CSE100: Design and Analysis of Algorithms
Lecture 04 – Recurrences, Asymptotics

Jan 27th 2022

MergeSort (cont.)

Big-O notation, more recurrences!!

Today (Part 1)

- Integer Multiplication (wrap up)
- Sorting Algorithms
  - InsertionSort: does it work and is it fast?
  - MergeSort: does it work and is it fast? (wrap up)
- Return of divide-and-conquer with Merge Sort
- Skills:
  - Analyzing correctness of iterative and recursive algorithms.
  - Analyzing running time of recursive algorithms.

Next Time:

- How do we measure the runtime of an algorithm?
  - Worst-case analysis
  - Asymptotic Analysis

MergeSort Pseudocode (review)

MERGESORT(A):

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

If A has length 1,
It is already sorted!

- L = MERGESORT(A[1 : n/2])

Sort the left half

- R = MERGESORT(A[n/2+1 : n])

Sort the right half

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

Merge the two halves

Assume that n is a power of 2 for convenience.

It’s fast

CLAIM:

MergeSort requires at most 11n (log(n) + 1) operations to sort n numbers.

- Proof coming soon.
- But first, how does this compare to InsertionSort?
    - Recall InsertionSort used on the order of n^2 operations.

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

Assume that n is a power of 2 for convenience.

It's fast

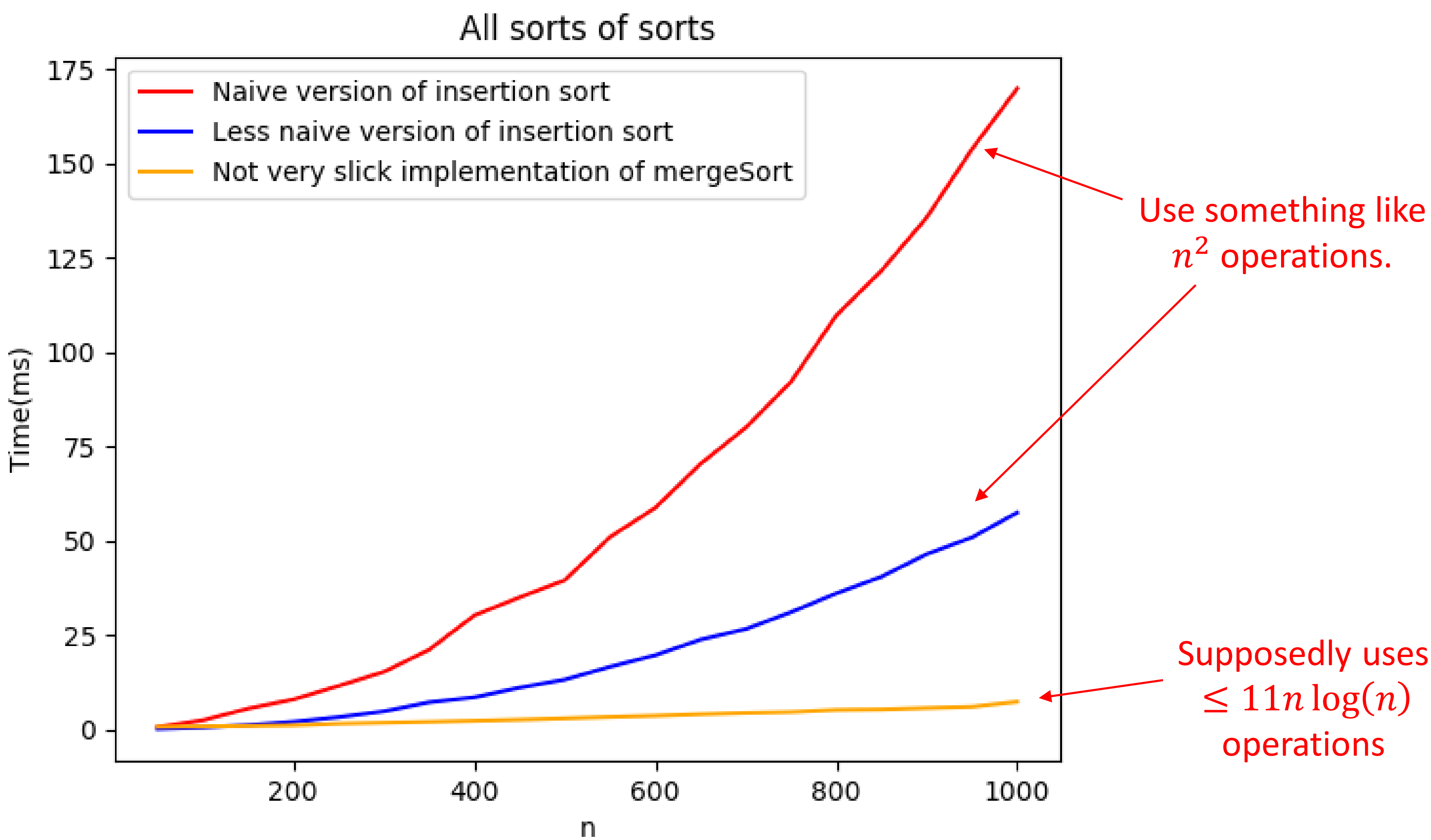
CLAIM:

MergeSort requires at most 11n (log(n) + 1) operations to sort n numbers.

- Proof coming soon.
- But first, how does this compare to InsertionSort?
  - Recall InsertionSort used on the order of n^2 operations.

What exactly is an "operation" here?
We're leaving that vague on purpose.
Also the number 11 is sort of made-up.

$n \log(n)$ vs. n^2? (Empirically)



Aside:

All logarithms in this course are base 2

Quick log refresher

- $2 \ \log \ n \quad \log \ n \ \log \log \ n \quad n \, nn \, n \ \log \ n \quad 2 \ \log \ n \quad = nn.$
- Intuition: log(n) is how many times you need to divide n by 2 in order to get down to 1.

$$32, \underbrace{16, 8, 4, 2, 1}_{\text{Halve 5 times}} \quad \Rightarrow \quad \log(32) = 5$$

$$64, \underbrace{32, 16, 8, 4, 2, 1}_{\text{Halve 6 times}} \quad \Rightarrow \quad \log(64) = 6$$

log(128) = 7

log(256) = 8

log(512) = 9

….

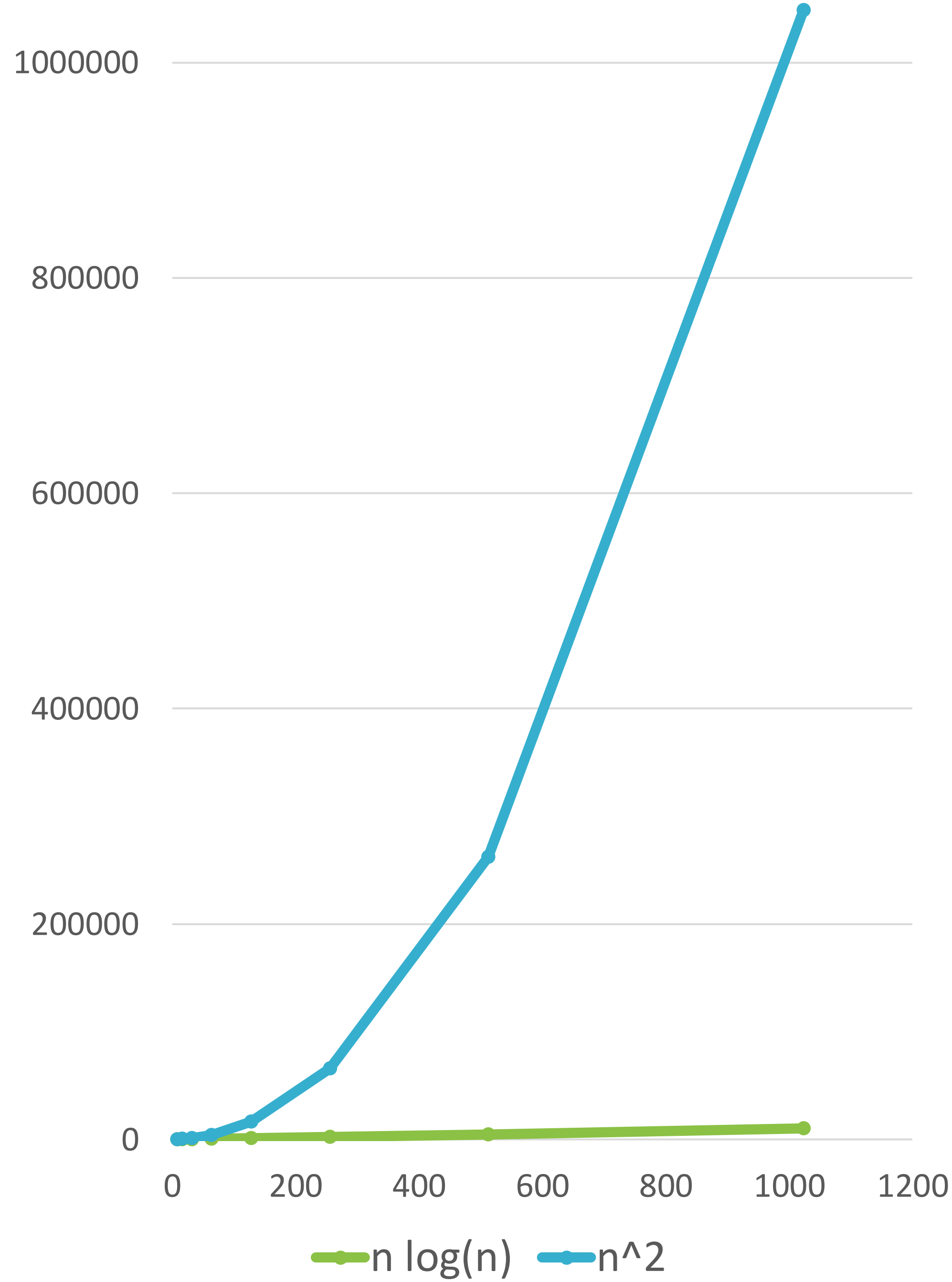
log(**# particles in the universe**) < 280

- log(n) grows very slowly!

n log(n) vs n^2

continued

n	n log(n)	n^2
8	24	64
16	64	256
32	160	1024
64	384	4096
128	896	16384
256	2048	65536
512	4608	262144
1024	10240	1048576



Assume that n is a power of 2 for convenience.

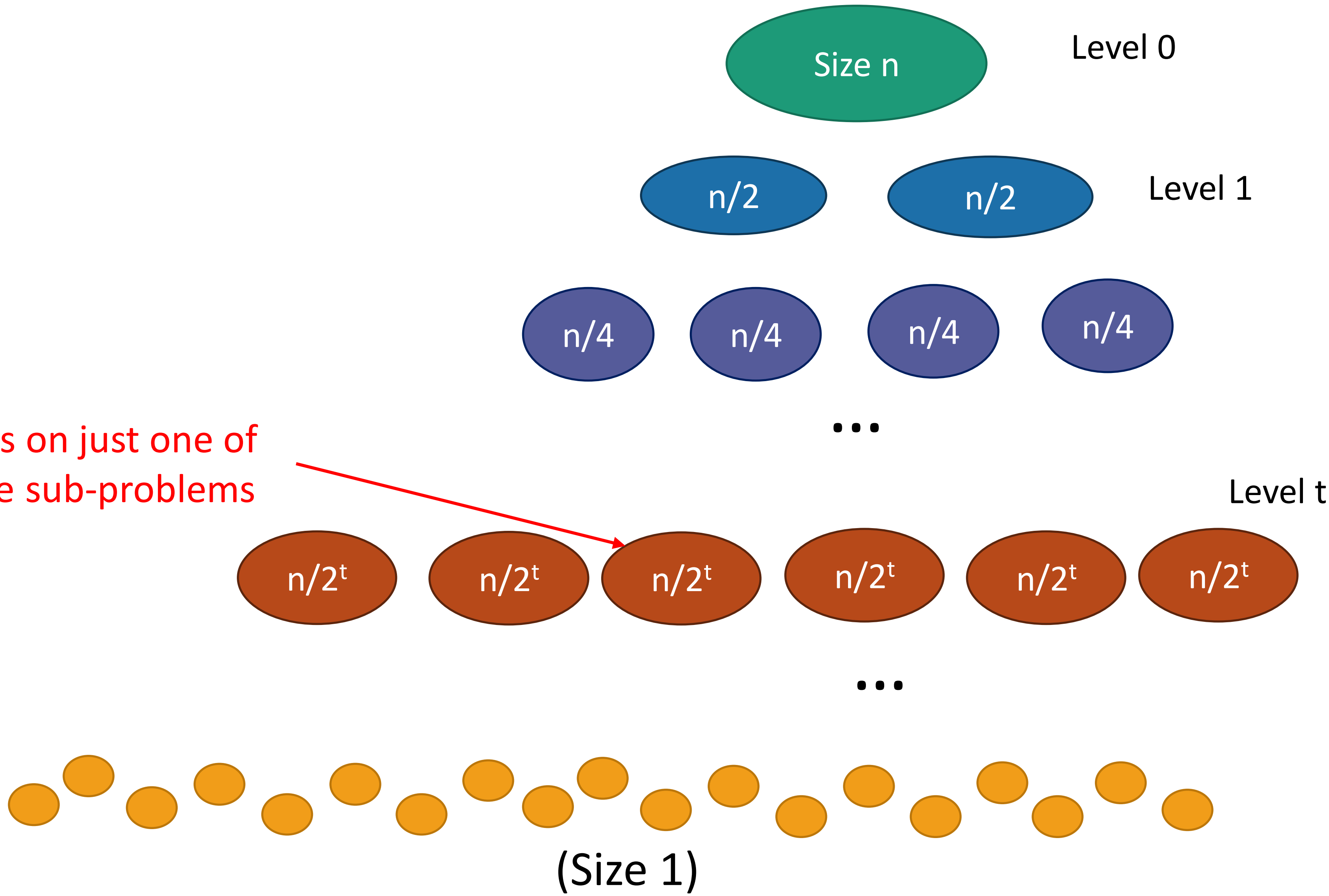
Now let's prove the claim

CLAIM:

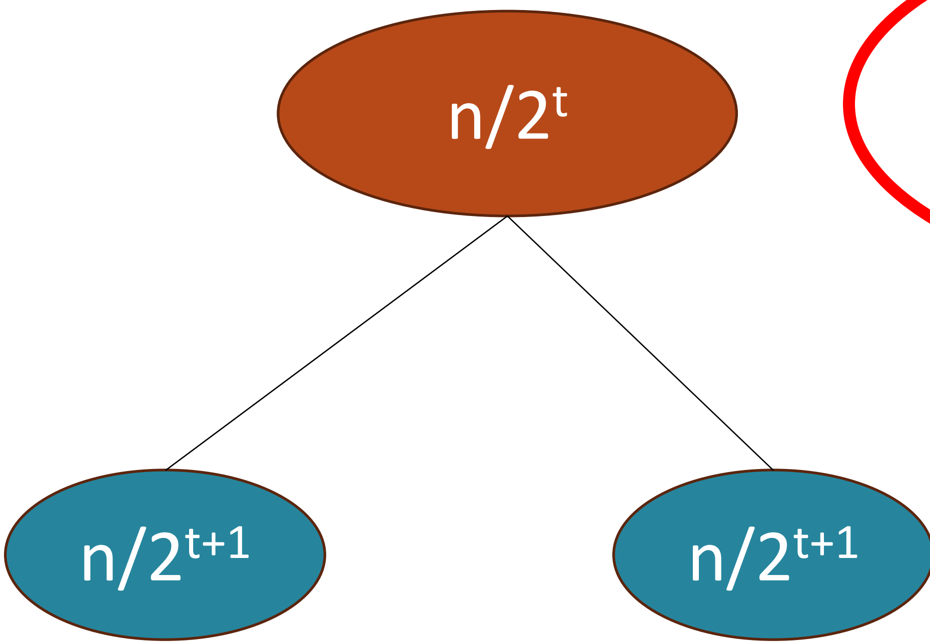
MergeSort requires at most $11n(\log(n) + 1)$ operations to sort n numbers.

Let’s prove the claim

Focus on just one of these sub-problems



How much work in this sub-problem?



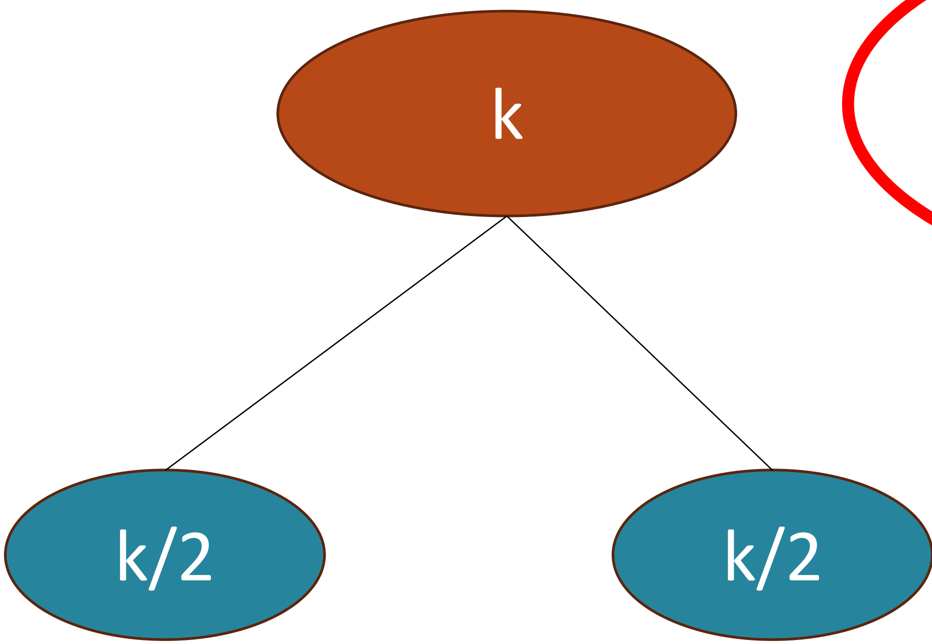
Time spent MERGE-ing the two subproblems

+

Time spent within the two sub-problems

How much work in this sub-problem?

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

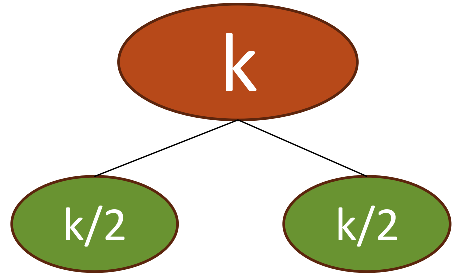


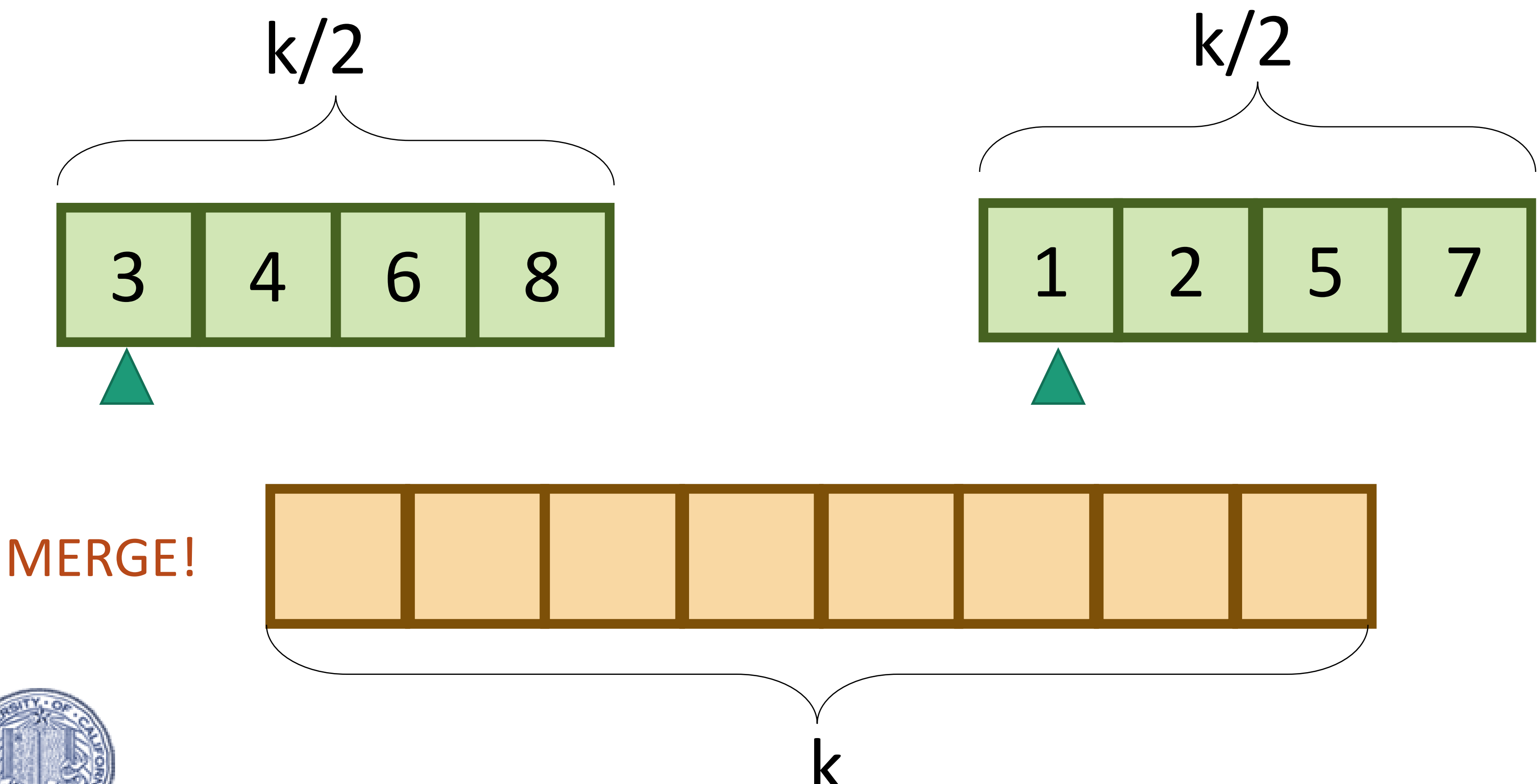
Time spent MERGE-ing the two subproblems

+

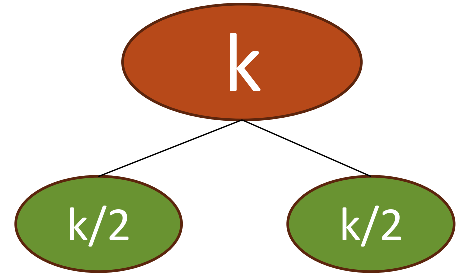
Time spent within the two sub-problems

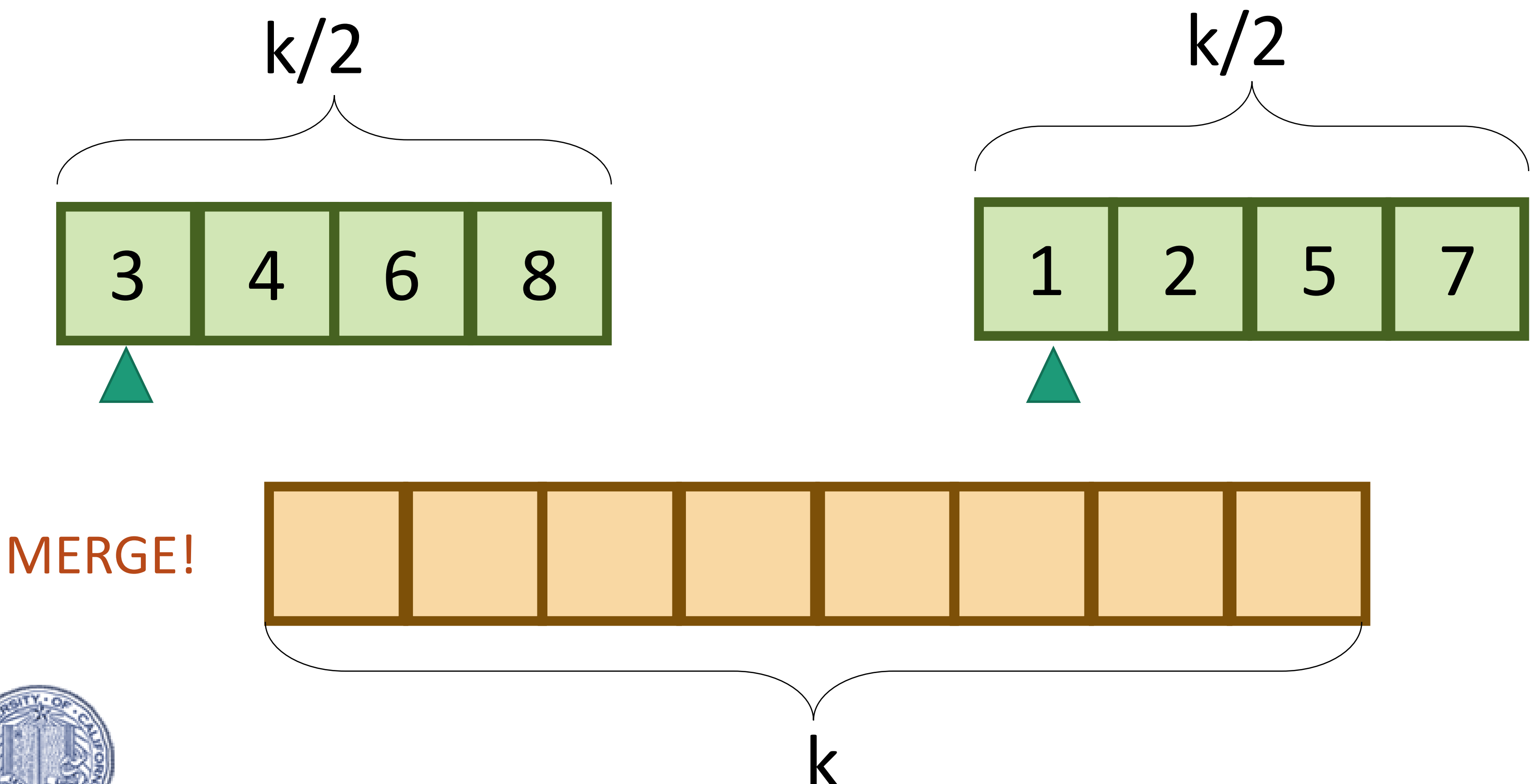
How long does it take to MERGE?



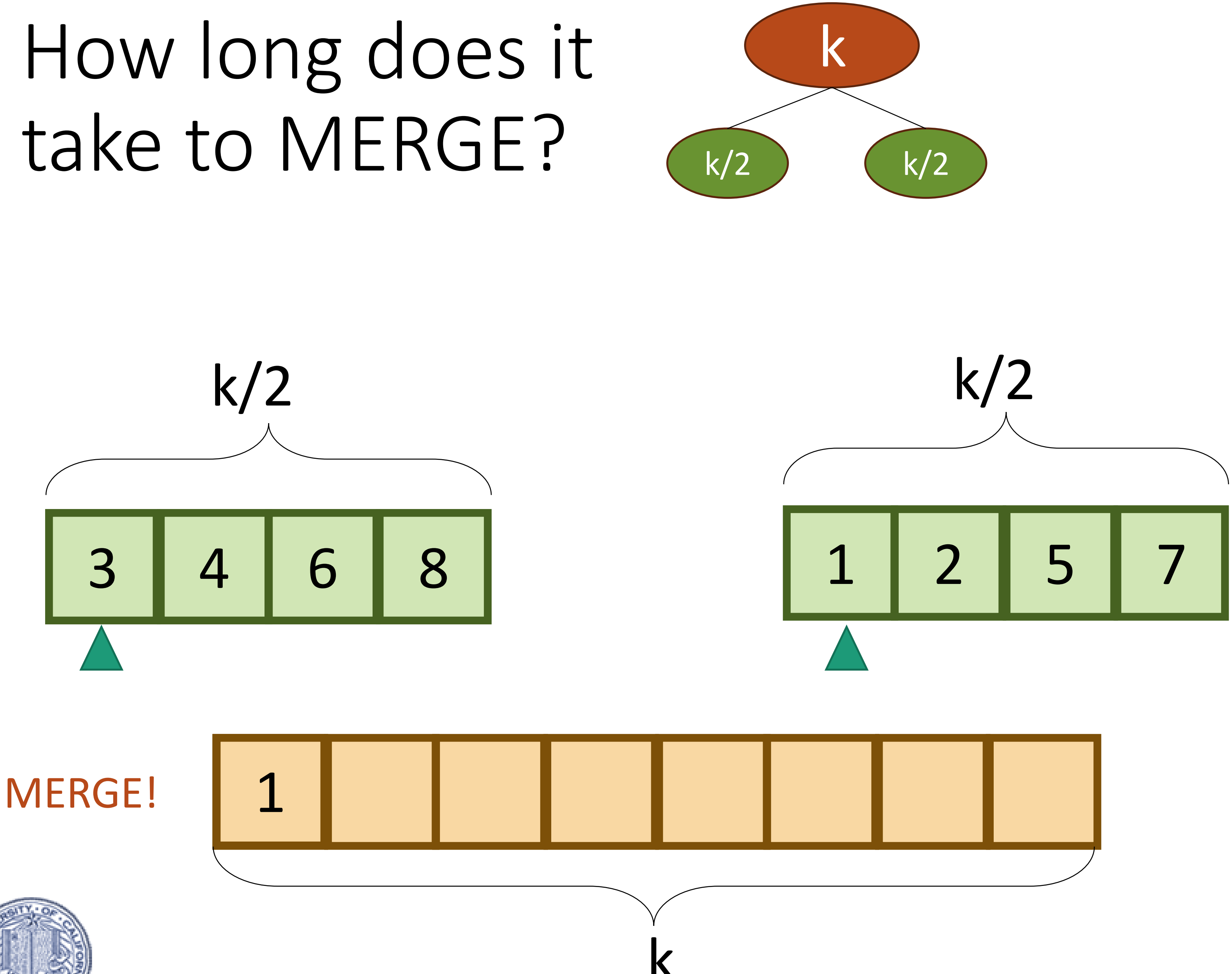


How long does it take to MERGE?

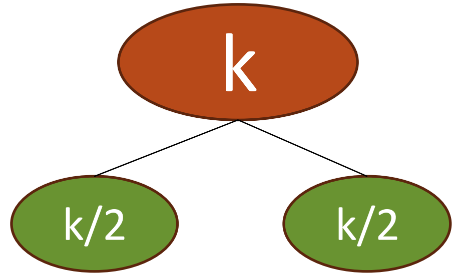


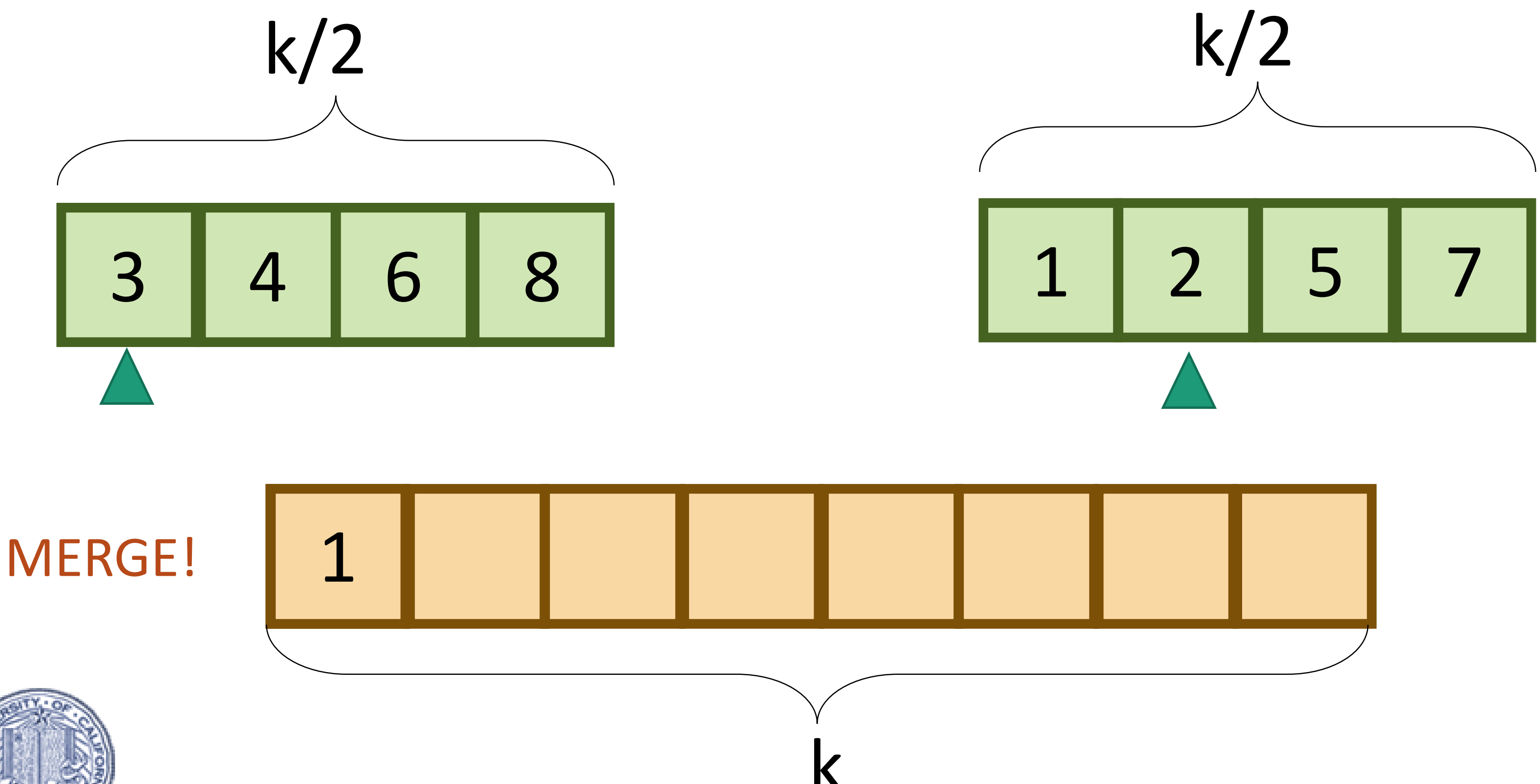


How long does it take to MERGE?

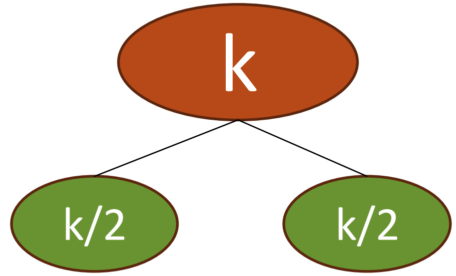


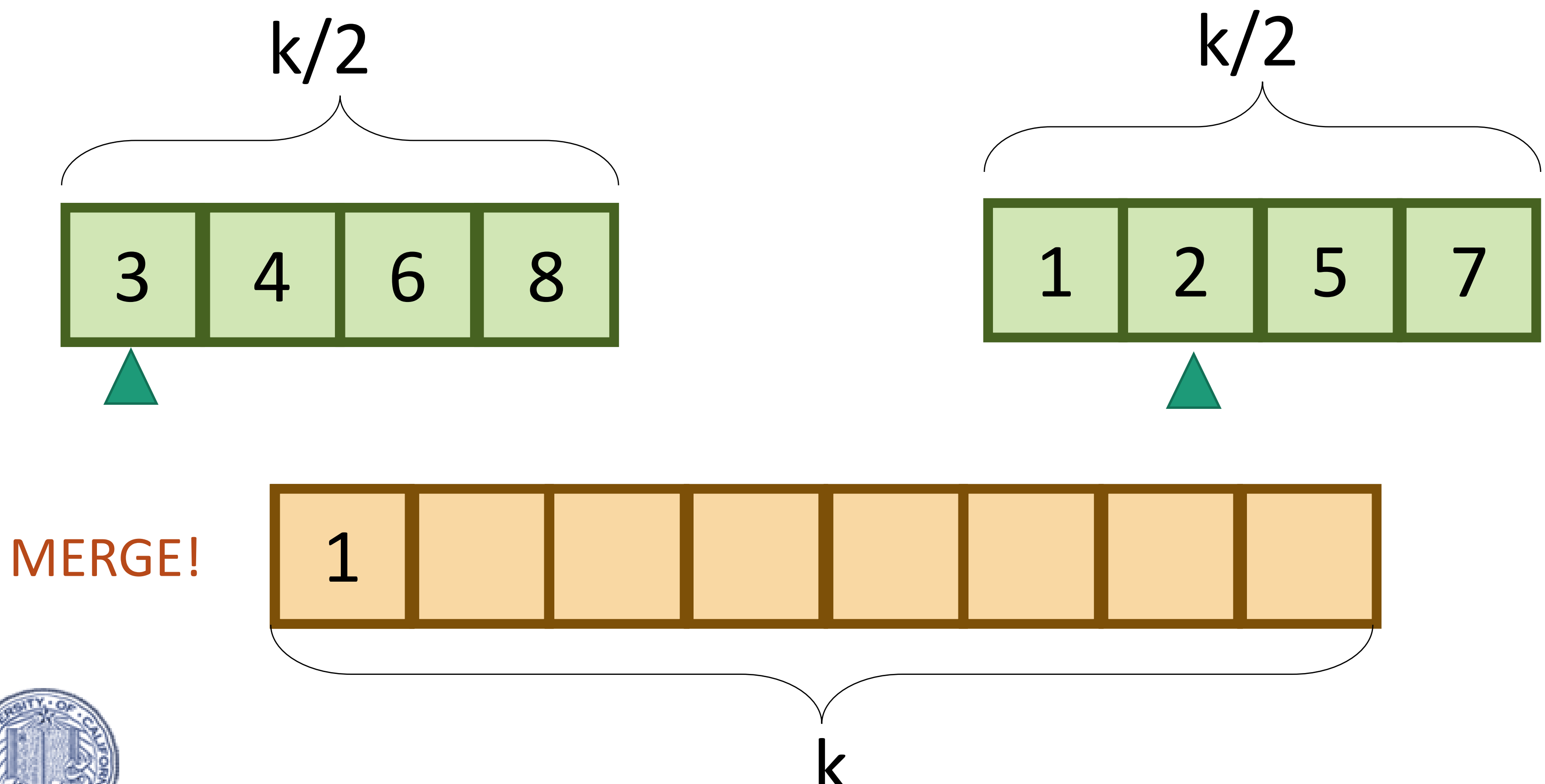
How long does it take to MERGE?



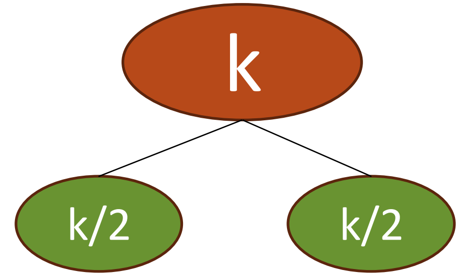


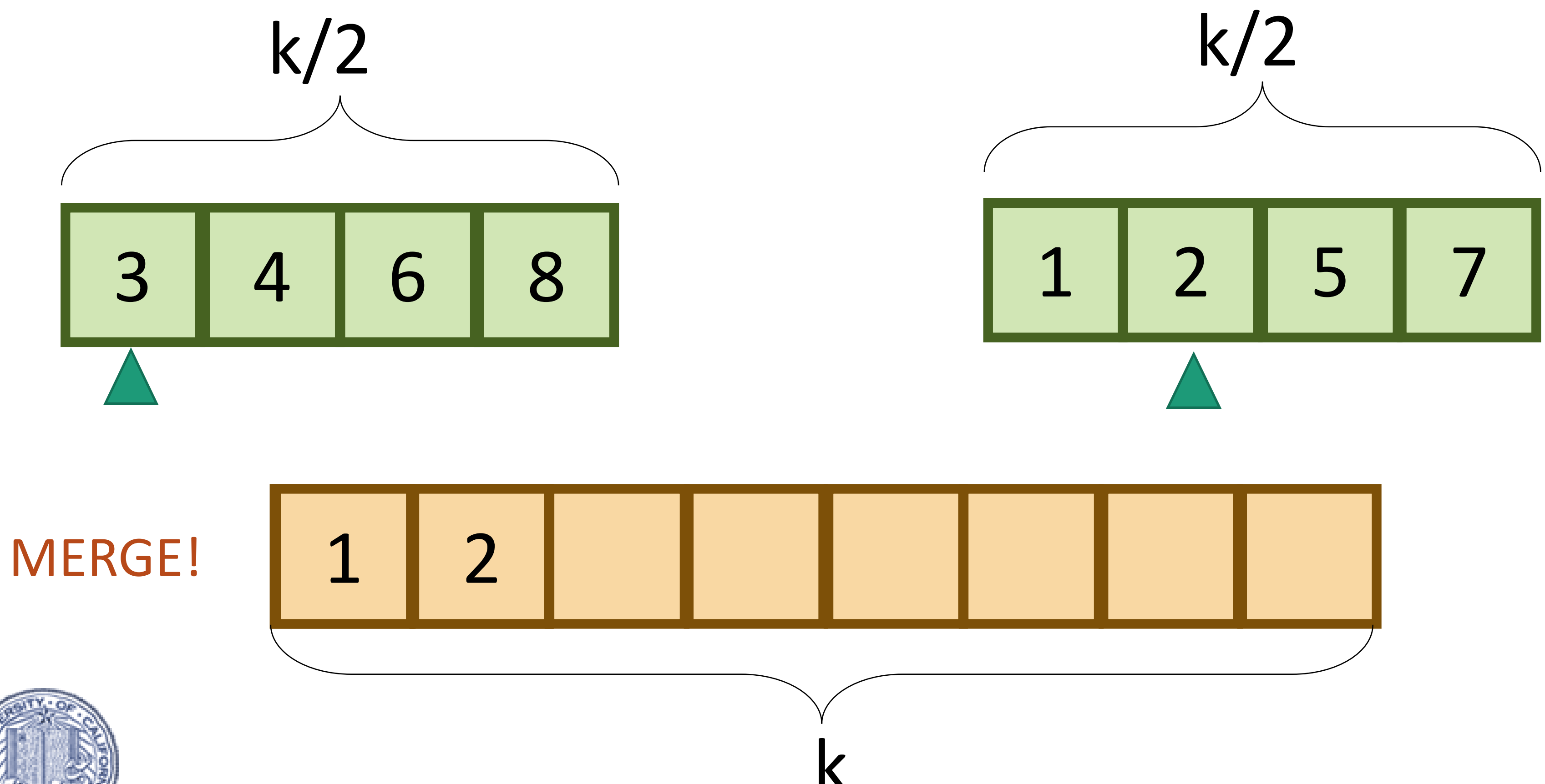
How long does it take to MERGE?



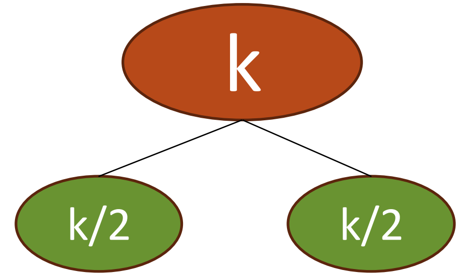


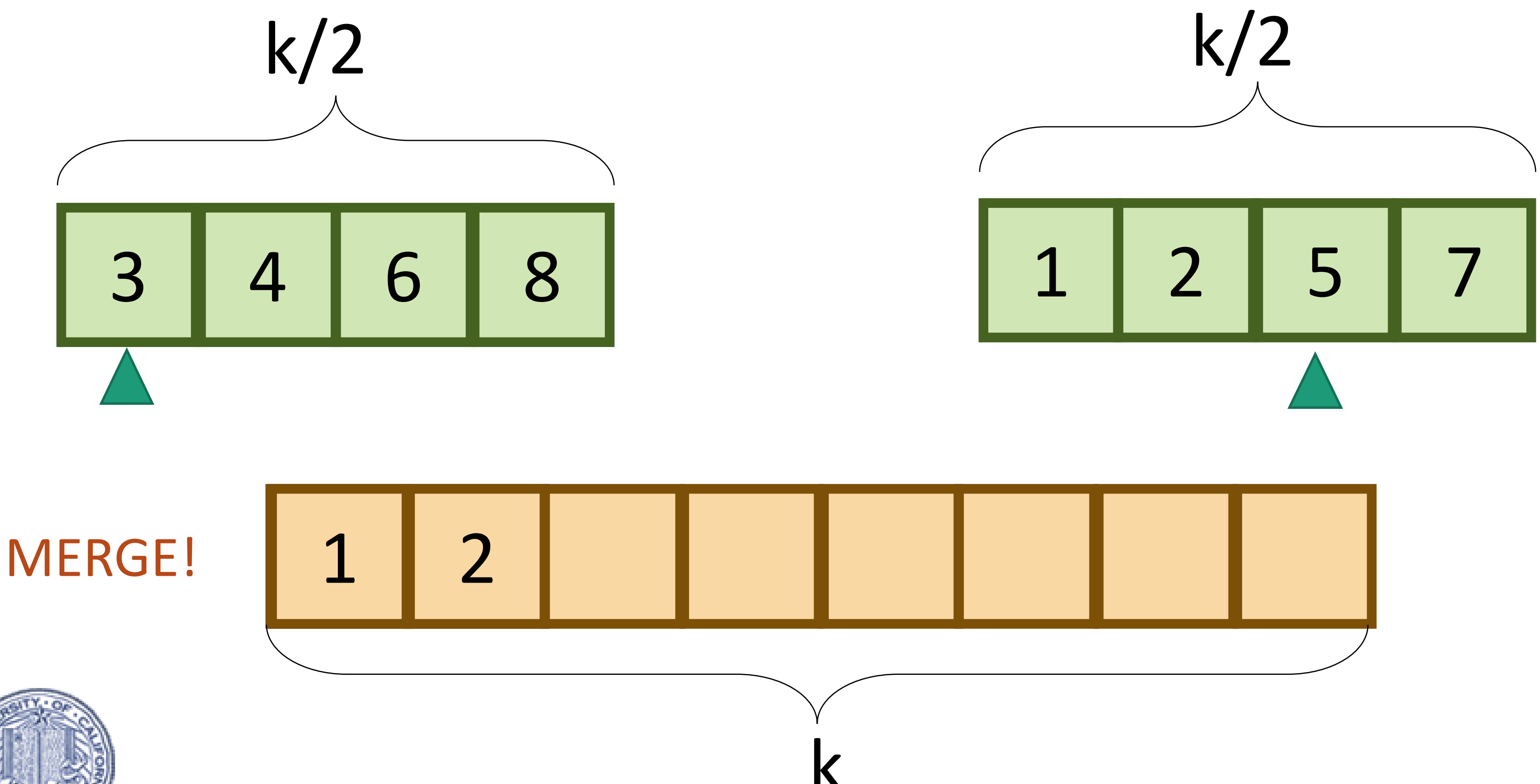
How long does it take to MERGE?



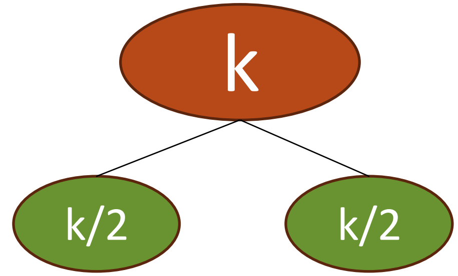


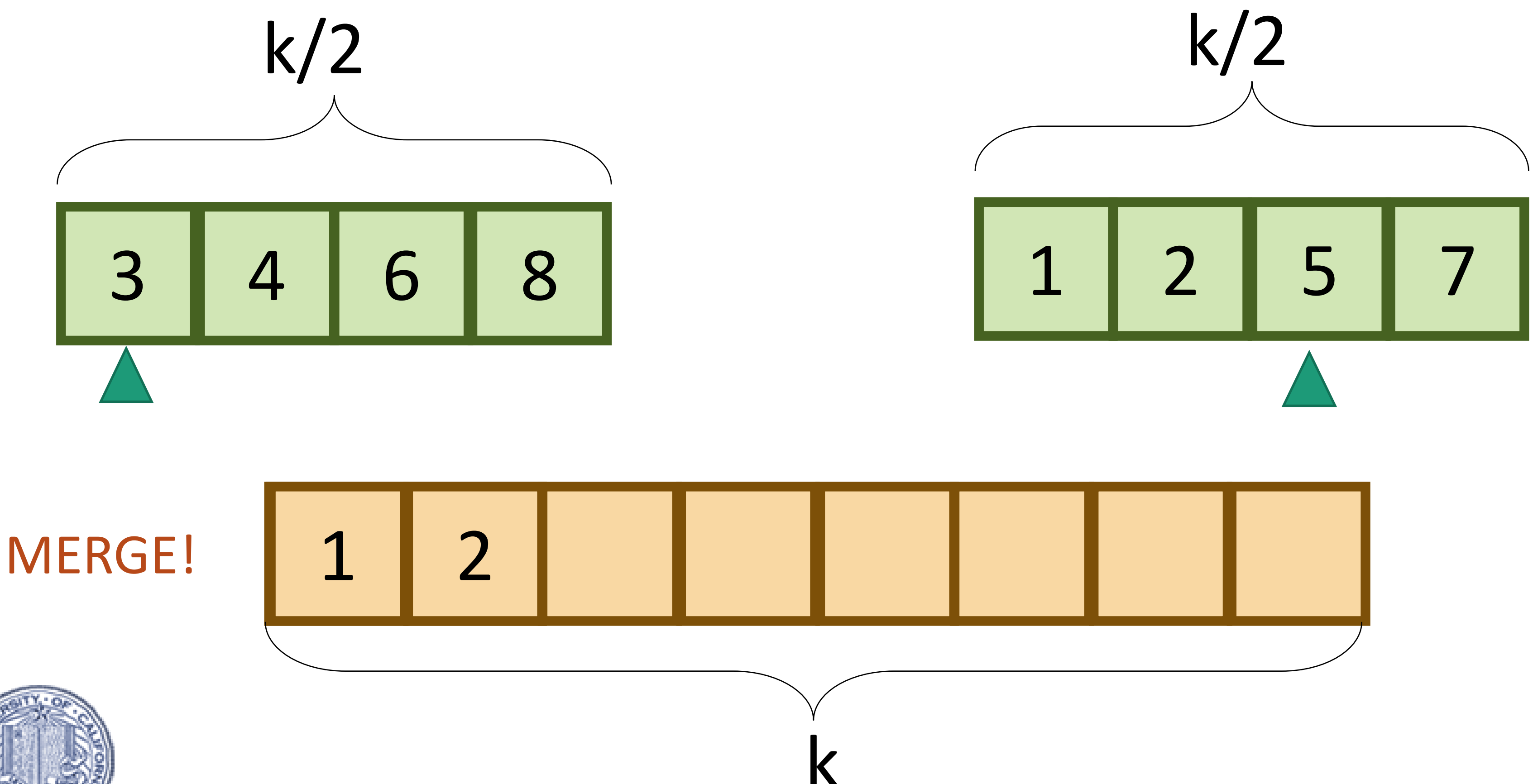
How long does it take to MERGE?



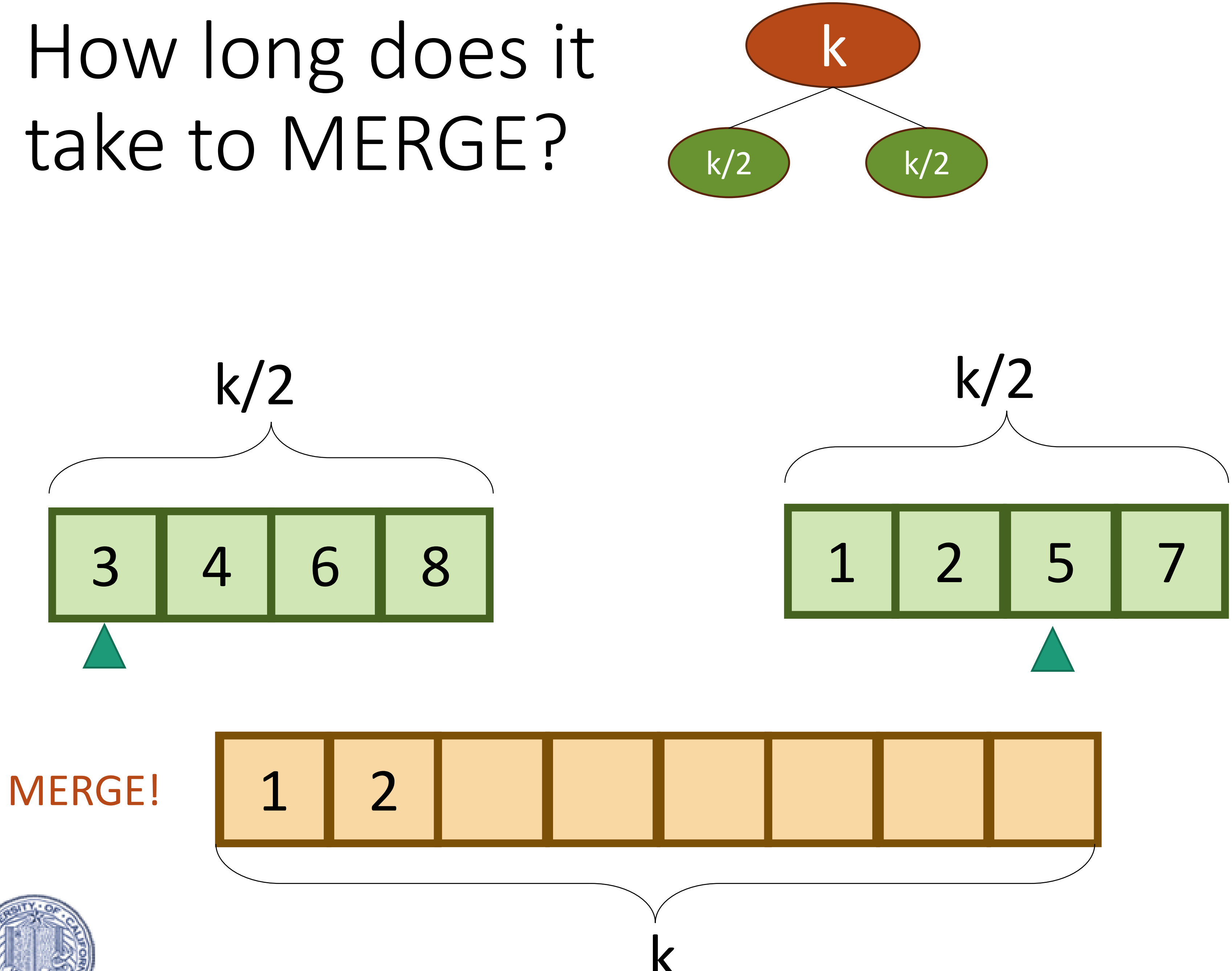


How long does it take to MERGE?

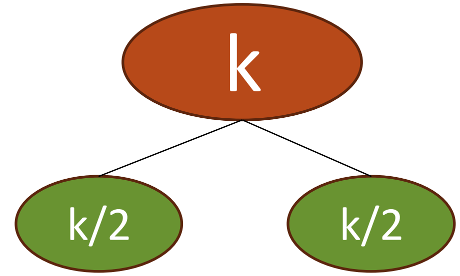


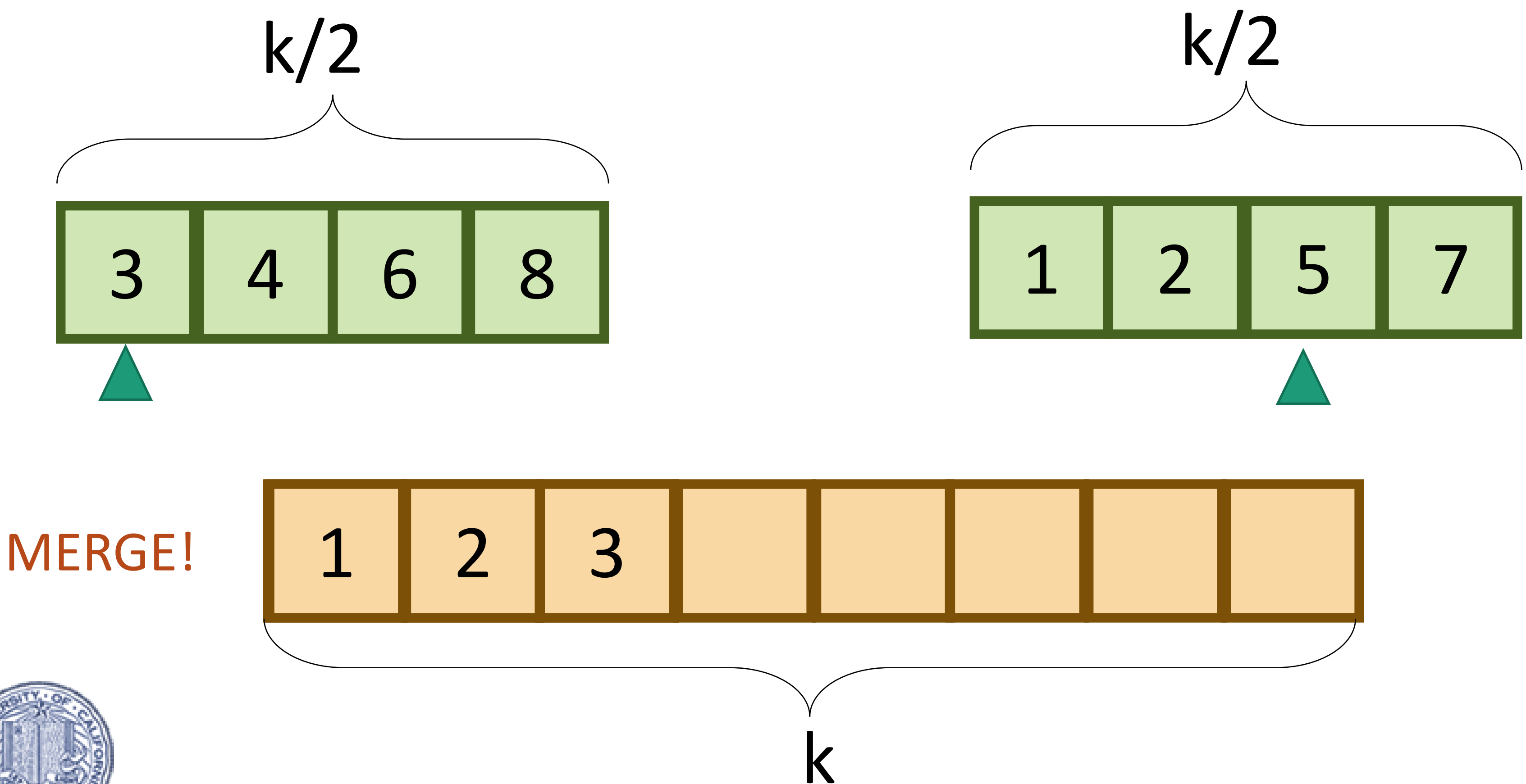


How long does it take to MERGE?

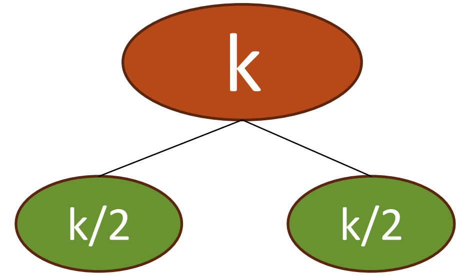


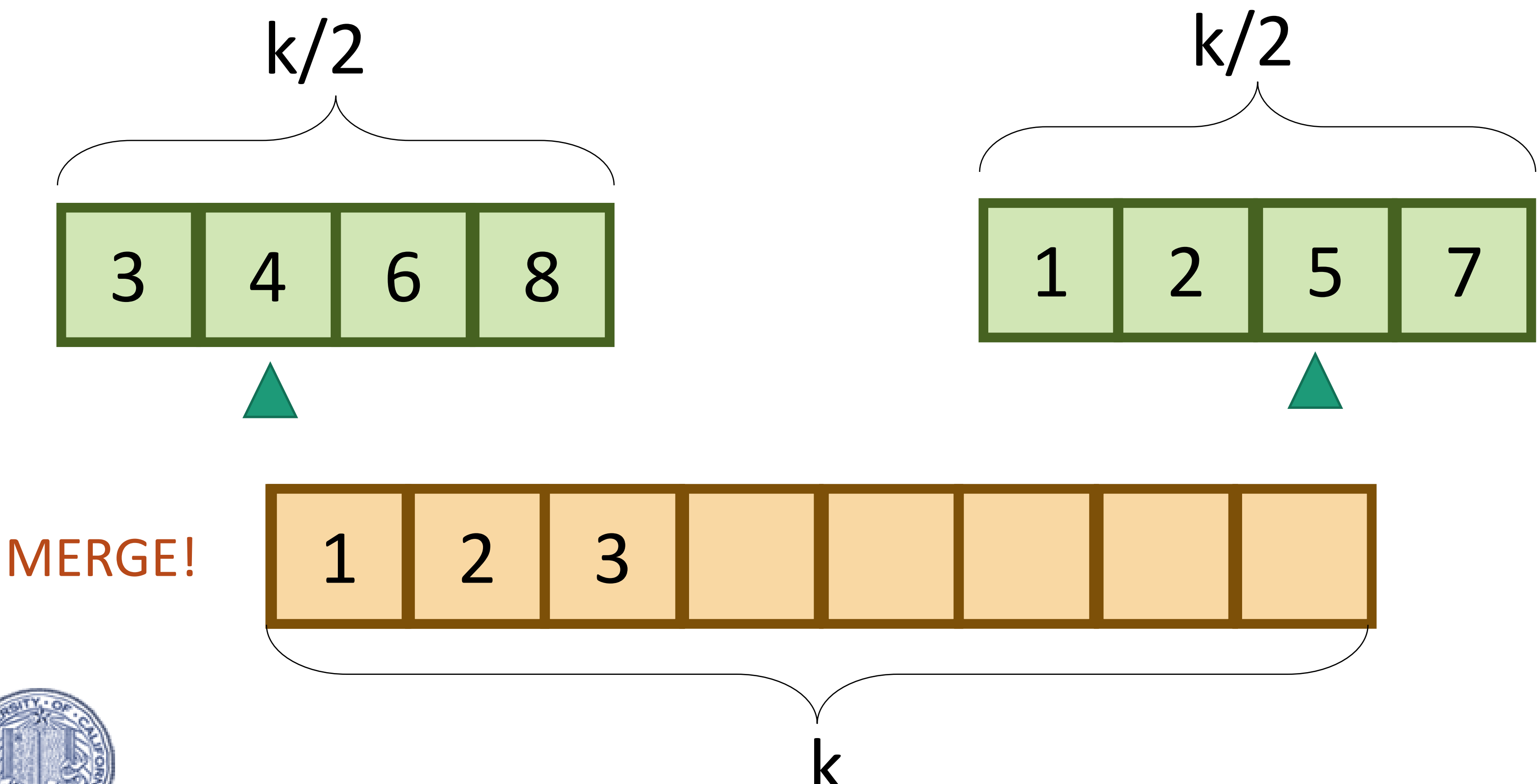
How long does it take to MERGE?



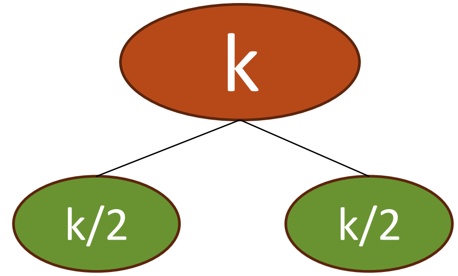


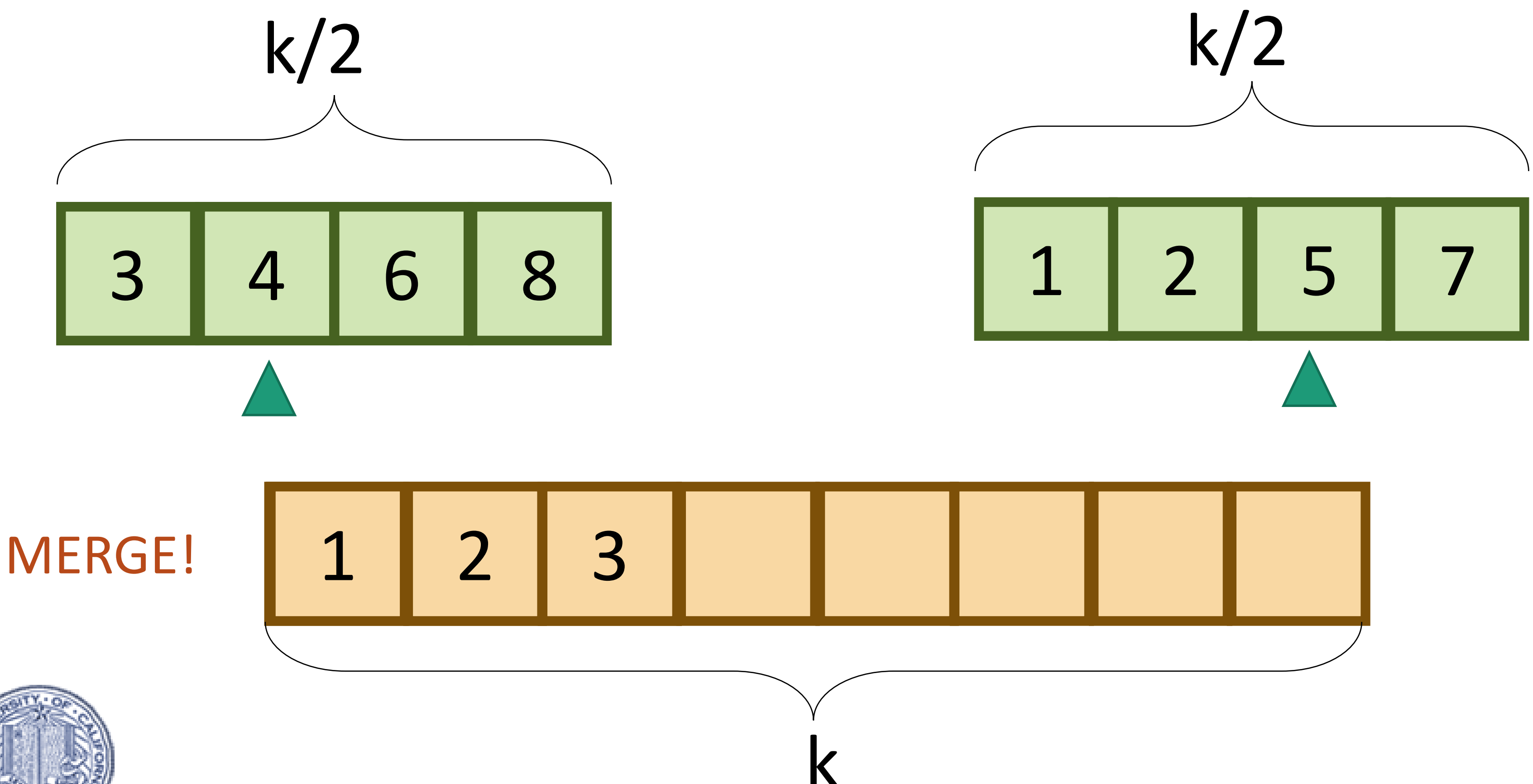
How long does it take to MERGE?



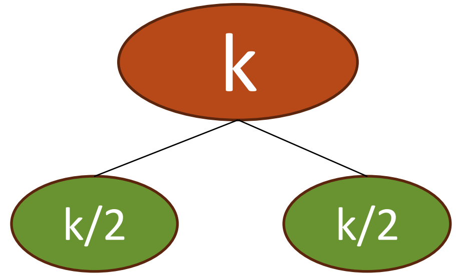


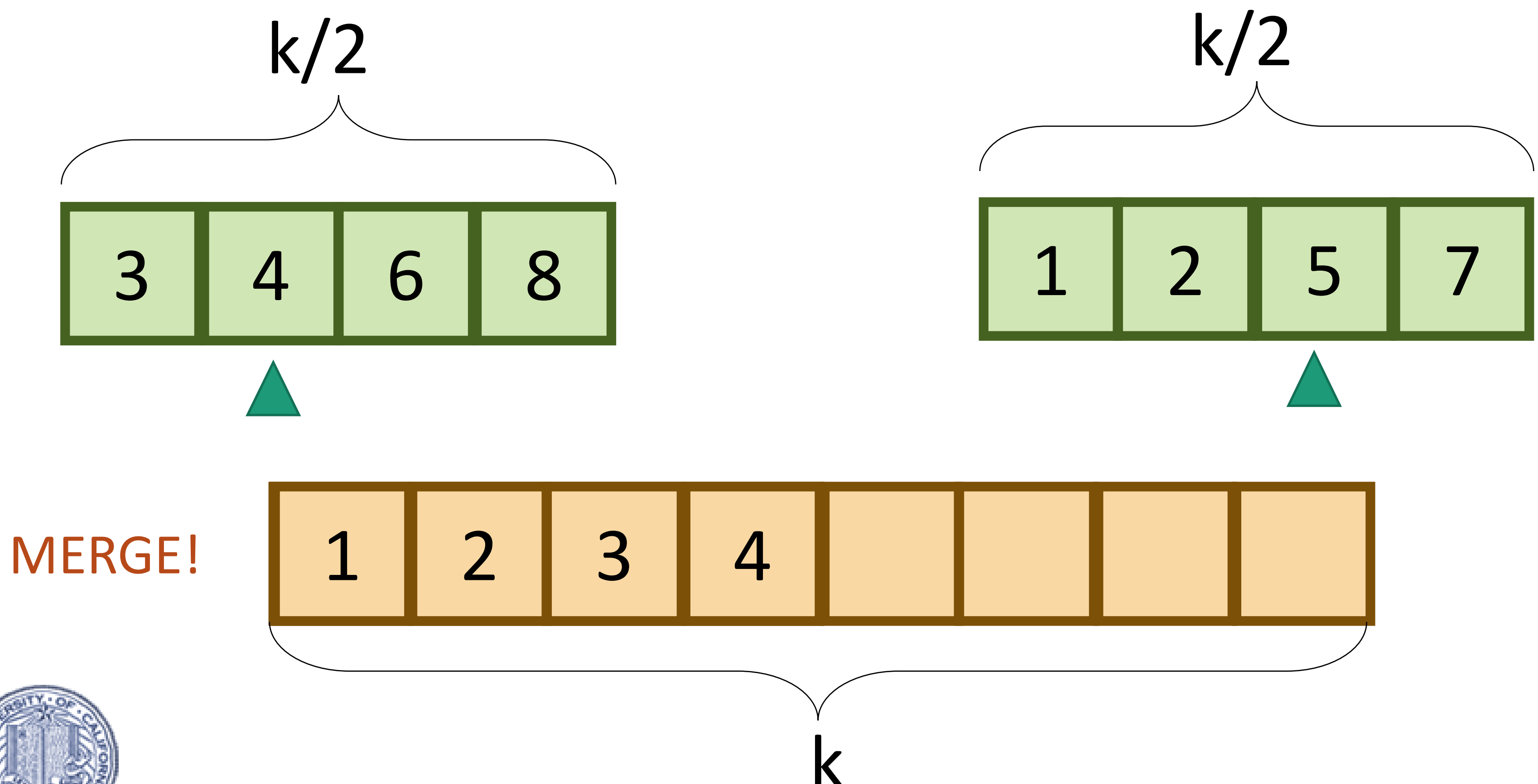
How long does it take to MERGE?



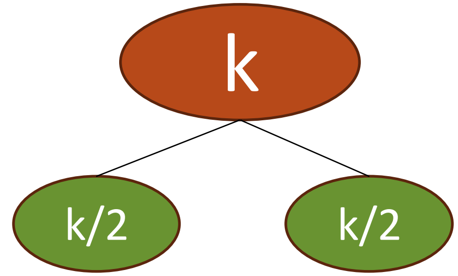


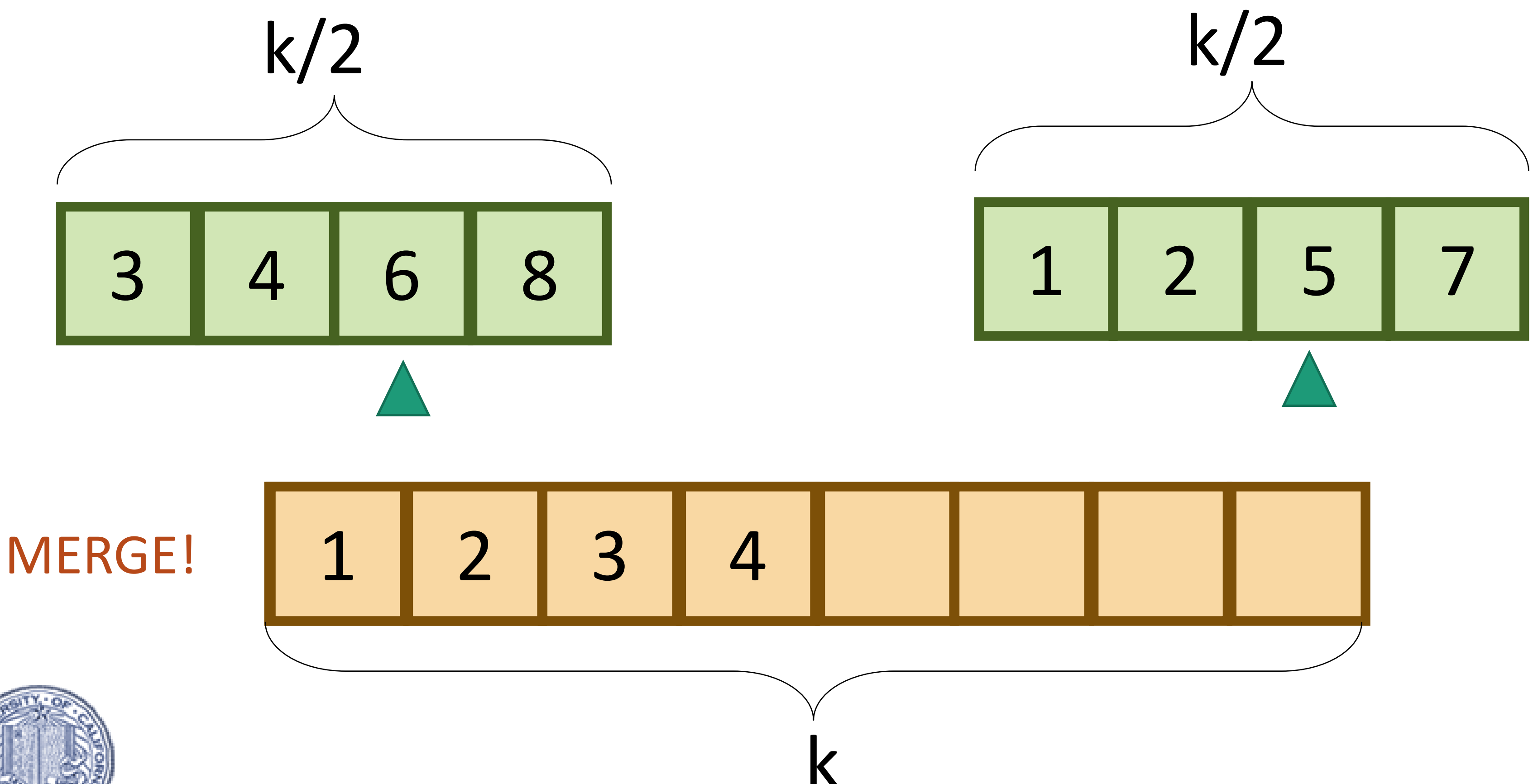
How long does it take to MERGE?



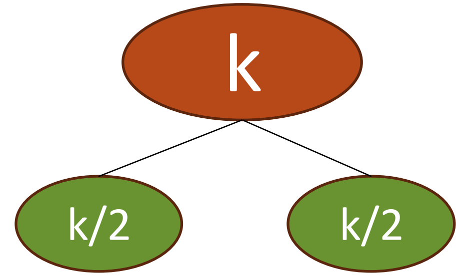


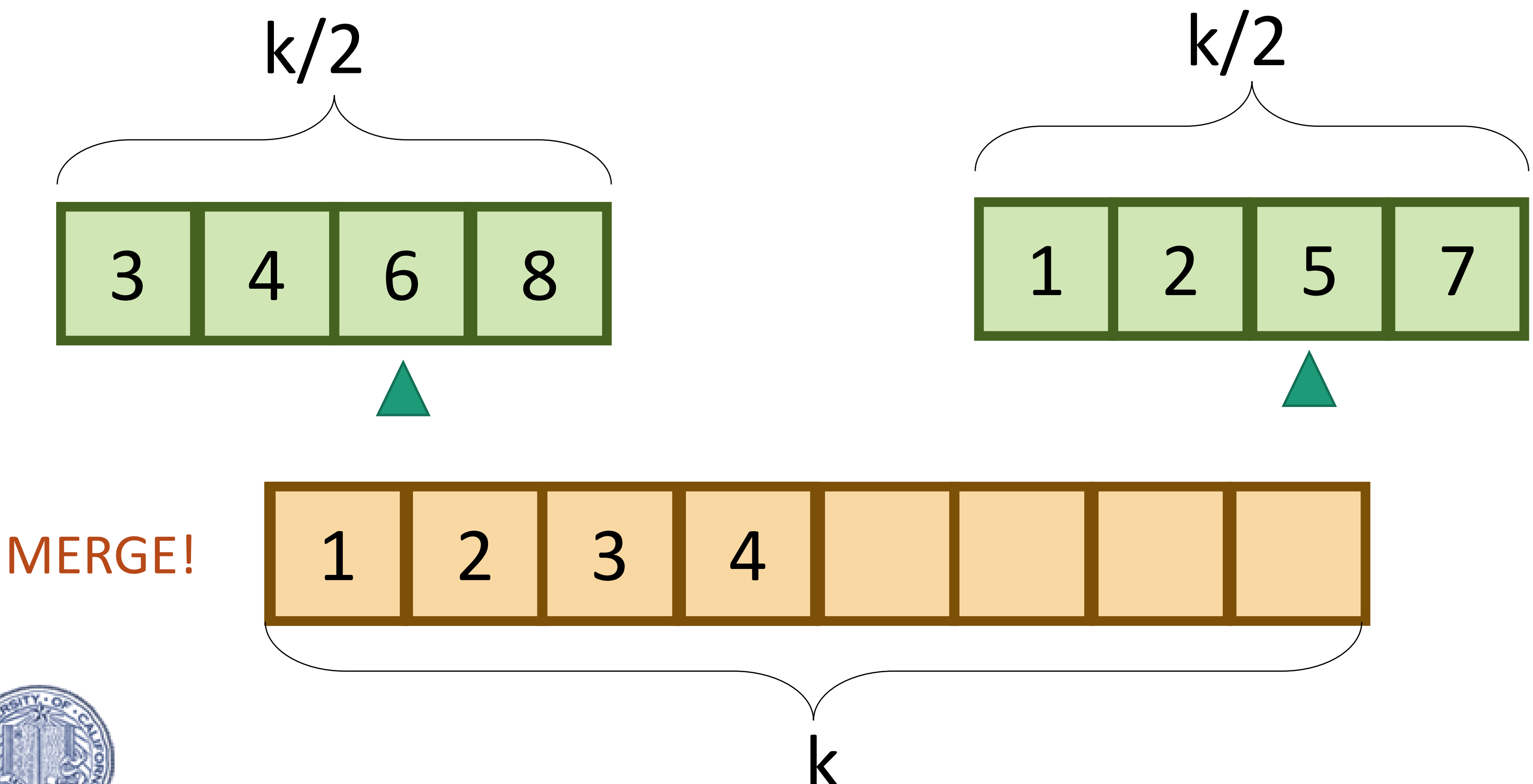
How long does it take to MERGE?



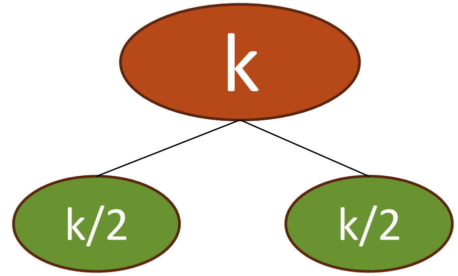


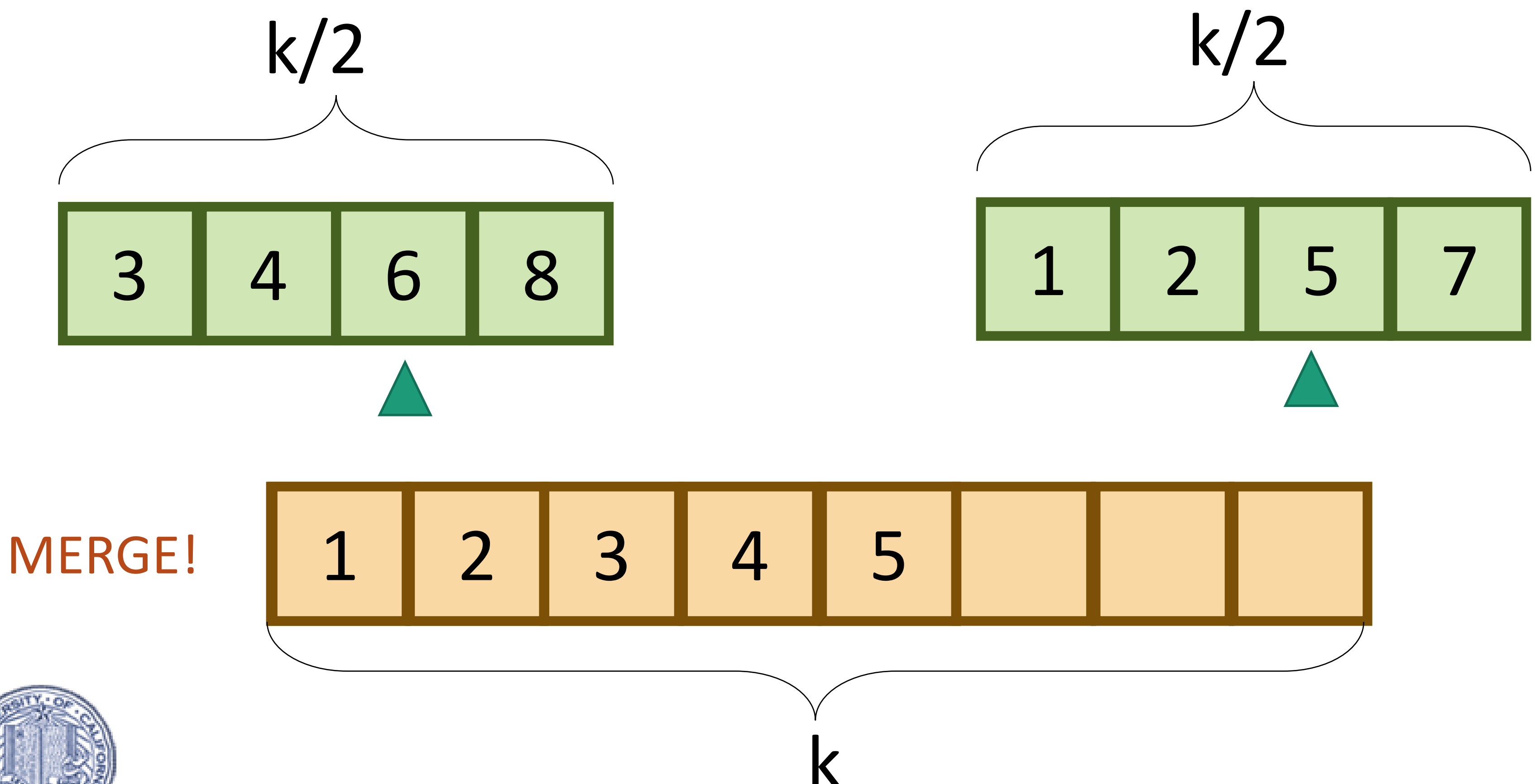
How long does it take to MERGE?



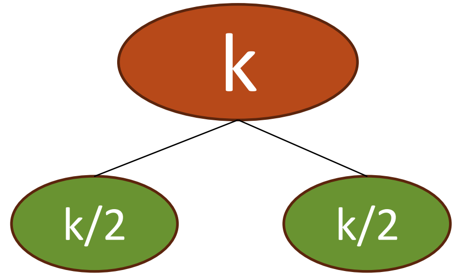


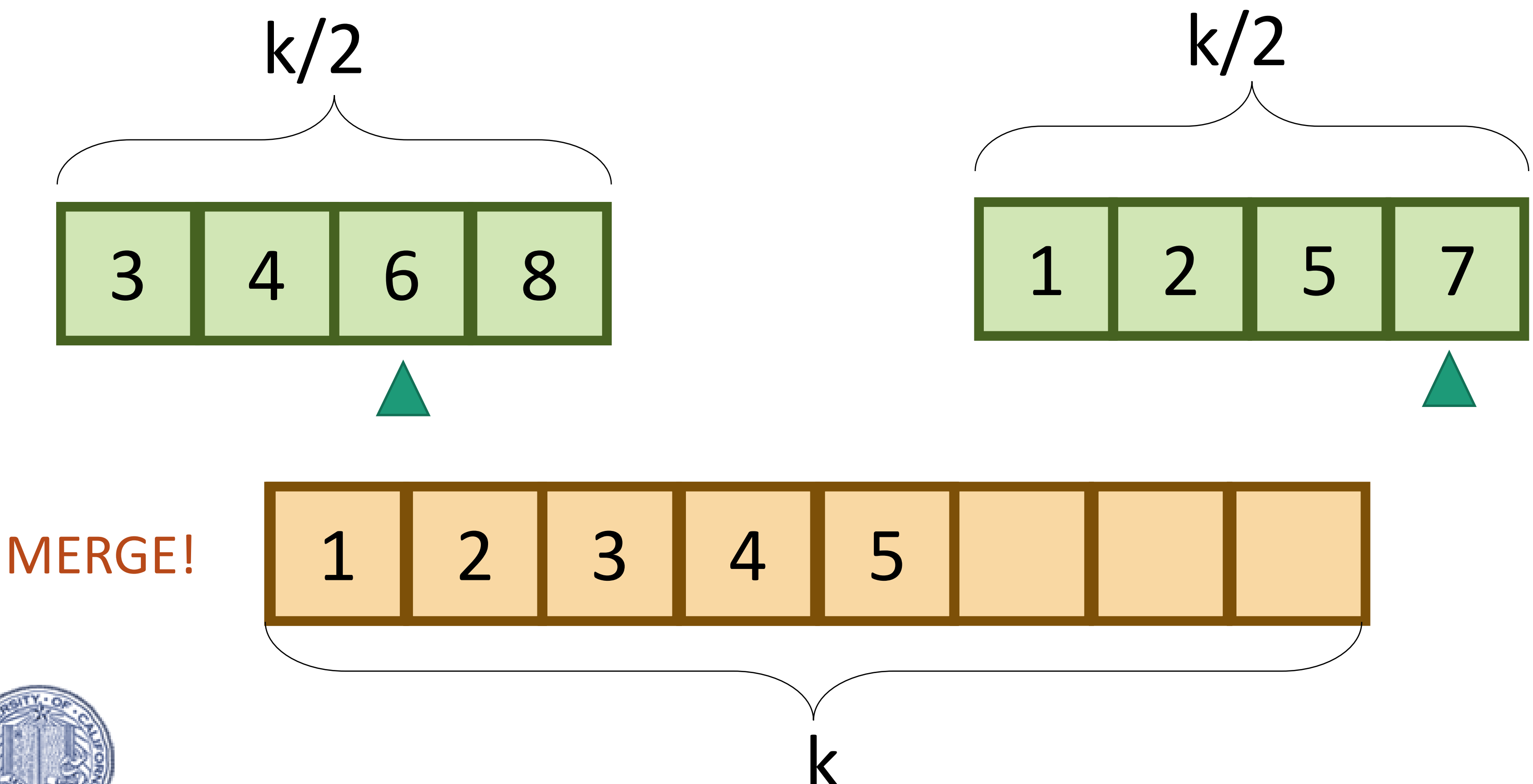
How long does it take to MERGE?



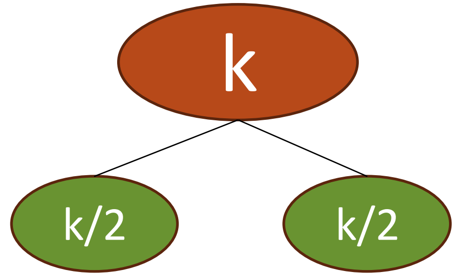


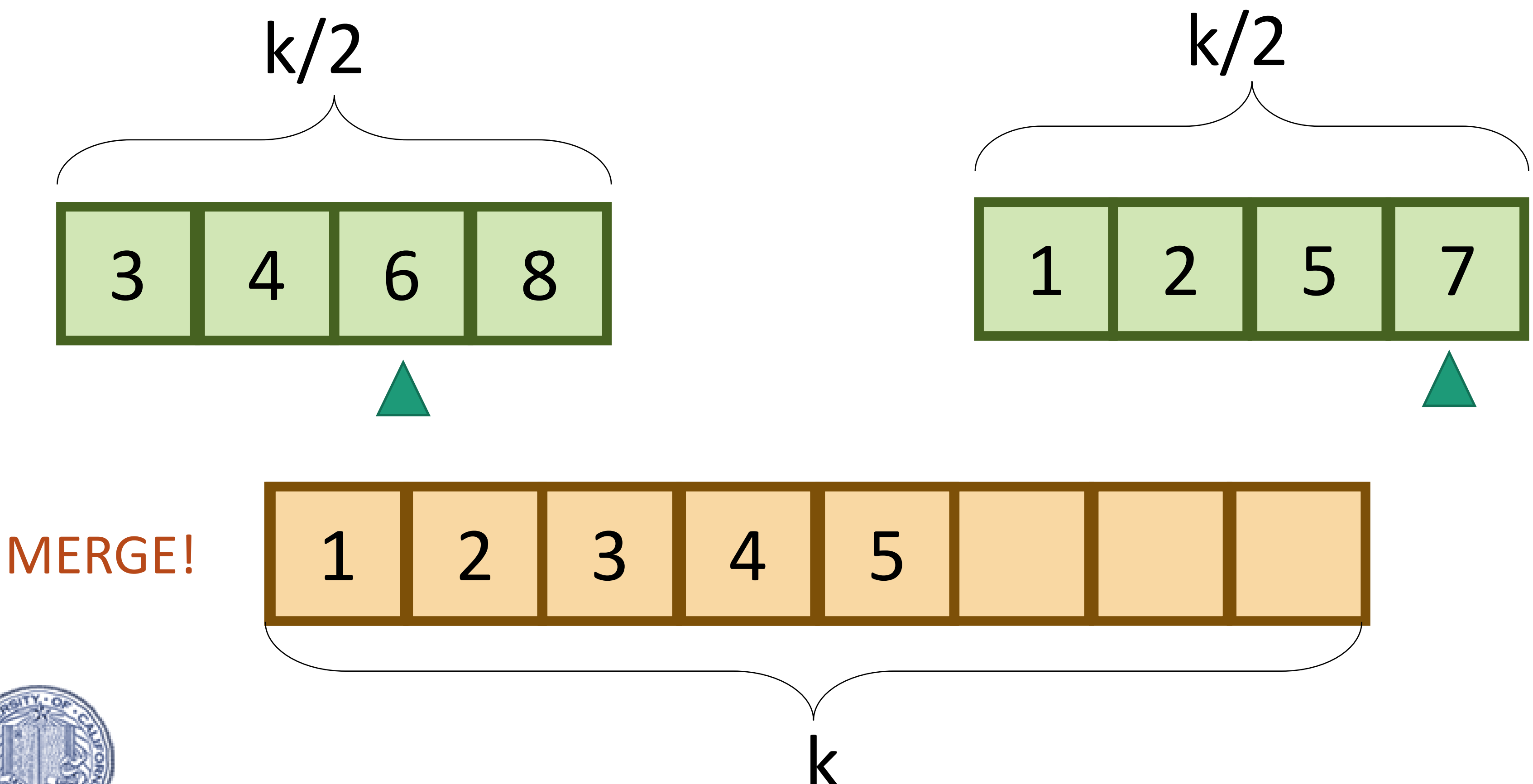
How long does it take to MERGE?



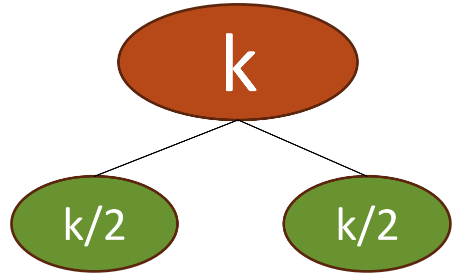


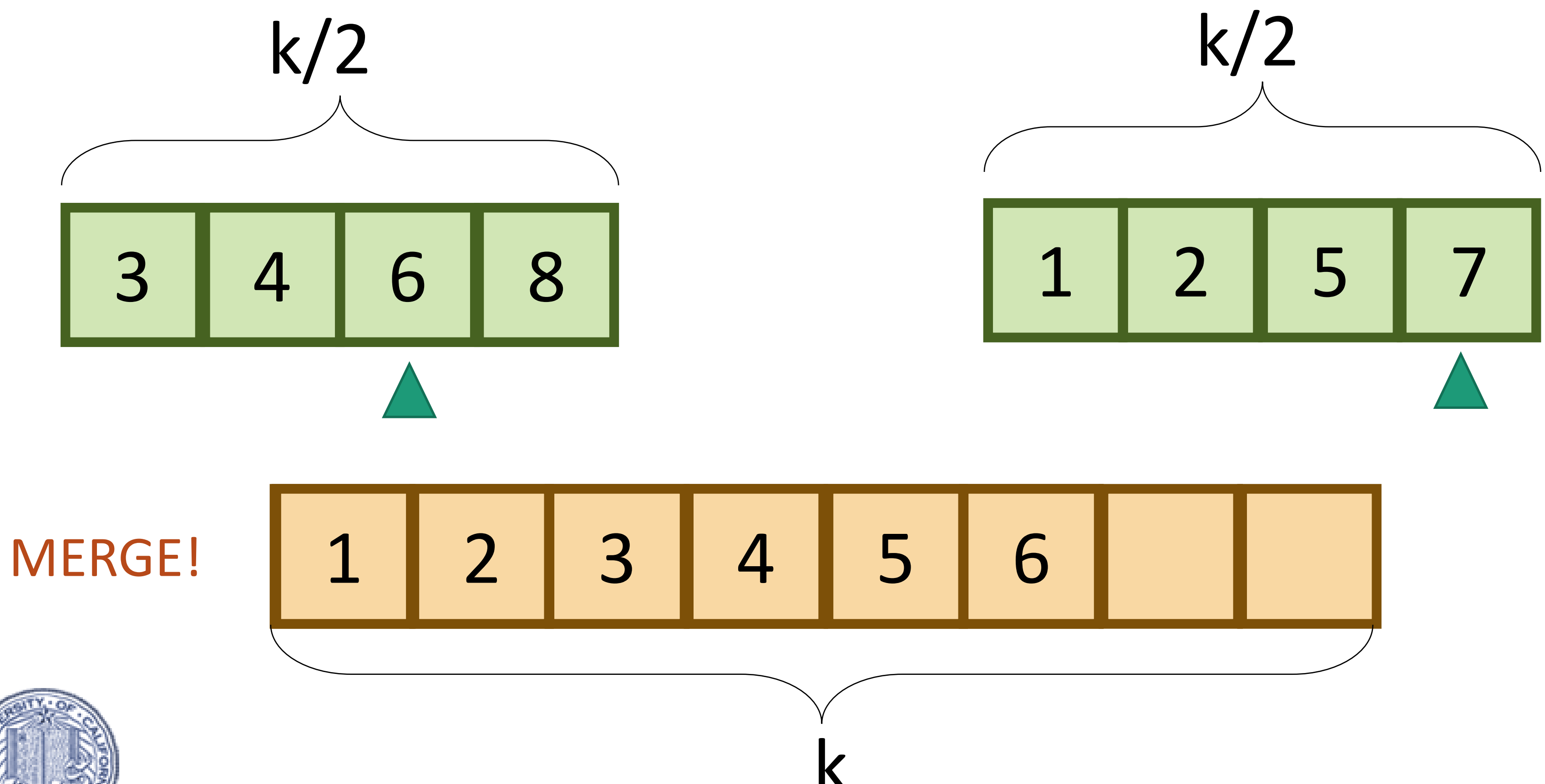
How long does it take to MERGE?



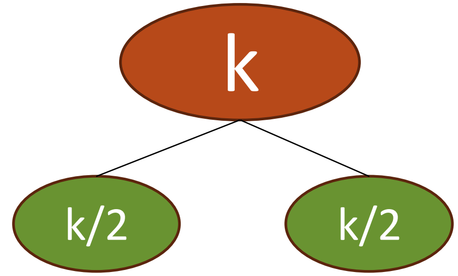


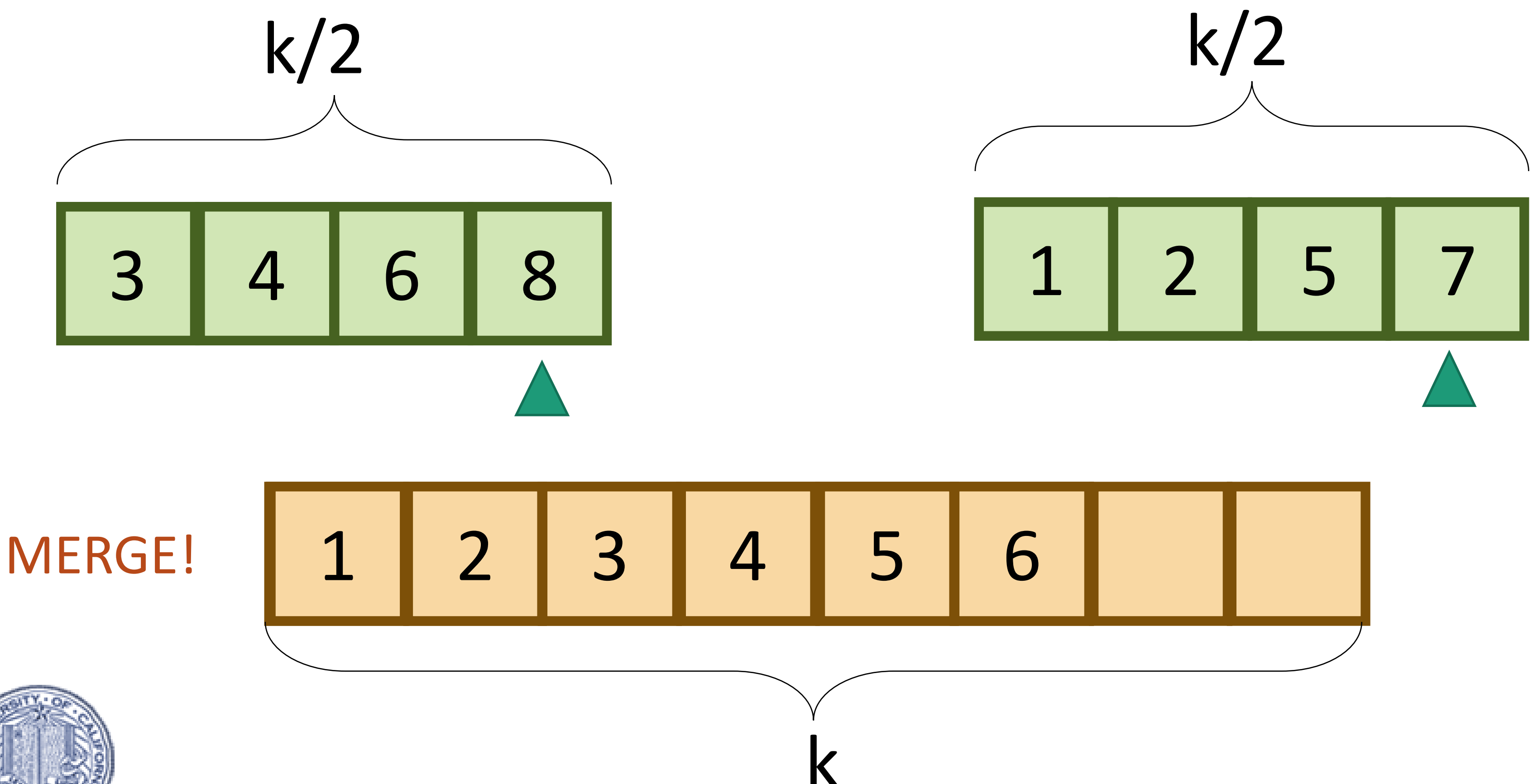
How long does it take to MERGE?



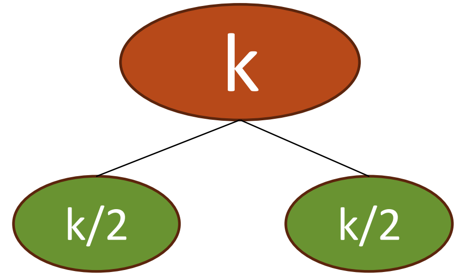


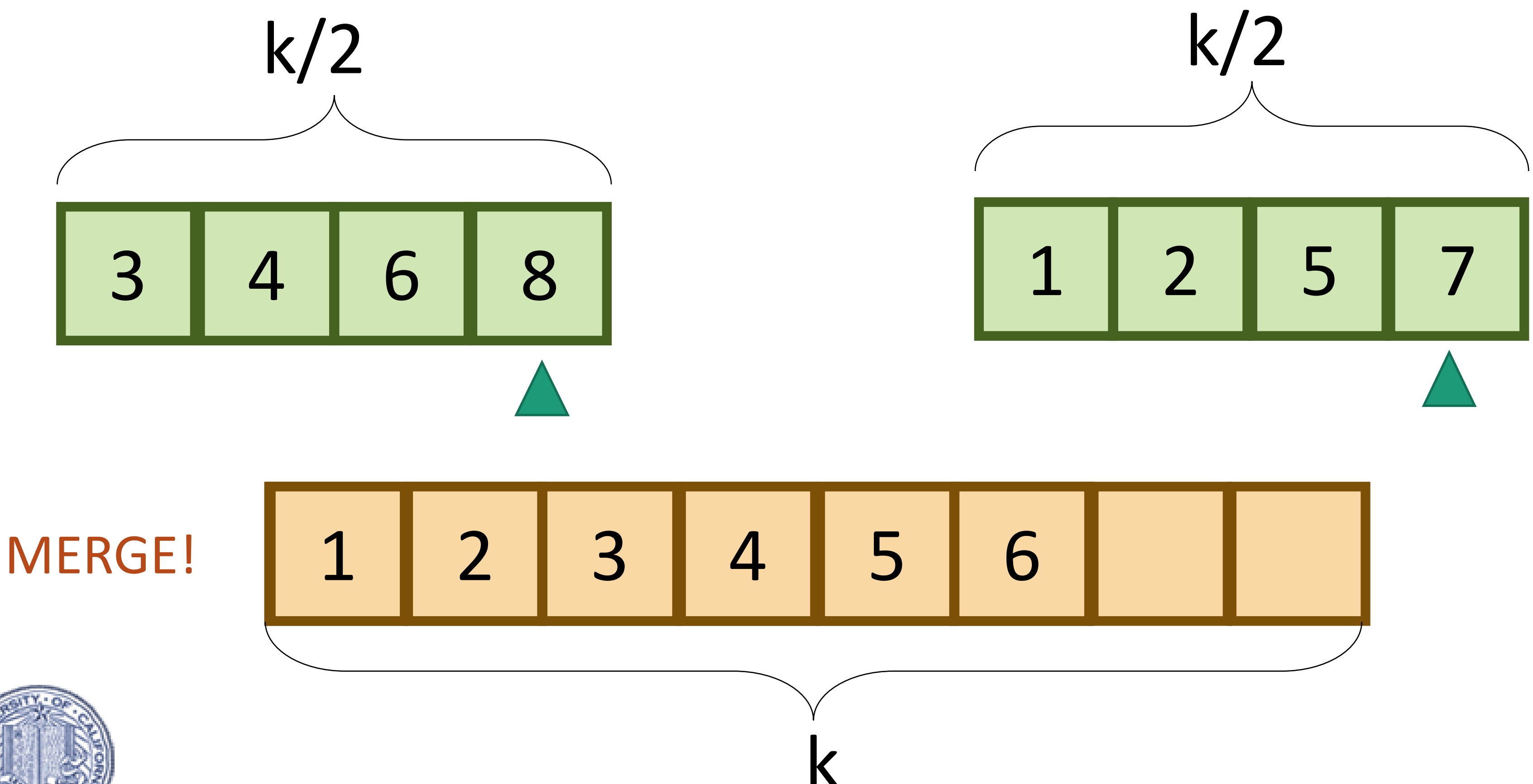
How long does it take to MERGE?



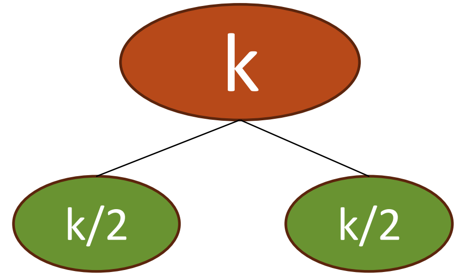


How long does it take to MERGE?

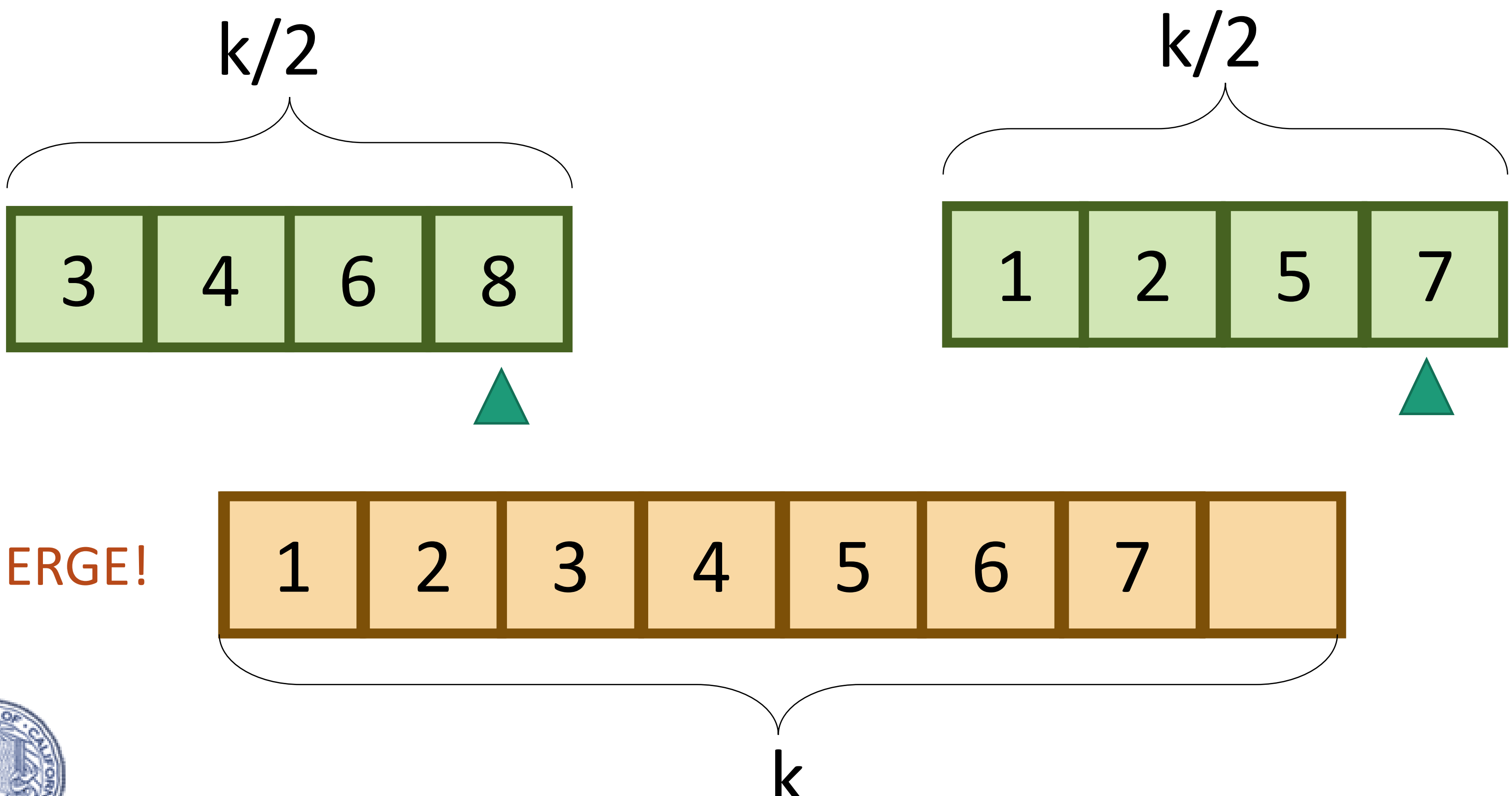




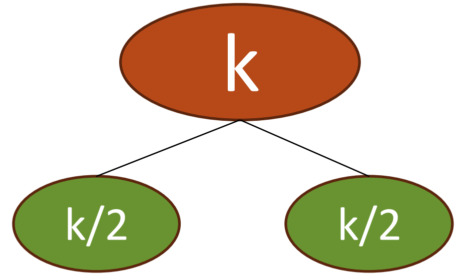
How long does it take to MERGE?

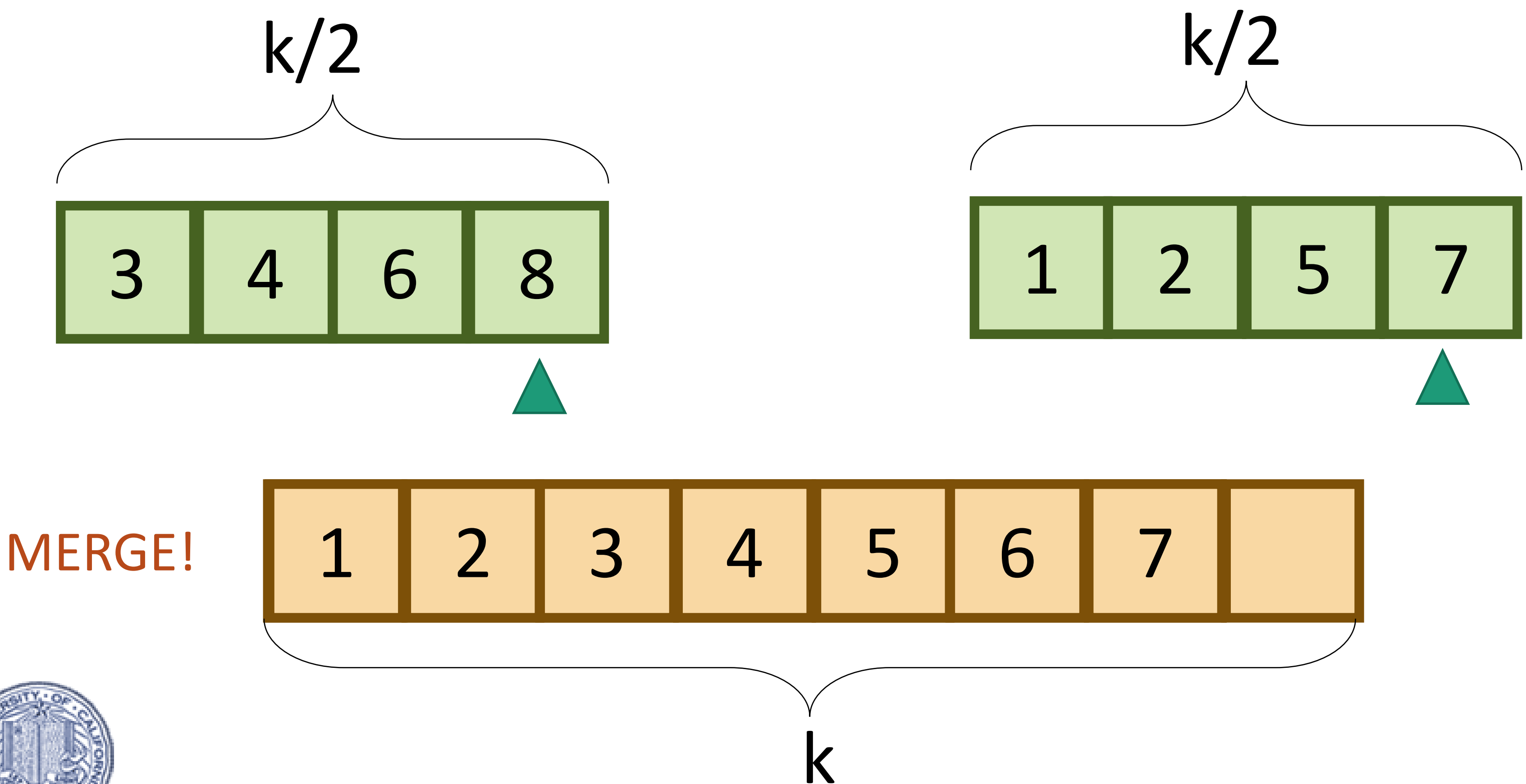


MERGE!

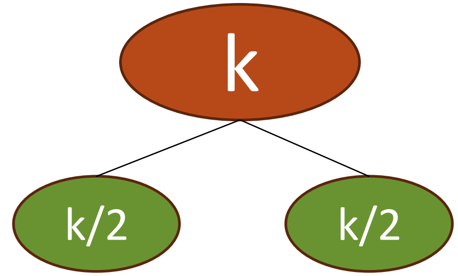


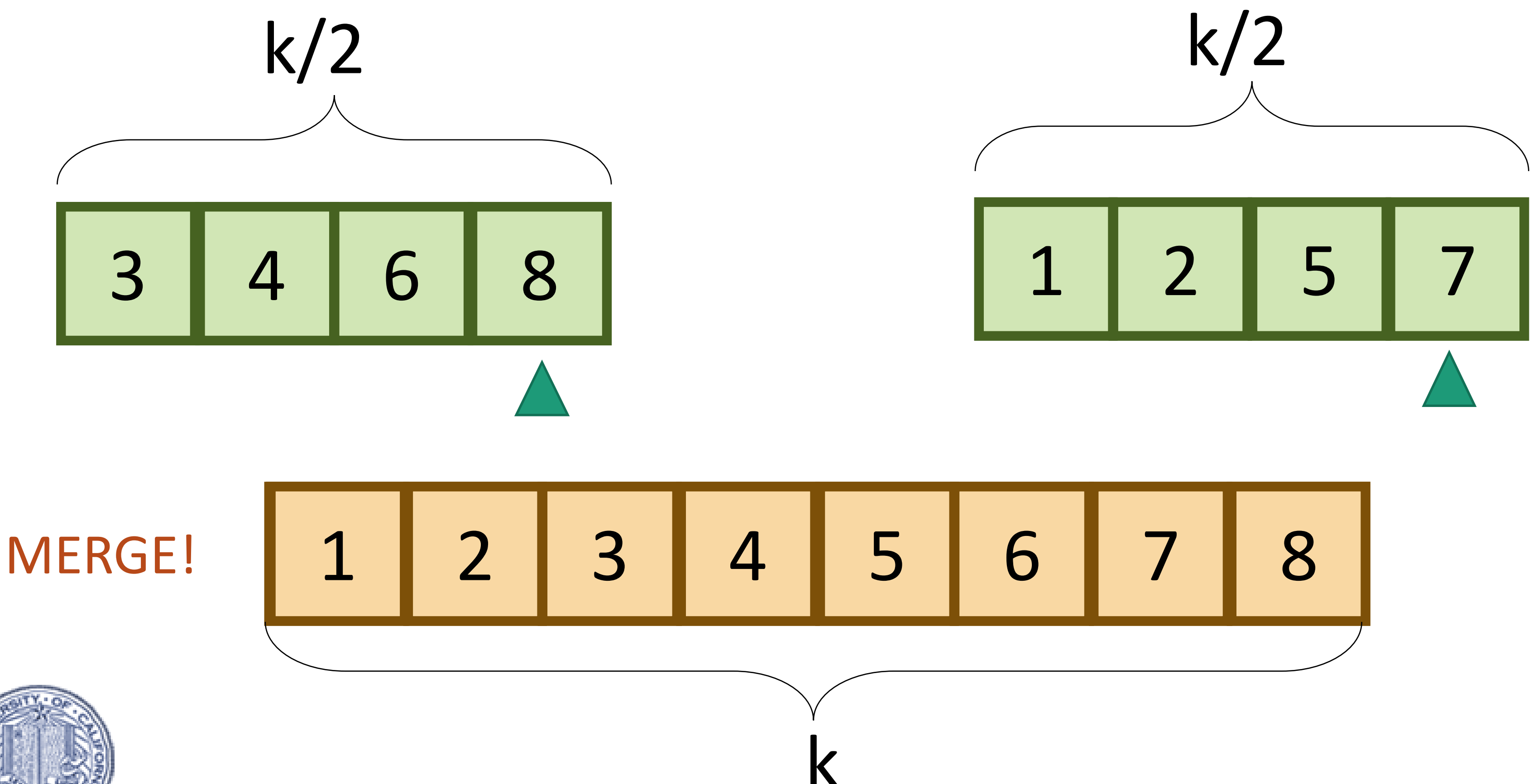
How long does it take to MERGE?



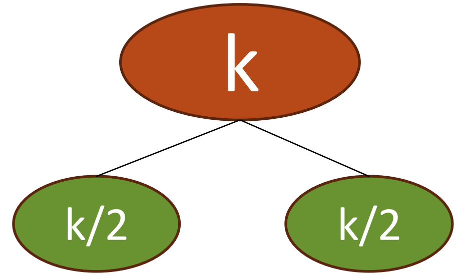


How long does it take to MERGE?

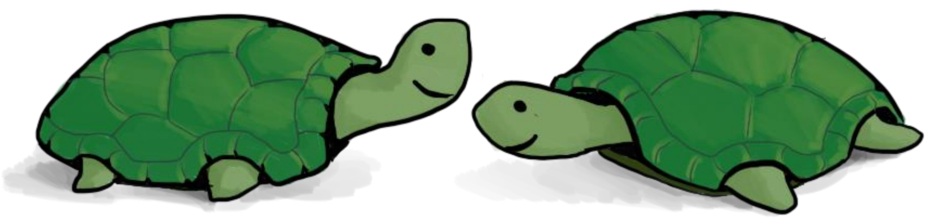




How long does it take to MERGE?



About how many operations does it take to run MERGE on two lists of size k/2?



Think-Pair-Share Terrapins

Operations within a subproblem of size k

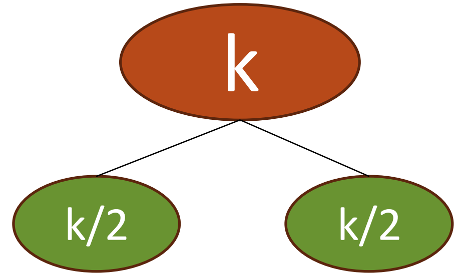
THIS SLIDE SKIPPED IN CLASS. It's here just in case you are curious how Lucky got "11". (But, we will see later why the number 11 doesn't really matter!)

The moral of the story is that **you don't need to pay attention to this slide!**

- Get the length of A (one op)
- Compare that length to 1 (one op)
- Initialize an array of size k (k ops)
- Pass L and R into MergeSort
  - If we implement this intelligently, this means assigning two pointers.
- Assign two initial pointers, one to each list.
- For k iterations:
  - Assign one pointer in the list you are writing into.
  - Write one value in the list you are writing into.
  - Increment two pointers

So this is $1 + 1 + k + 2 + 2 + k(1 + 1 + 2) = 5k + 6 \leq 11k$ as long as $k \geq 1$.

How long does it take to MERGE?



- 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…

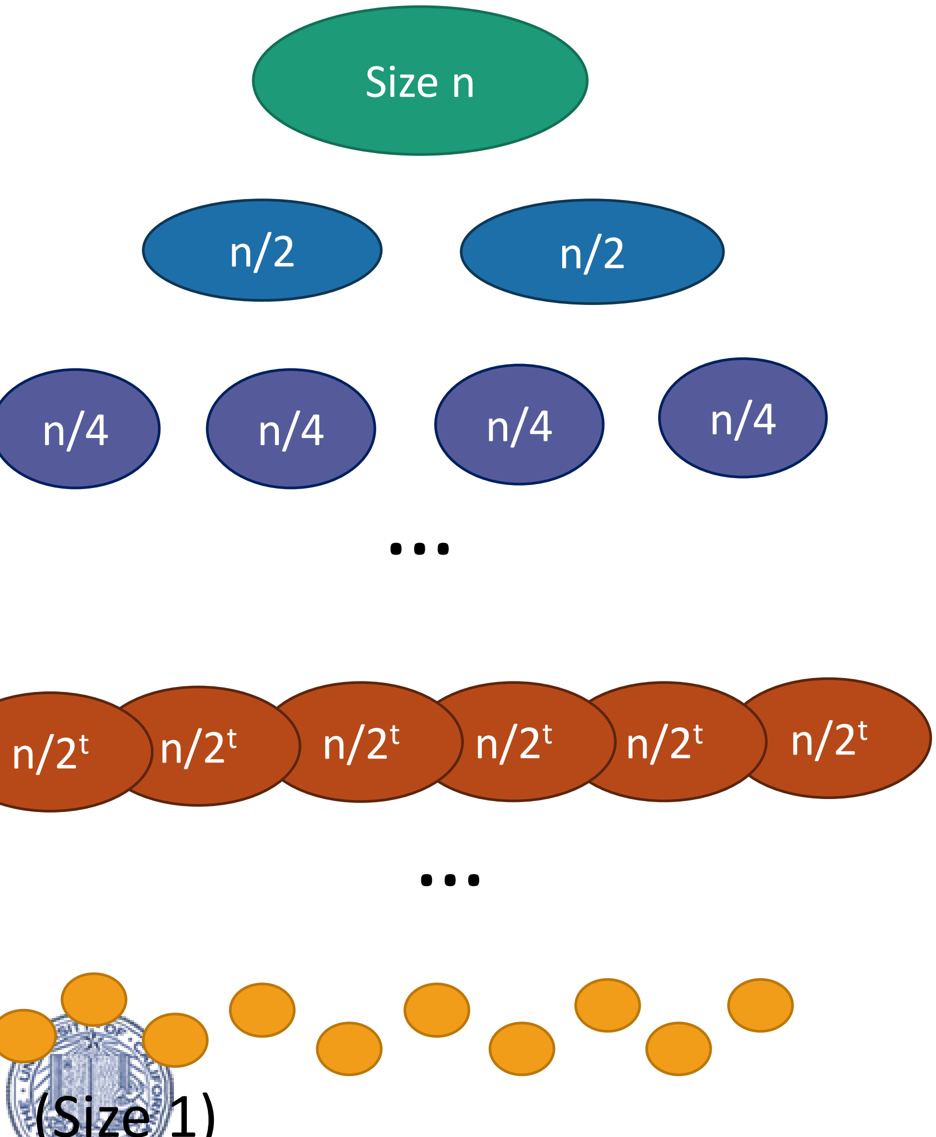
Plucky the Pedantic Penguin

Let’s say no more than 11k operations.

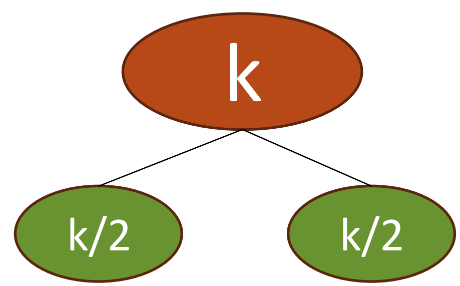
There’s a hidden slide which sort of explains this number “11,” but it’s a bit silly and we’ll see in a little by why it doesn’t matter.

Lucky the lackadaisical lemur

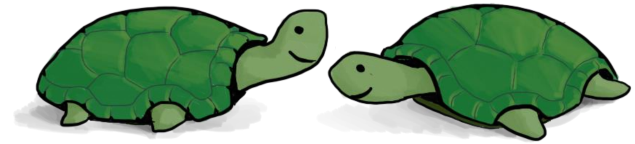
Recursion tree



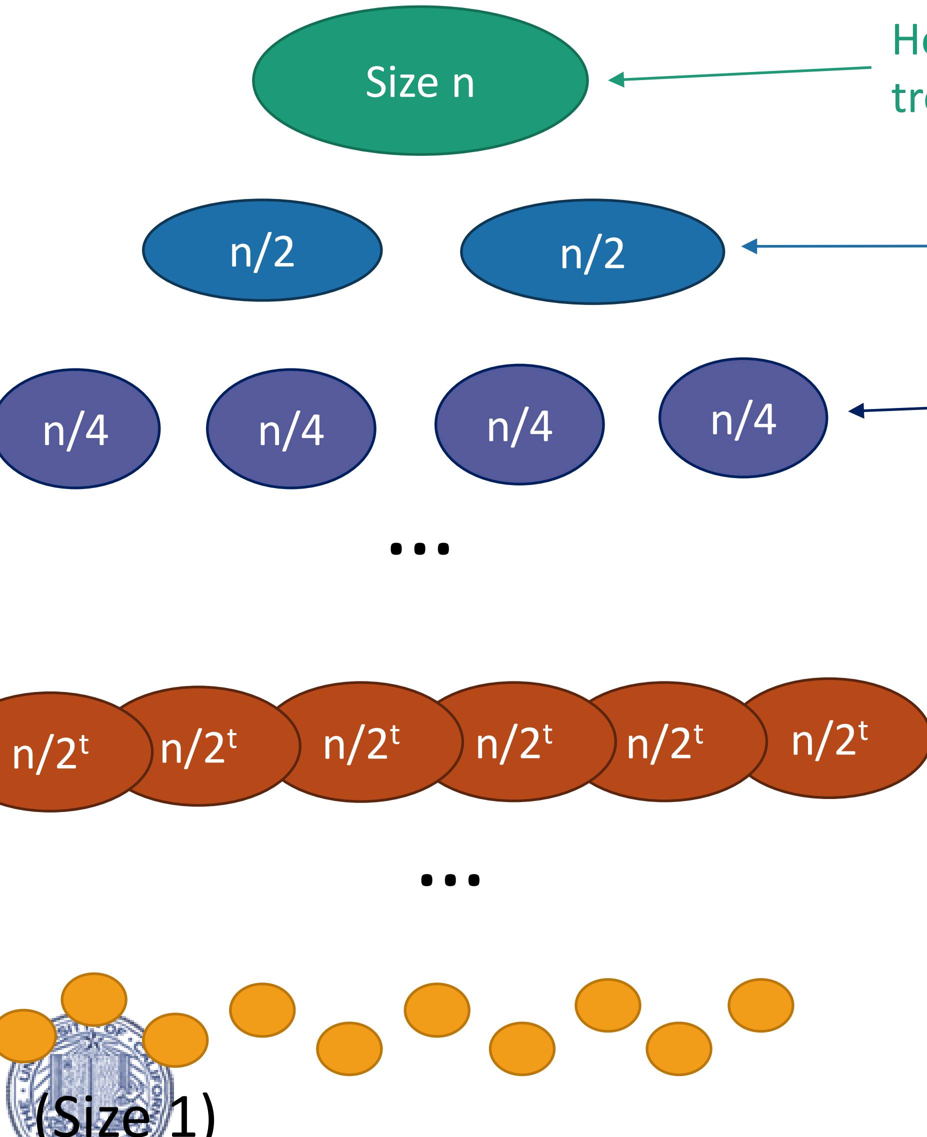
There are 11k operations done at this node.



Recursion tree



Think, Pair, Share!



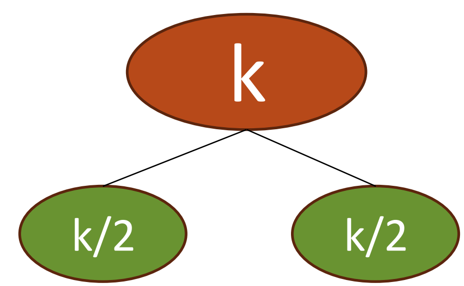
How many operations are done at this level of the tree? (Just MERGE-ing subproblems).

How about at this level of the tree? (between both n/2-sized problems)

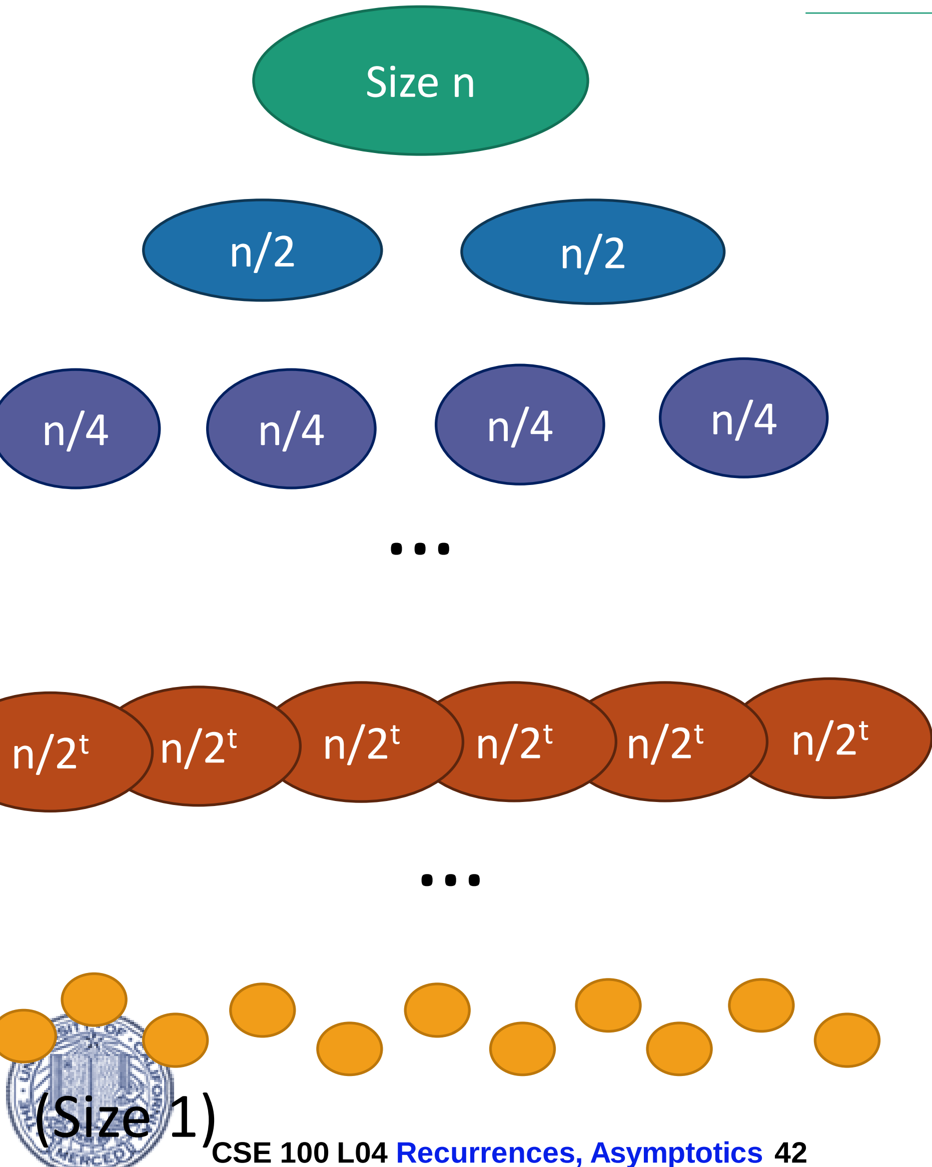
This level?

This level?

There are 11k operations done at this node.



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
...			Note: At the lowest level we only have two operations per problem, to get the length of the array and compare it to 1.
log(n)	n	1	$2n \leq 11n$

Total runtime…

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

That was the claim!

What have we learned?

- MergeSort correctly sorts a list of n integers in at most $11n(\log(n) + 1)$ operations.

A few reasons to be grumpy

- Sorting

1	2	3	4	5	6	7	8

should take zero steps…

- What’s with this 11k bound?
  - You (Lucky) made that number “11” up.
  - Different operations don’t take the same amount of time.

Wrap up

- Sorting: InsertionSort and MergeSort
- Analyzing correctness of iterative + recursive algs
  - Via "loop invariant" and induction
- Analyzing running time of recursive algorithms
  - By writing out a tree and adding up all the work done.

Today (Part 2)

- How do we measure the runtime of an algorithm?
  - Worst-case analysis
  - Asymptotic Analysis
- Recurrence Relations!
  - How do we calculate the runtime a recursive algorithm?
- The Master Method
  - A useful theorem so we don’t have to answer this question from scratch each time.

Recall …

- We analyzed INSERTION SORT and
- MERGESORT.
- They were both correct!
- INSERTION SORT took time about n^2
- MERGESORT took time about $n\log(n)$.

A few reasons to be grumpy

- Sorting

1	2	3	4	5	6	7	8

should take zero steps…why $n\log(n)$??

- What's with this $T(\text{MERGE}) < 11n$?

How we will deal with **grumpiness**

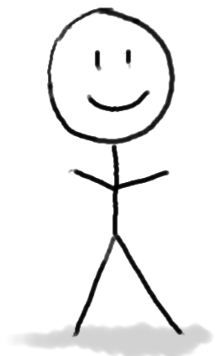
- Take a deep breath…
- Worst case analysis
- Asymptotic notation

Worst-case analysis

Sorting a sorted list should be fast!!

1	2	3	4	5	6	7	8

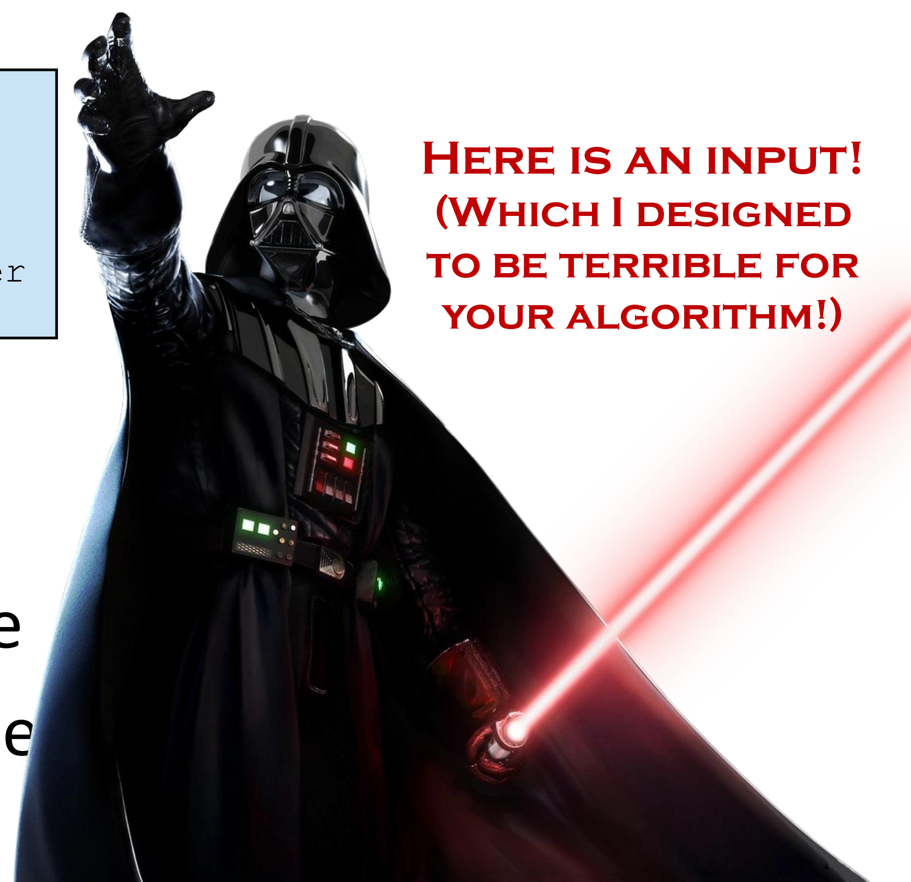
The "running time" for an algorithm is its running time on the worst possible input.



Here is my algorithm!

```
Algorithm:
   Do the thing
   Do the stuff
   Return the answer
```

Algorithm designer



HERE IS AN INPUT!
(WHICH I DESIGNED TO BE TERRIBLE FOR YOUR ALGORITHM!)

- Pros: very strong guarantee
- Cons: very strong guarantee

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).

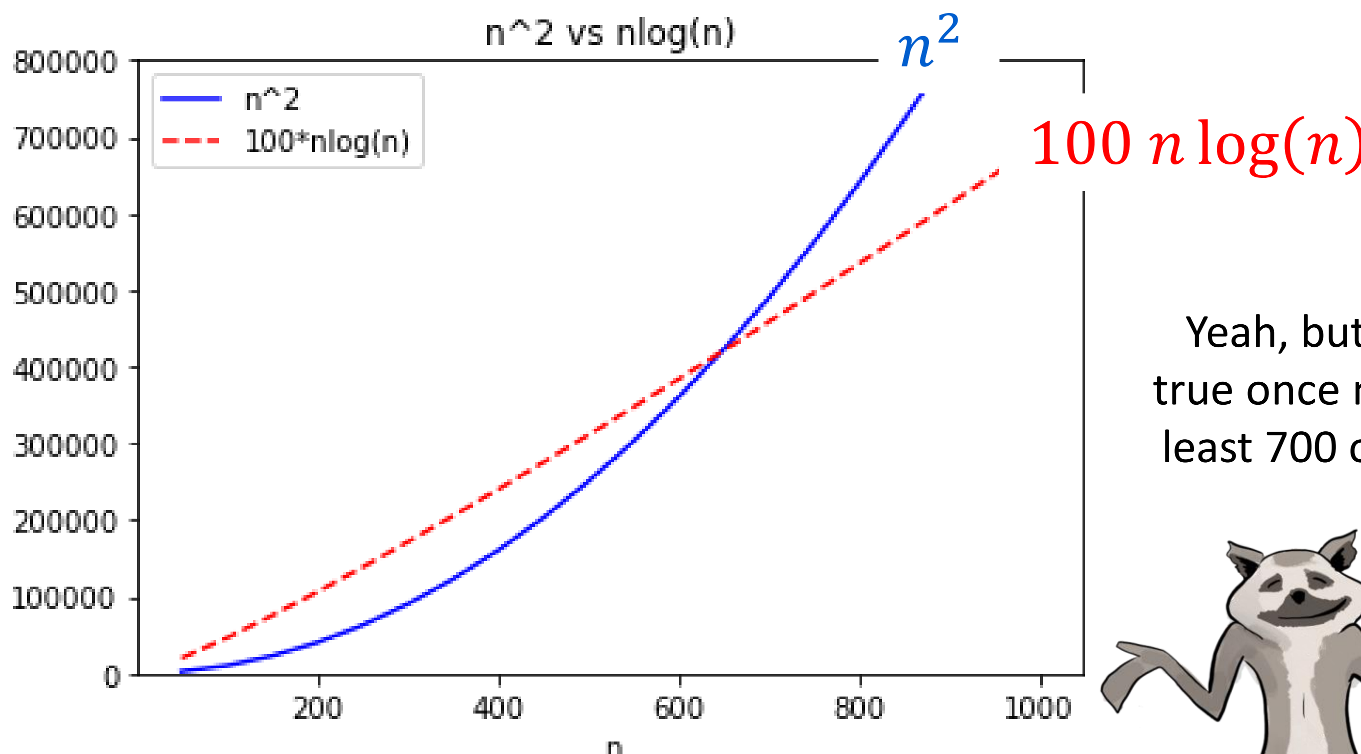
Informally….

(Only pay attention to the largest function of n that appears.)

Number of operations	Asymptotic Running Time
$\frac{1}{10} \cdot n^2 + 100$	$O(n^2)$
$0.063 \cdot n^2 - .5\,n + 12.7$	$O(n^2)$
$100 \cdot n^{1.5} - 10^{10000}\sqrt{n}$	$O(n^{1.5})$
$11 \cdot n\log(n) + 1$	$O(n\log(n))$

We say this algorithm is “asymptotically faster” than the others.

So $100\, n \log(n)$ operations is "better" than n^2 operations?



But when n=200, that's not true at all!

Yeah, but it's true once n is at least 700 or so.

Asymptotic Analysis

How does the running time scale as n gets large?

One algorithm is "faster" than another if its runtime grows more "slowly" as n gets large.

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 ?!?!

Now for some definitions…

- Quick reminders:
  - ∃: "There exists"
  - ∀: "For all"
  - Example: ∀ students in CSE100, ∃ an algorithms problem that really excites the student.
  - Much stronger statement: ∃ an algorithms problem so that, ∀ students in CSE100, the student is excited by the problem.
- We're going to formally define an upper bound:
  - "$T(n)$ grows no faster than $f(n)$"

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

$O(\ldots)$ means an upper bound

- Let $T(n)$, $g(n)$ be functions of positive integers.
  - Think of $T(n)$ as a runtime: positive and increasing in n.
- We say “$T(n)$ is $O\big(g(n)\big)$” if $T(n)$ grows no faster than $g(n)$ as n gets large.
- Formally,

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

$$\Leftrightarrow$$

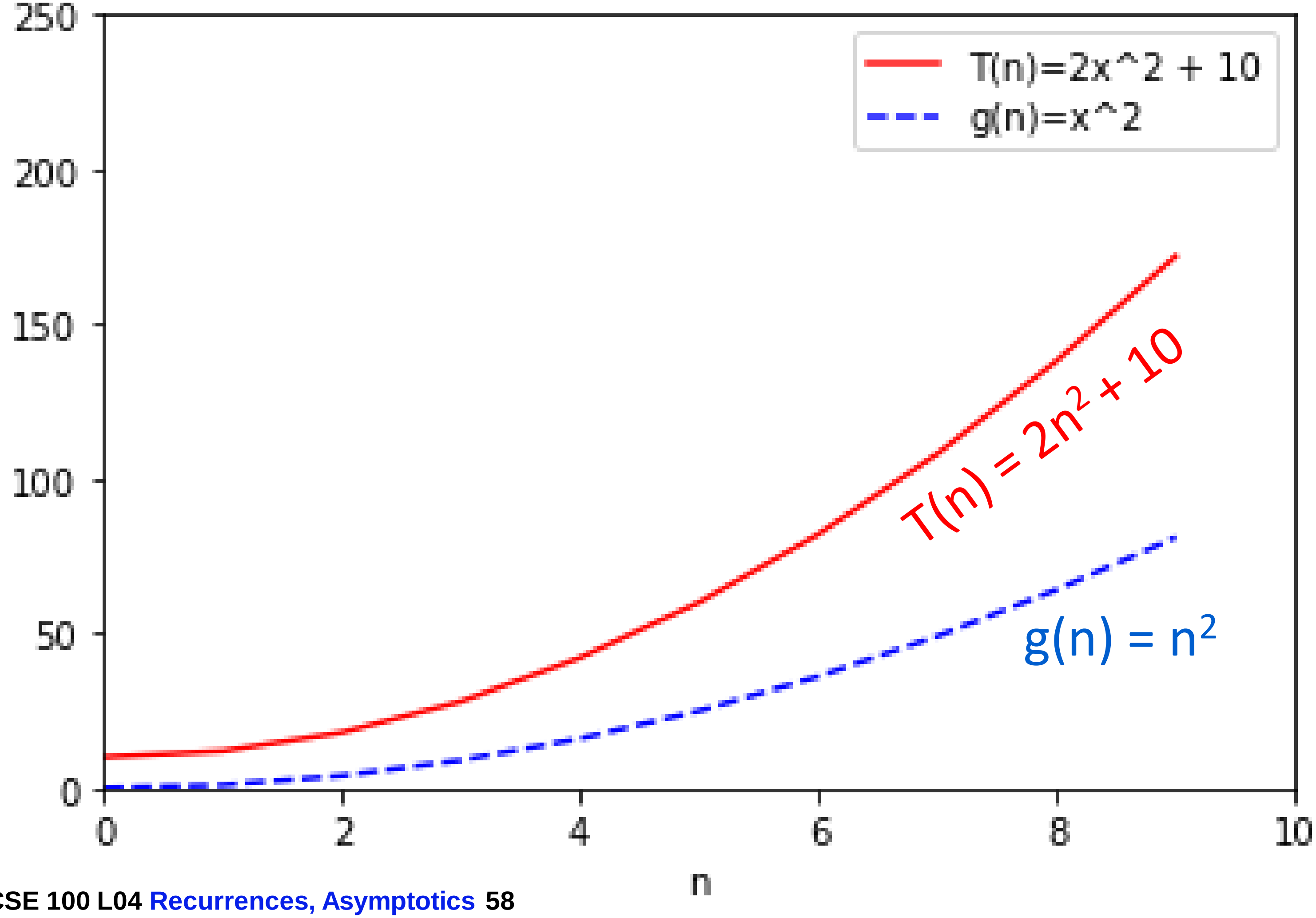
$$\exists c, n_0 > 0 \;\; s.t. \;\; \forall n \geq n_0,$$

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

Example

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

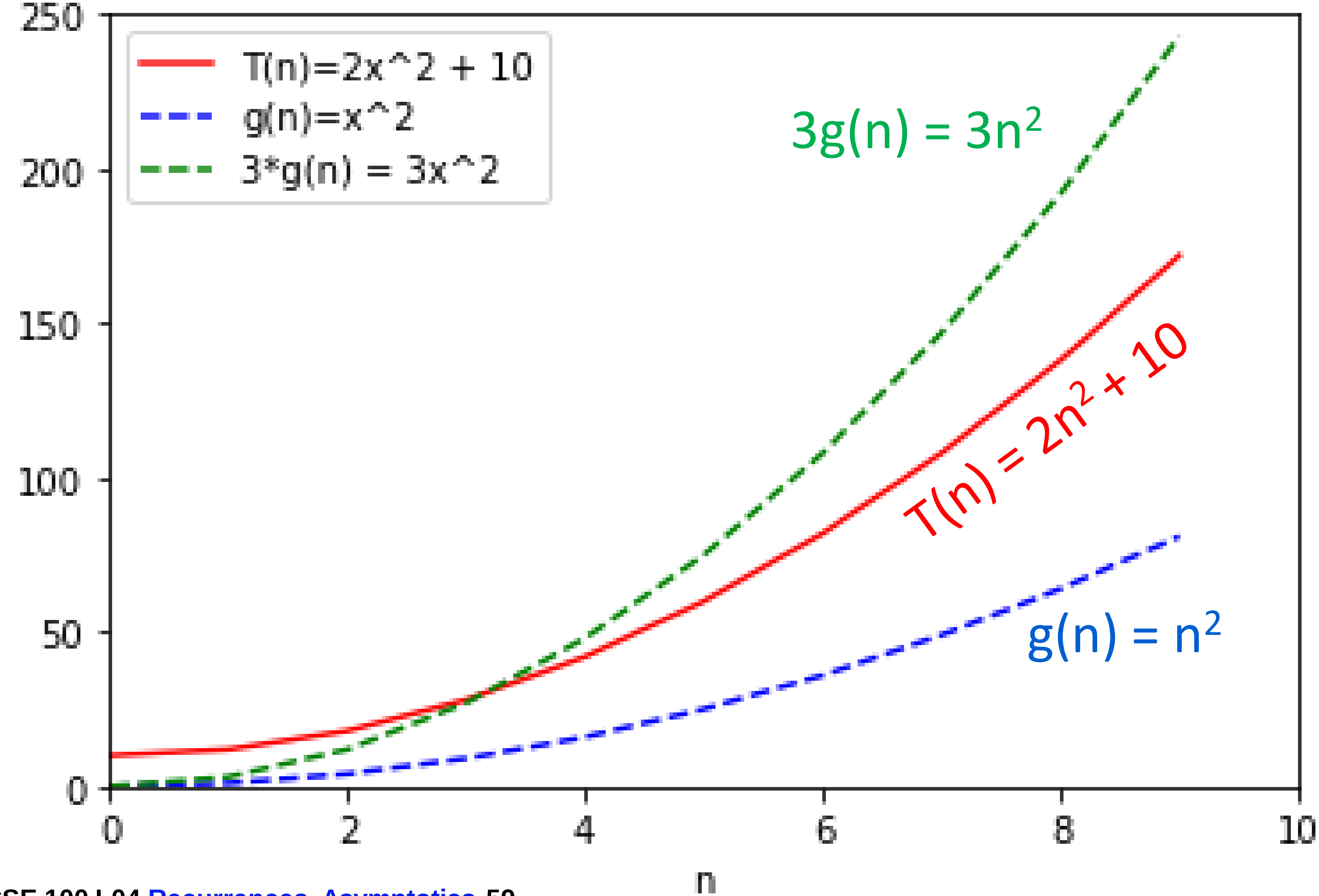
$$T(n) = O(g(n)) \Leftrightarrow \exists c, n_0 > 0 \ \ s.t. \ \ \forall n \geq n_0, \ 0 \leq T(n) \leq c \cdot g(n)$$



Example

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

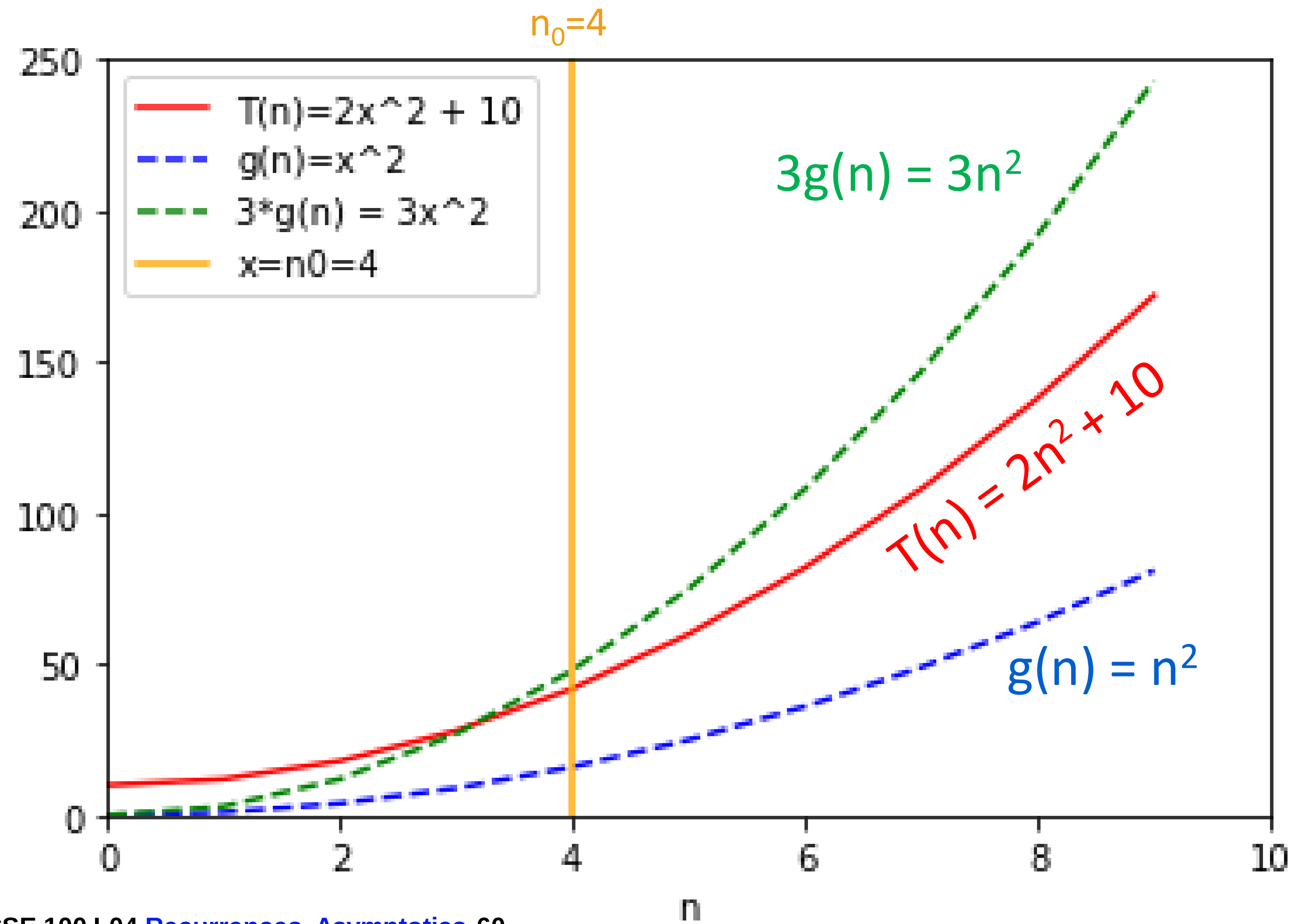
$T(n) = O(g(n))$
$\Leftrightarrow$
$\exists c, n_0 > 0 \;\; s.t. \;\; \forall n \geq n_0,$
$0 \leq T(n) \leq c \cdot g(n)$



Example

$$2n^2 + 10 = O\left(n^2\right)$$

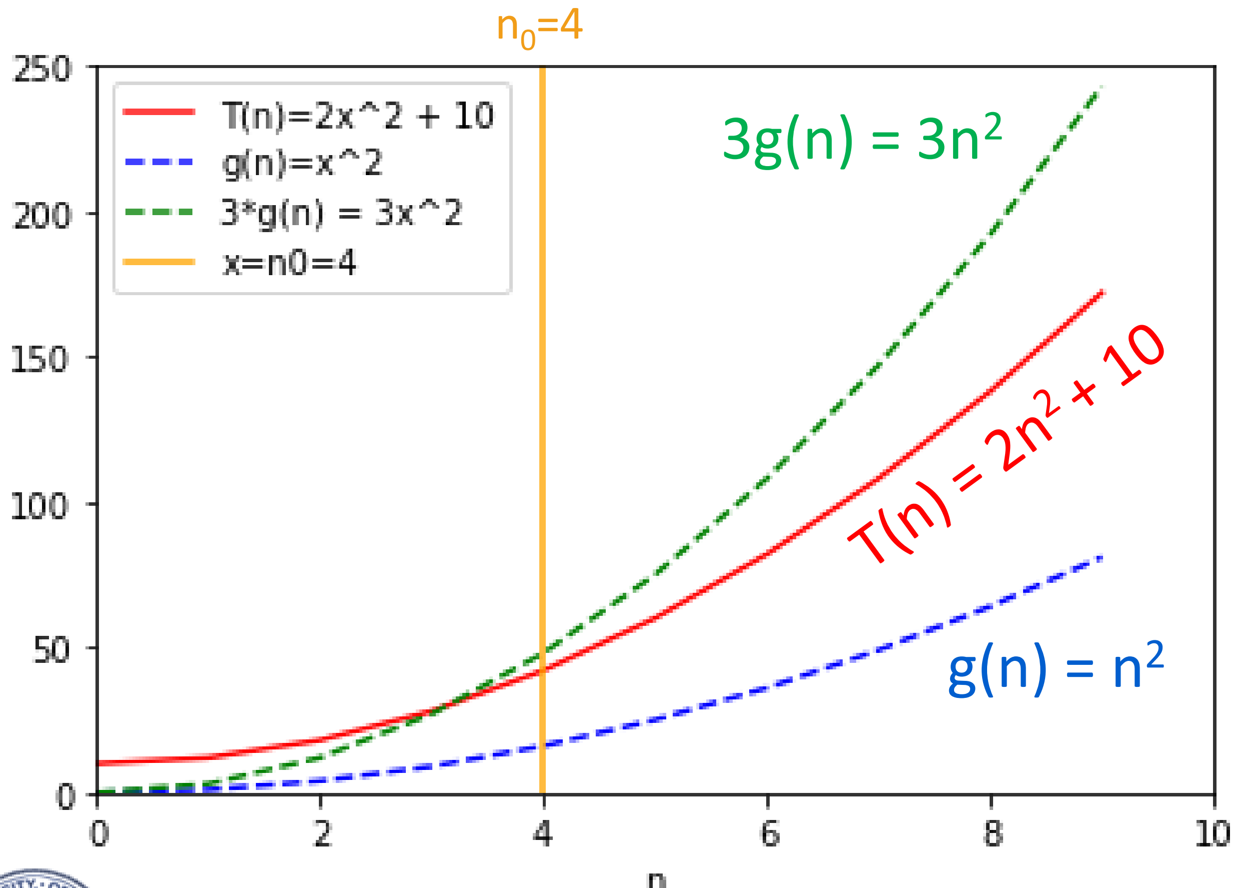
$$T(n) = O\big(g(n)\big)$$
$$\Leftrightarrow$$
$$\exists c, n_0 > 0 \;\; s.t. \;\; \forall n \geq n_0,$$
$$0 \leq T(n) \leq c \cdot g(n)$$



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:

$$\forall n \geq 4,$$

$$0 \leq 2n^2 + 10 \leq 3 \cdot n^2$$

In order to **formally prove** $2n^2 + 10 = O(n^2)$

- Choose $n_0 = 4$ and $c = 3$.
- Claim: For all $n \geq 4$, we have $0 \leq 2 \cdot n^2 + 10 \leq 3 \cdot n^2$.
- To prove the claim, first notice that for $n \geq 4$,

$$2 \cdot n^2 + 10 \leq 3 \cdot n^2$$
$$\Leftrightarrow$$
$$10 \leq n^2$$
$$\Leftrightarrow$$
$$\sqrt{10} \leq n$$

This is sufficient rigor for a midterm problem

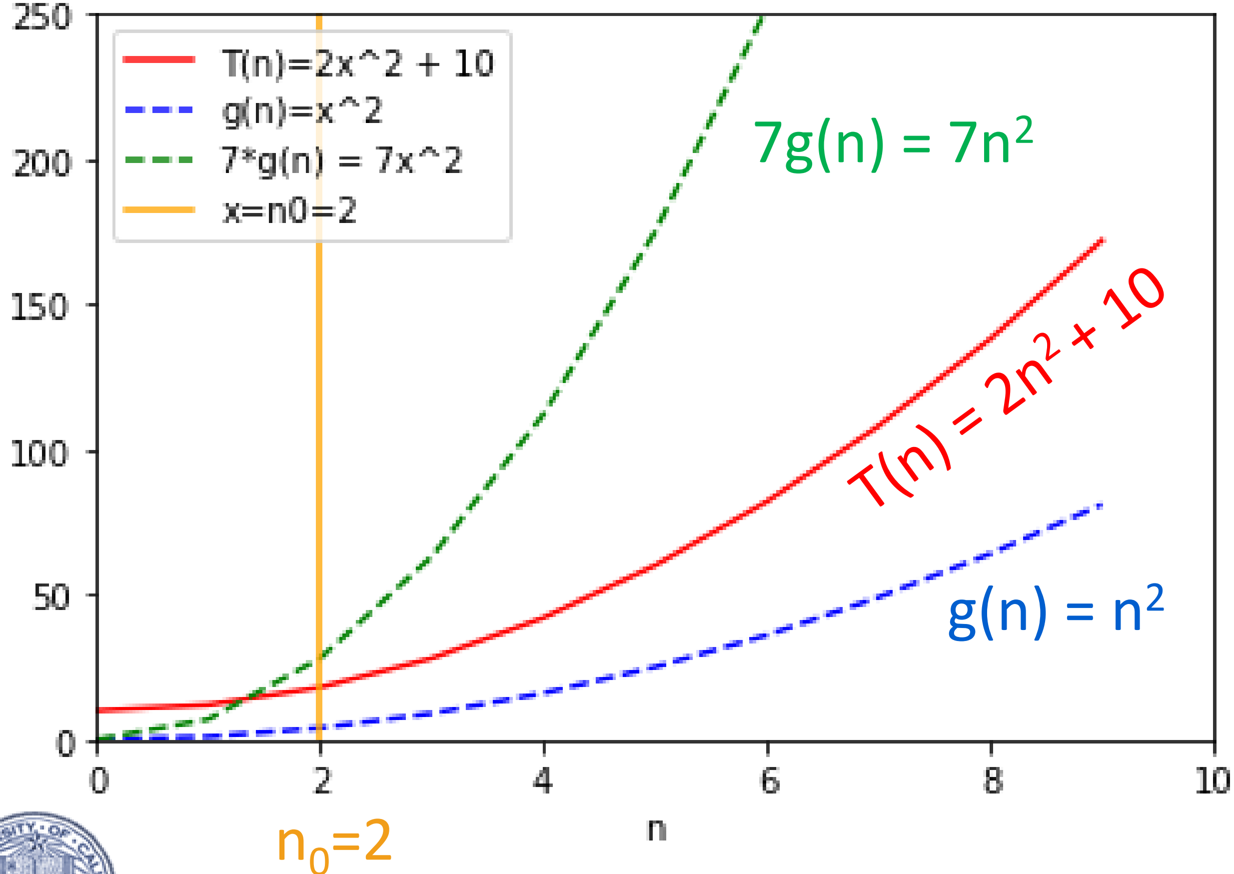
- This last thing is true for any $n \geq 4$, since

$$\sqrt{10} \approx 3.16 \leq 4.$$

- We also have $0 \leq 2 \cdot n^2 + 10$ for all n, since $n^2 \geq 0$ is always positive.

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

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



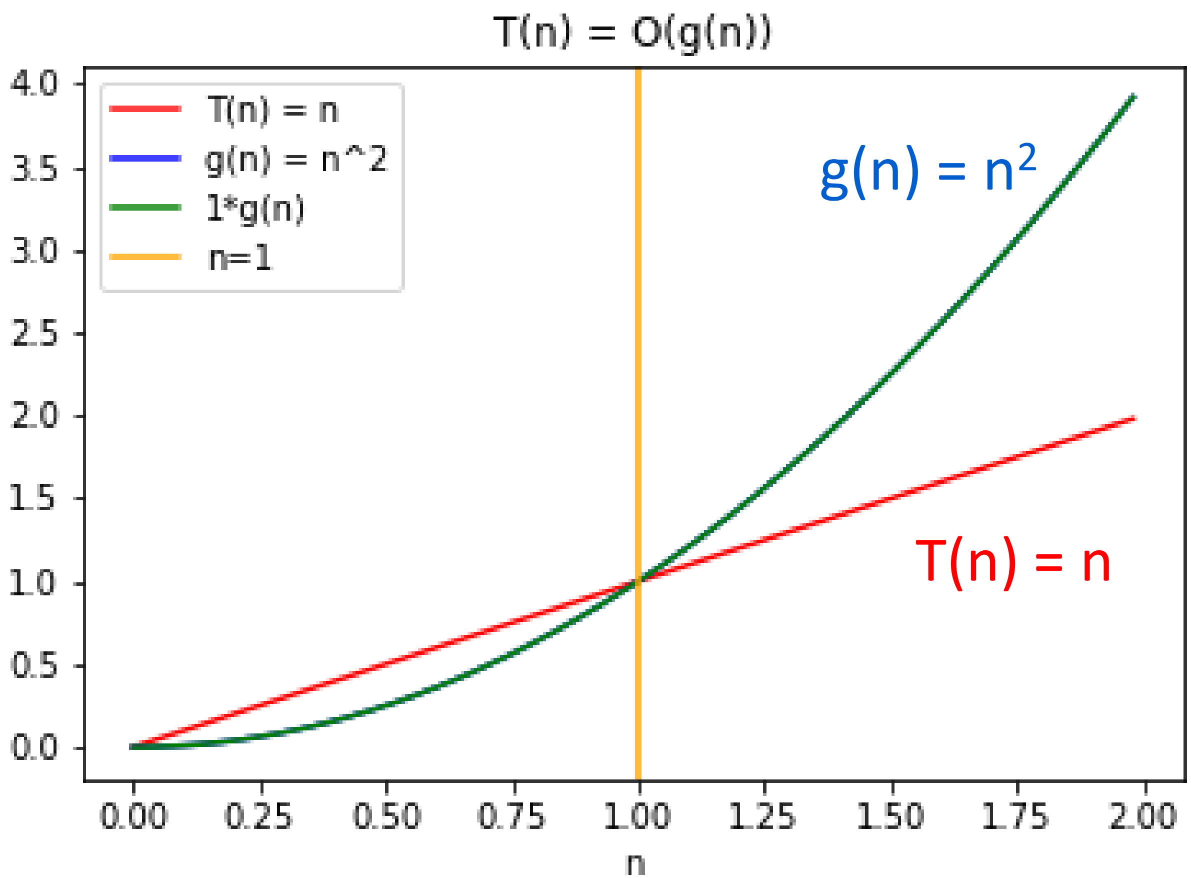
Formally:
- Choose $c = 7$
- Choose $n_0 = 2$
- Then:

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

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

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

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



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

$$\forall n \geq 1,$$

$$0 \leq n \leq n^2$$

$\Omega(\ldots)$ means a lower bound

- We say "$T(n)$ is $\Omega\big(g(n)\big)$" if $T(n)$ grows at least as fast as $g(n)$ as n gets large.

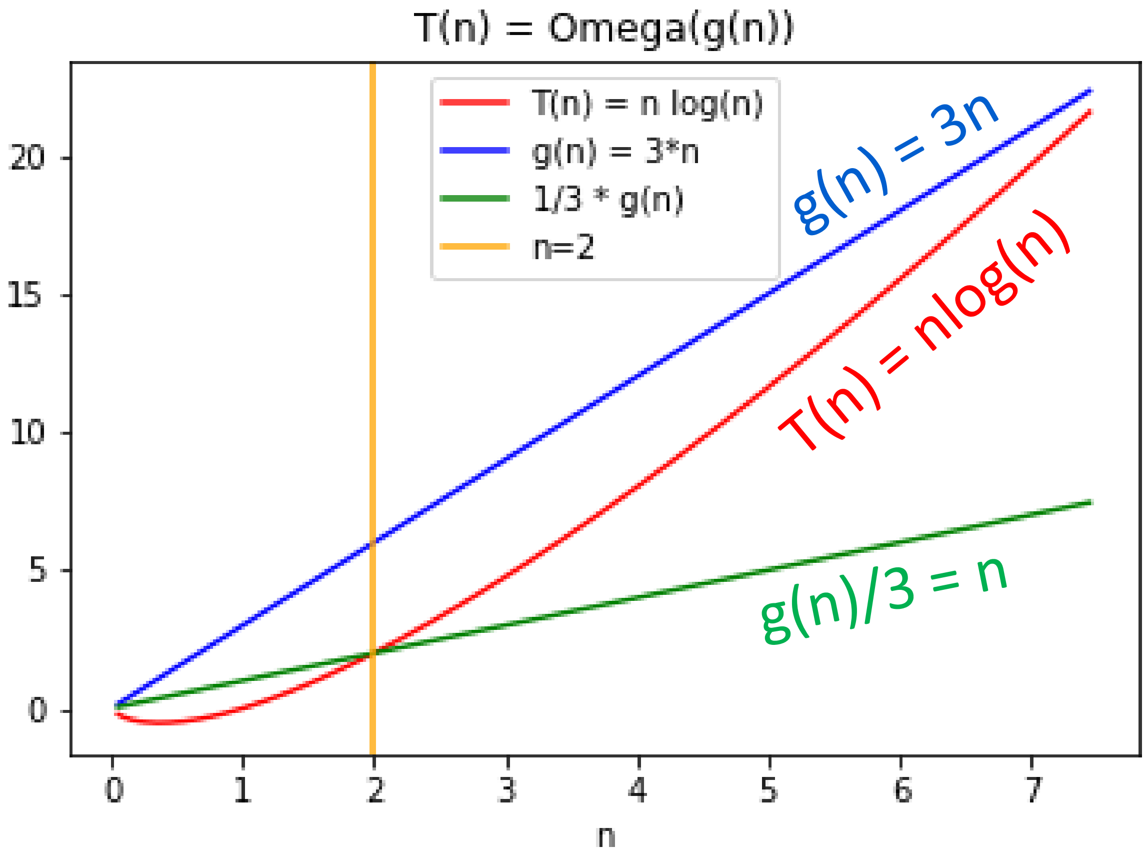
- Formally,

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

Switched these!!

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

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



- Choose $c = \frac{1}{3}$
- Choose $n_0 = 2$
- Then

$$\forall n \geq 2,$$

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

$\Theta(\ldots)$ means both!

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

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

and

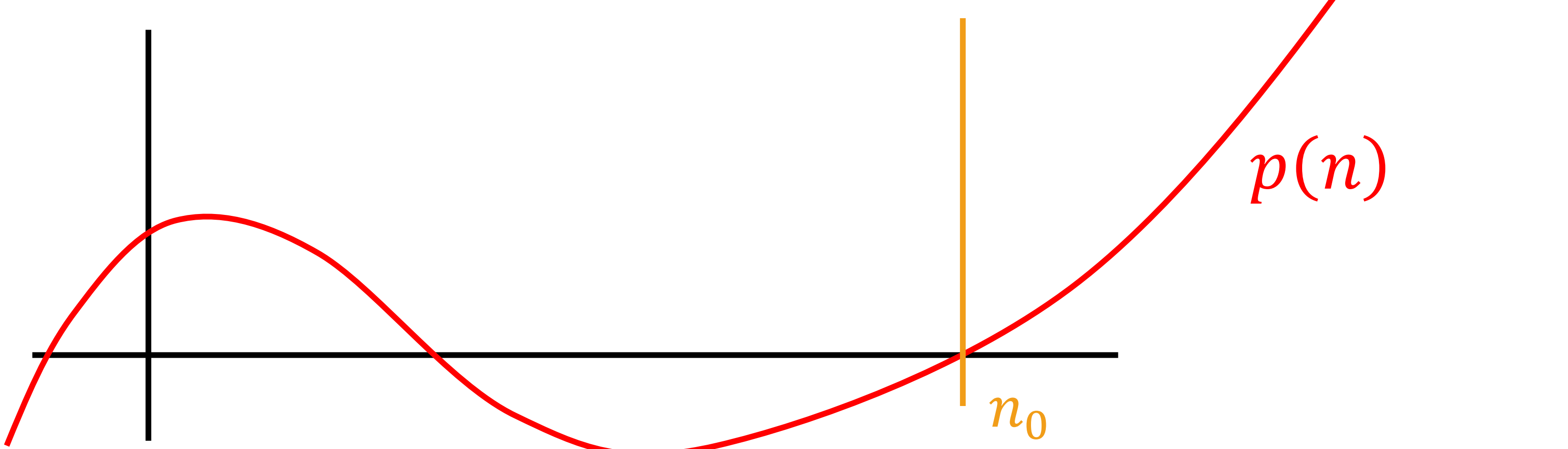
$$T(n) = \Omega\big(g(n)\big)$$

Example: polynomials

- Suppose the $p(n)$ is a polynomial of degree k:

$p(n) = a_0 + a_1 n + a_2 n^2 + \cdots + a_k n^k$ where $a_k > 0$.

- Then $p(n) = O\left(n^k\right)$
- Proof:
  - Choose $n_0 \geq 1$ so that $p(n) \geq 0$ for all $n \geq n_0$.
  - Choose $c = |a_0| + |a_1| + \cdots + |a_k|$



Example: polynomials

- Suppose the $p(n)$ is a polynomial of degree k:

$p(n) = a_0 + a_1 n + a_2 n^2 + \cdots + a_k n^k$ where $a_k > 0$.

- Then $p(n) = O\left(n^k\right)$
- Proof:
  - Choose $n_0 \geq 1$ so that $p(n) \geq 0$ for all $n \geq n_0$.
  - Choose $c = |a_0| + |a_1| + \cdots + |a_k|$
  - Then for all $n \geq n_0$:
  - $0 \leq p(n) \leq |p(n)| \leq |a_0| + |a_1|n + \cdots + |a_k|n^k$
  - $\leq |a_0|n^k + |a_1|n^k + \cdots + |a_k|n^k$
  - $= c \cdot n^k$

Triangle inequality!

Because $n \leq n^k$ for $n \geq n_0 \geq 1$.

Definition of c

Example: more polynomials

- For any $k \geq 1, n^k$ is NOT $O\left(n^{k-1}\right)$.
- Proof:
  - Suppose that it were. Then there is some c, n_0 so that

    $$n^k \leq c \cdot n^{k-1} \text{ for all } n \geq n_0$$
  - Aka, $n \leq c$ for all $n \geq n_0$
  - But that's not true! What about $n = n_0 + c + 1$?!
  - We have a contradiction! It *can't* be that $n^k = O\left(n^{k-1}\right)$.

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 be lying to you by deriving a contradiction.

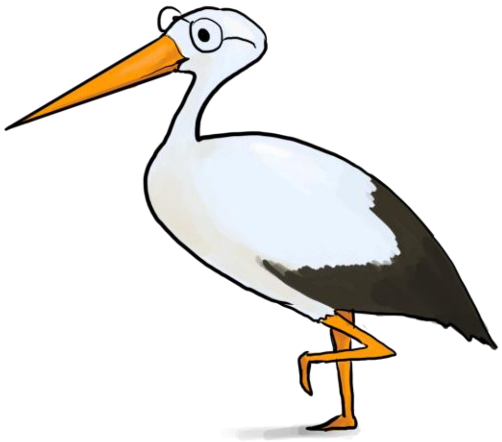
Yet more examples

- $n^3 + 3n = O(n^3 - n^2)$
- $n^3 + 3n = \Omega(n^3 - n^2)$
- $n^3 + 3n = \Theta(n^3 - n^2)$

- 3^n is **NOT** $O(2^n)$
- $\log(n) = \Omega(\ln(n))$
- $\log(n) = \Theta(2^{\log(\log(n))})$

remember that $\log = \log_2$ in this class.

Work through these on your own!

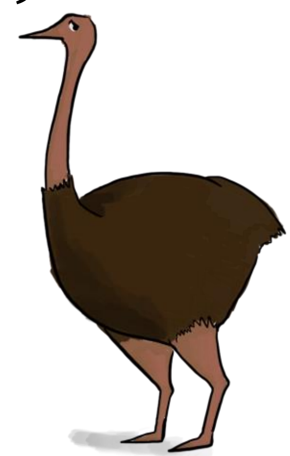


Siggi the Studious Stork

Some brainteasers

- Are there functions f, g so that NEITHER $f = O(g)$ nor $f = \Omega(g)$?
- Are there non-decreasing functions f, g so that the above is true?
- Define the n'th fibonacci number by $F(0) = 1, F(1) = 1, F(n) = F(n-1) + F(n-2)$ for $n > 2$.
  - $1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$

  True or false:
  - $F(n) = O(2^n)$
  - $F(n) = \Omega(2^n)$



Ollie the Over-achieving Ostrich

Recap: Asymptotic Notation

This is my
happy face!

- This makes both Plucky and Lucky happy.
  - **Plucky the Pedantic Penguin** is happy because there is a precise definition.
  - **Lucky the Lackadaisical Lemur** is happy because we don't have to pay close attention to all those pesky constant factors like "11".
- But we should always be careful not to abuse it.
- In the course, (almost) every algorithm we see will be actually practical, without needing to take $n \geq n_0 = 2^{10000000}$.

Questions about asymptotic notation?