + 1$ *one 1*

$= (T(n-2) + 1) + 1$ *two 1s*

...

$= T(1) + 1 + \ldots + 1$ *$n-1$ 1s*

Therefore
$T(n) = n$

$T(n) = T(n-1) + 1, T(1) = 1$

- Or: start at the bottom and find the pattern.

$T(1) = 1$

$T(2) = T(1) + 1 = 1 + 1 = 2$

$T(3) = T(2) + 1 = 2 + 1 = 3$

etc. (we could make this and the previous argument more rigorous via induction)

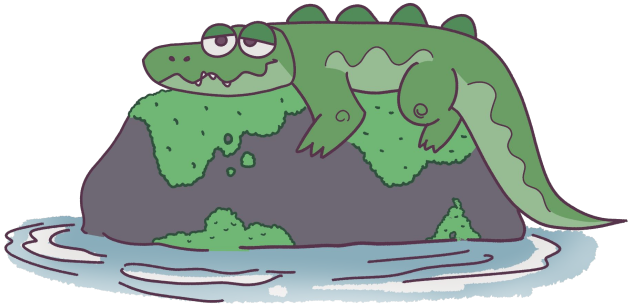
Therefore $T(n) = n$

$T(n) = T(n-1) + n, T(1) = 1$

- How about this one?

The secret to CS theory (and math)

- Work some small examples
- Spot a pattern
- Prove the pattern
- Pretend that you came up with the pattern out of nowhere while sitting in a comfortable armchair, sipping some brandy by a fire

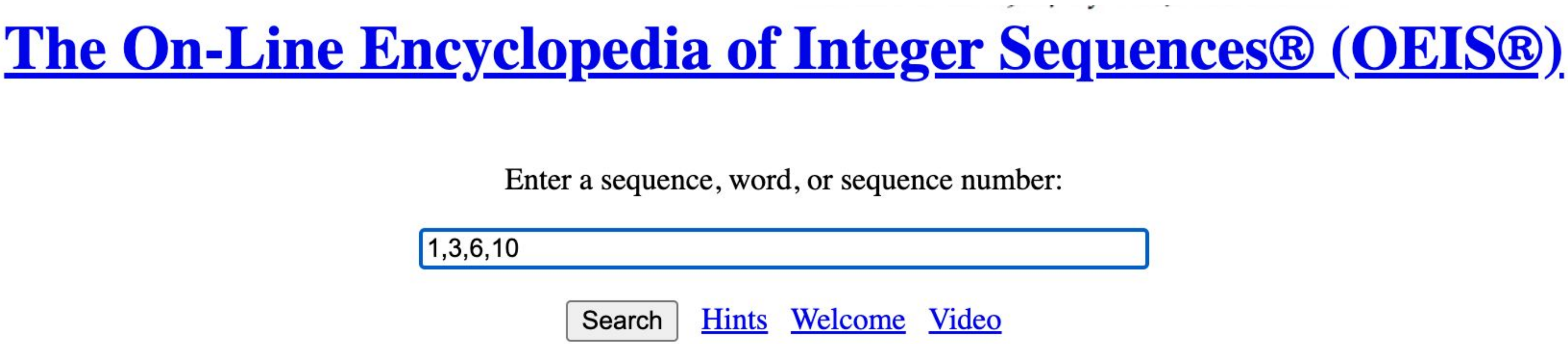


T(n) = T(n-1) + n, T(1) = 1

- $T(2) = T(1) + 2 = 1 + 2 = 3$
- $T(3) = T(2) + 3 = 3 + 3 = 6$
- $T(4) = T(3) + 4 = 6 + 4 = 10$

1, 3, 6, 10… this looks familiar

Another secret to CS theory (and math)



If you have a sequence, put it into oeis.org. *Congrats! Now you are a number theorist*

Search: **seq:1,3,6,10**

Displaying 1-10 of 533 results found. page 1 2 3 4 5 6 7 8 9 10 ... 54

Sort: relevance | references | number | modified | created Format: long | short | data

A000217 Triangular numbers: a(n) = binomial(n+1,2) = n*(n+1)/2 = 0 + 1 + 2 + ... + n. +30 4312
(Formerly M2535 N1002)

0, **1, 3, 6, 10**, 15, 21, 28, 36, 45, 55, 66, 78, 91, 105, 120, 136, 153, 171, 190, 210, 231, 253, 276, 300, 325, 351, 378, 406, 435, 465, 496, 528, 561, 595, 630, 666, 703, 741, 780, 820, 861, 903, 946, 990, 1035, 1081, 1128, 1176, 1225, 1275, 1326, 1378, 1431 (list; graph; refs; listen; history; text; internal format)

$T(n) = n(n+1) / 2$
you say? Hm,
very interesting

A000004	The zero sequence. (Formerly M0000)

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 (list; graph; refs; listen; history; text; internal format)

OFFSET 0,1

LINKS

N. J. A. Sloane, Table of n, a(n) for n = 0..1000 [Useful when plotting one sequence against another. See Swayne link.]

Luis Manuel Rivera, Integer sequences and k-commuting permutations, arXiv preprint arXiv:1406.3081 [math.CO], 2014-2015.

D. E. Swayne, Plot pairs of sequences in the OEIS

There are all kinds of fun and beautiful sequences to discover!

$T(n) = T(n-1) + n, T(1) = 1$

- **Claim:** $T(n) = n(n+1) / 2$, for all $n \geq 1$.
- **Base case:** $T(1) = 1$ (given!)
- **Inductive step:** Suppose the claim holds for all $1 \leq n < k$. We'll show it holds for $n = k$:
  - $T(k) = T(k-1) + k$ (definition)
  - $T(k) = (k-1)(k) / 2 + k$ (inductive step with $n-1$)
  - $T(n) = k^2 / 2 - k / 2 + k$
    $= k^2 / 2 + k / 2 = (k^2 + k)/ 2 = k(k+1) / 2$ ■

Next week!

- How fast can we find the median of a list?
    - Mind-blowing algorithm!
- We can't beat O(n log n) for sorting!
- We can beat O(n log n) for sorting!
- Randomness is our friend and helps us sort!
- Absolutely amazing randomized algorithm for graph cuts (Ian's favorite algorithm)