Lecture 2

Asymptotic Notation,

Worst-Case Analysis, and MergeSort

Last time

Philosophy

- Algorithms are awesome!
- Our motivating questions:
  - Does it work?
  - Is it fast?
  - Can I do better?

Technical content

- Karatsuba integer multiplication
- Example of “Divide and Conquer”
- Not-so-rigorous analysis

Today

- We are going to ask:
  - Does it work?
  - Is it fast?

- We'll start to see how to answer these by looking at some examples of sorting algorithms.
  - InsertionSort
  - MergeSort

The Plan

- Sorting!
- Worst-case analyisis
  - InsertionSort: Does it work?
- Asymptotic Analysis
  - InsertionSort: Is it fast?
- MergeSort
  - Does it work?
  - Is it fast?

The Sorting Problem

- **Input:**
  - A sequence of n numbers $a_1, a_2, \dots, a_n$
- **Output:**
  - A permutation (reordering) $a_1', a_2', \dots, a_n'$ of the input sequence such that $a_1' \le a_2' \le \cdots \le a_n'$

Structure of data

- Usually, the numbers to be sorted are part of a collection of data called a record

- Each record contains a key, which is the value to be sorted

example of a record

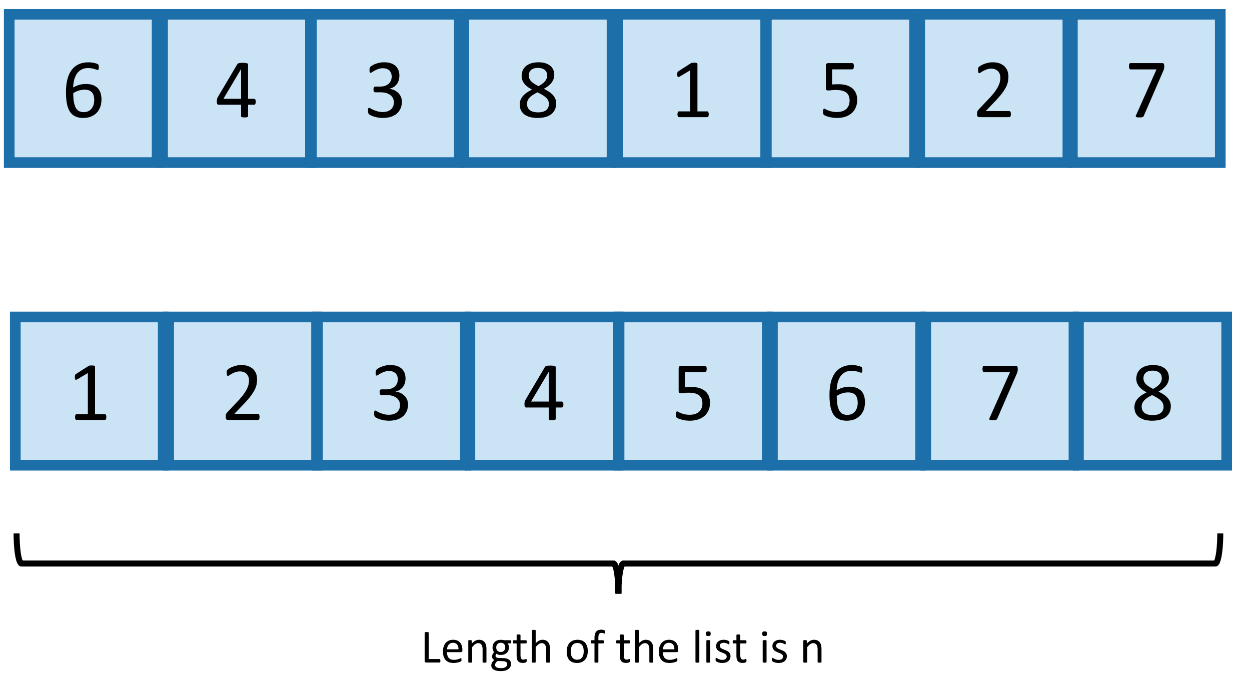
Key	**other data**

- Note that when the keys must be rearranged, the data associated with the keys must also be rearranged (time consuming !!)

- Pointers can be used instead (space consuming !!)

Sorting

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



Why Study Sorting Algorithms?

- There are a variety of situations that we can encounter
  - Do we have randomly ordered keys?
  - Are all keys distinct?
  - How large is the set of keys to be ordered?
  - Need guaranteed performance?

- Various algorithms are better suited to some of these situations

Some Definitions

- Internal Sort
  - The data to be sorted is all stored in the computer’s main memory.
- External Sort
  - Some of the data to be sorted might be stored in some external, slower, device.
- In Place Sort
  - The amount of extra space required to sort the data is constant with the input size.

Stability

- A STABLE sort preserves relative order of records with equal keys

Sorted on first key:

Aaron	4	A	664-480-0023	097 Little
Andrews	3	A	874-088-1212	121 Whitman
Battle	4	C	991-878-4944	308 Blair
Chen	2	A	884-232-5341	11 Dickinson
Fox	1	A	243-456-9091	101 Brown
Furia	3	A	766-093-9873	22 Brown
Gazsi	4	B	665-303-0266	113 Walker
Kanaga	3	B	898-122-9643	343 Forbes
Rohde	3	A	232-343-5555	115 Holder
Quilici	1	C	343-987-5642	32 McCosh

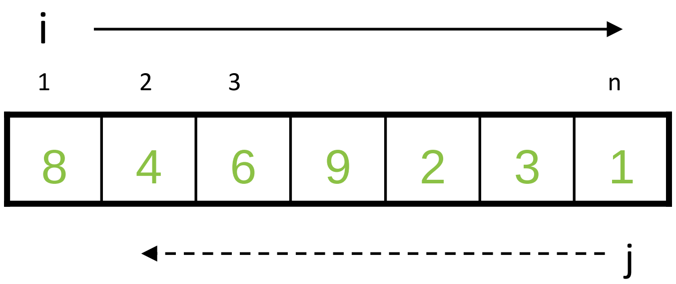
Sort file on second key:

Records with key value 3 are not in order on first key!!

Fox	1	A	243-456-9091	101 Brown
Quilici	1	C	343-987-5642	32 McCosh
Chen	2	A	884-232-5341	11 Dickinson
Kanaga	3	B	898-122-9643	343 Forbes
Andrews	3	A	874-088-1212	121 Whitman
Furia	3	A	766-093-9873	22 Brown
Rohde	3	A	232-343-5555	115 Holder
Battle	4	C	991-878-4944	308 Blair
Gazsi	4	B	665-303-0266	113 Walker
Aaron	4	A	664-480-0023	097 Little

Bubble Sort

- Idea:
  - Repeatedly pass through the array
  - Swaps adjacent elements that are out of order



- Easier to implement, but slower than Insertion sort

Selection Sort

- Idea:
  - Find the smallest element in the array
  - Exchange it with the element in the first position
  - Find the second smallest element and exchange it with the element in the second position
  - Continue until the array is sorted
- Disadvantage:
  - Running time depends only slightly on the amount of order in the file

Example

8	4	6	9	2	3	1

1	4	6	9	2	3	8

1	2	6	9	4	3	8

1	2	3	9	4	6	8

1	2	3	4	9	6	8

1	2	3	4	6	9	8

1	2	3	4	6	8	9

1	2	3	4	6	8	9

Selection Sort

8	4	6	9	2	3	1

Alg.: SELECTION-SORT*(A)*

```
n ← length[A]
for j ← 1 to n - 1
      do smallest ← j
         for i ← j + 1 to n
               do if A[i] < A[smallest]
                     then smallest ← i
         exchange A[j] ↔ A[smallest]
```

Analysis of Selection Sort

Alg.: SELECTION-SORT*(A)*

	cost	times
n ← length[A]	c_1	1
for j ← 1 **to** n - 1	c_2	n
do smallest ← j	c_3	n-1
for i ← j + 1 **to** n	c_4	$\sum_{j=1}^{n-1}(n-j+1)$
do if A[i] < A[smallest]	c_5	$\sum_{j=1}^{n-1}(n-j)$
then smallest ← i	c_6	$\sum_{j=1}^{n-1}(n-j)$
exchange A[j] ↔ A[smallest]	c_7	n-1

≈$n^2/2$ comparisons

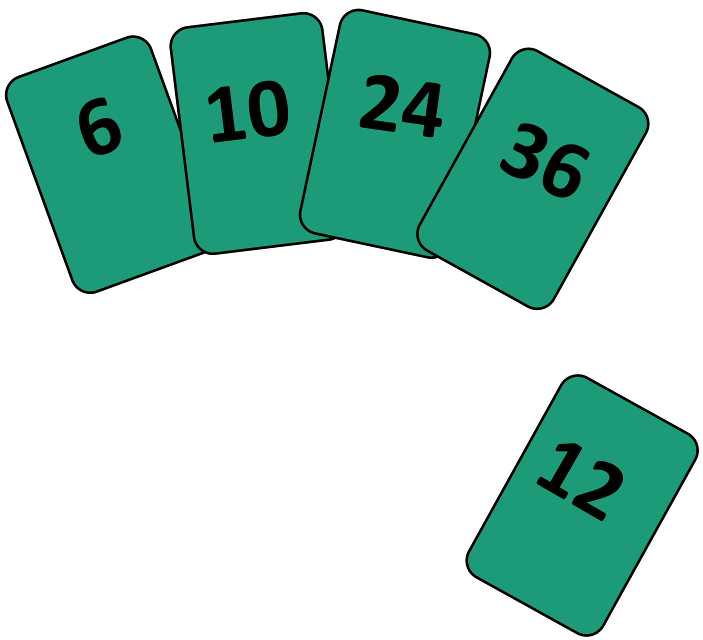
≈n exchanges

$$T(n) = c_1 + c_2 n + c_3(n-1) + c_4\sum_{j=1}^{n-1}(n-j+1) + c_5\sum_{j=1}^{n-1}(n-j) + c_6\sum_{j=2}^{n-1}(n-j) + c_7(n-1) = \Theta(n^2)$$

Insertion Sort

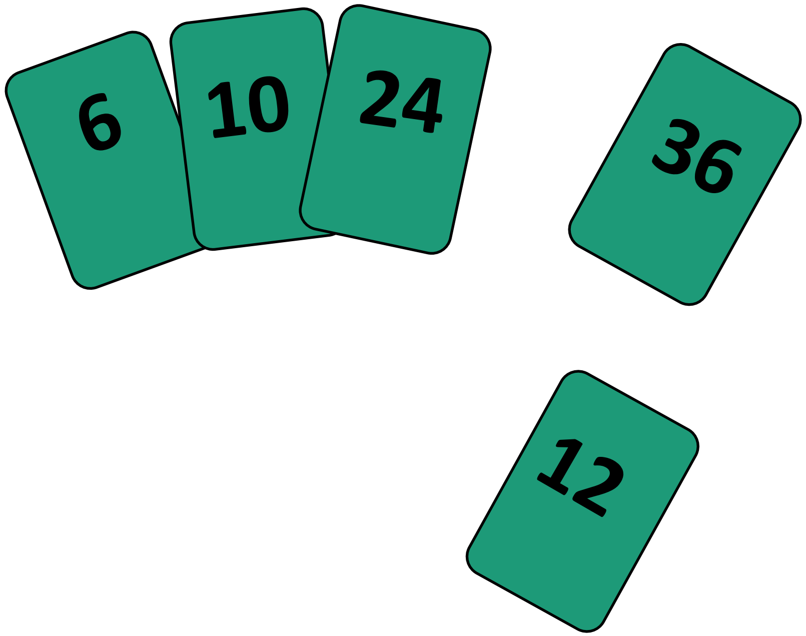
- Idea: like sorting a hand of playing cards
  - Start with an empty left hand and the cards facing down on the table.
  - Remove one card at a time from the table, and insert it into the correct position in the left hand
    - compare it with each of the cards already in the hand, from right to left
  - The cards held in the left hand are sorted
    - these cards were originally the top cards of the pile on the table

Insertion Sort

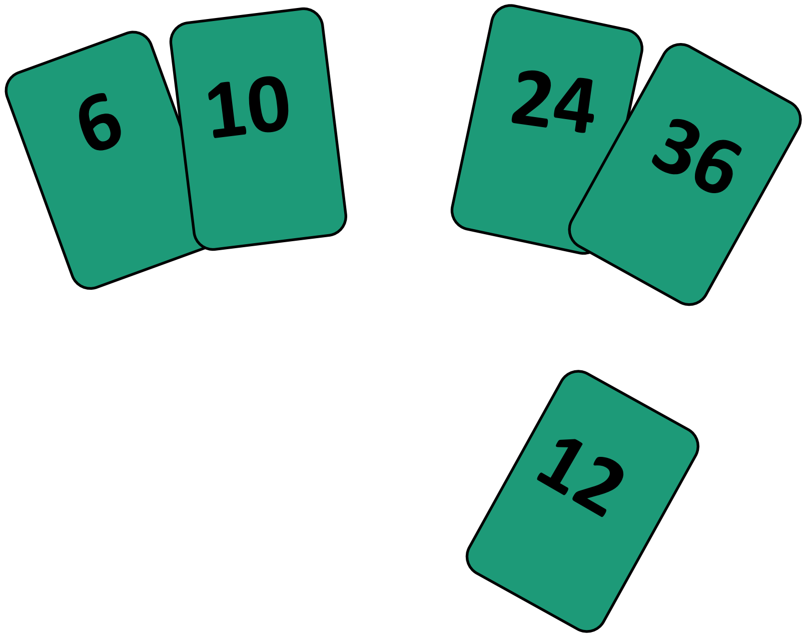


To insert 12, we need to make room for it by moving first 36 and then 24.

Insertion Sort



Insertion Sort



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

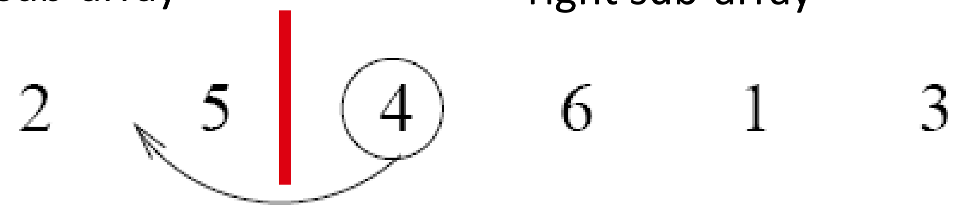
input array

5 2 4 6 1 3

at each iteration, the array is divided in two sub-arrays:

left sub-array

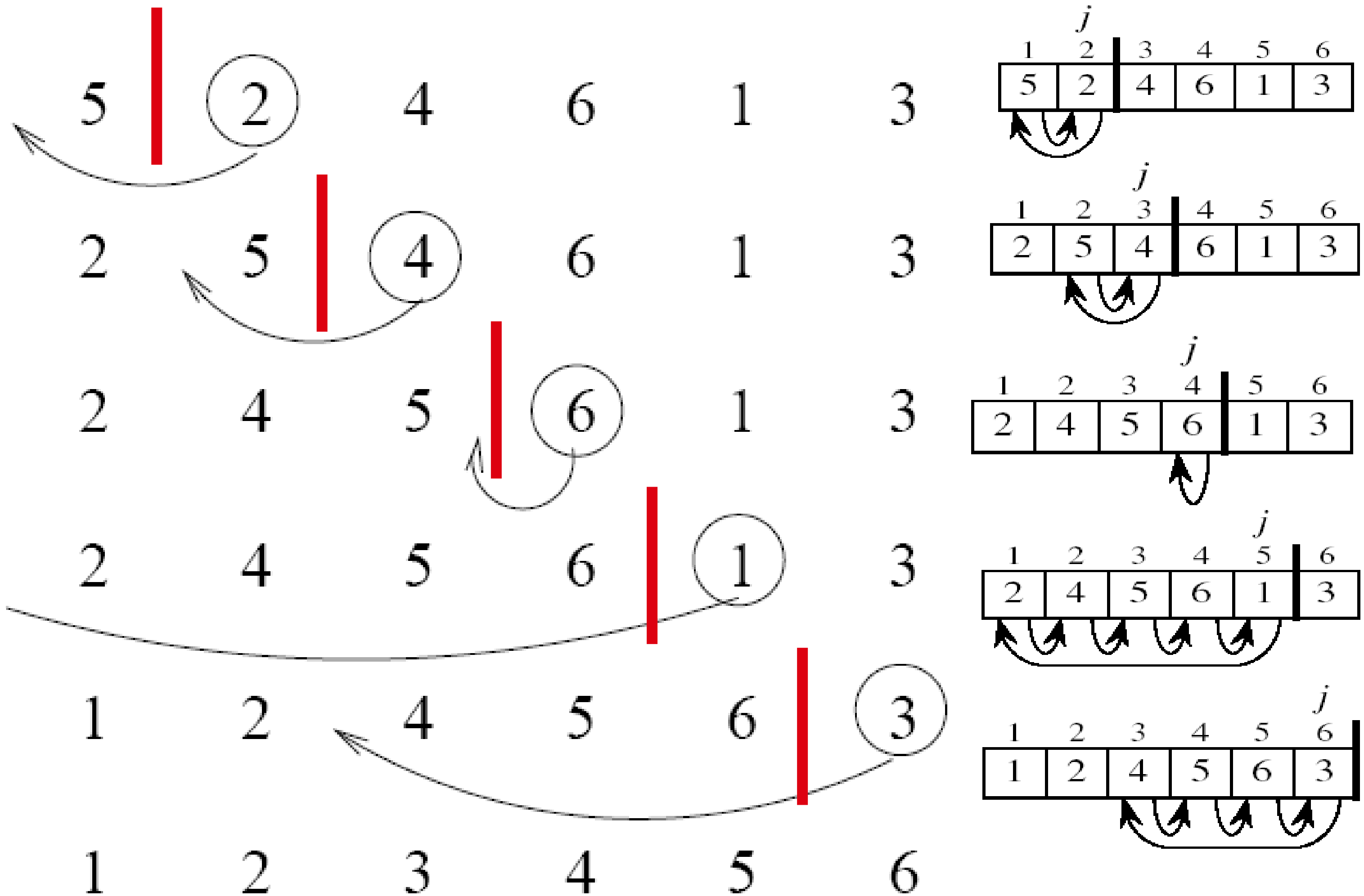
right sub-array



sorted

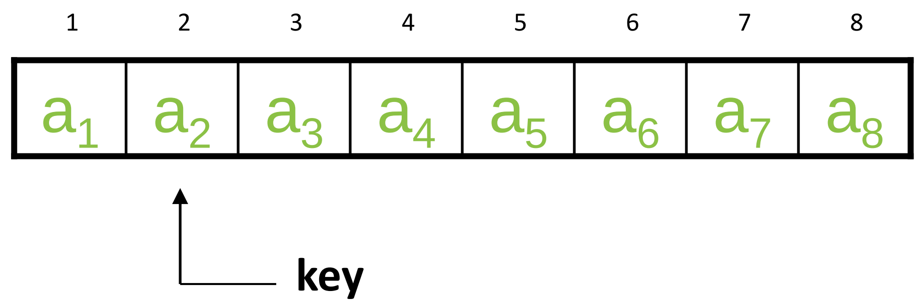
unsorted

Insertion Sort



INSERTION-SORT

Alg.: INSERTION-SORT*(A)*



for j ← 2 **to** n

  do key ← *A*[j]

    ▷Insert *A*[j] into the sorted sequence *A*[1 . . j -1]

    i ← j - 1

    while i > 0 and *A*[i] > key

      do *A*[i + 1] ← *A*[i]

        i ← i - 1

    A[i + 1] ← key

- Insertion sort – sorts the elements in place

Loop Invariant for Insertion Sort

Alg.: INSERTION-SORT*(A)*

for j ← 2 **to** n

do key ← A[j]

Insert A[j] into the sorted sequence A[1 . . j -1]

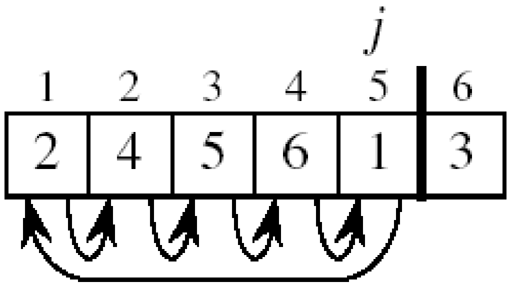
i ← j - 1

while i > 0 and A[i] > key

do A[i + 1] ← A[i]

i ← i – 1

A[i + 1] ← key



Invariant: at the start of the **for** loop the elements in A[1 . . j-1] are in sorted order

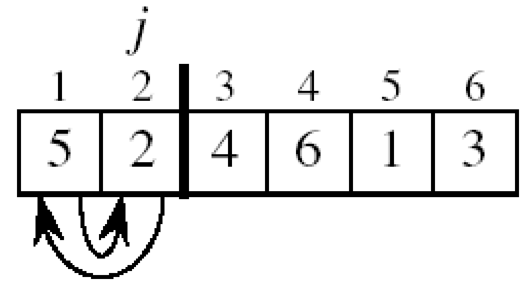
Proving Loop Invariants

- Proving loop invariants works like induction
- **Initialization (base case):**
  - It is true prior to the first iteration of the loop
- **Maintenance (inductive step):**
  - If it is true before an iteration of the loop, it remains true before the next iteration
- **Termination:**
  - When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct
  - Stop the induction when the loop terminates

Loop Invariant for Insertion Sort

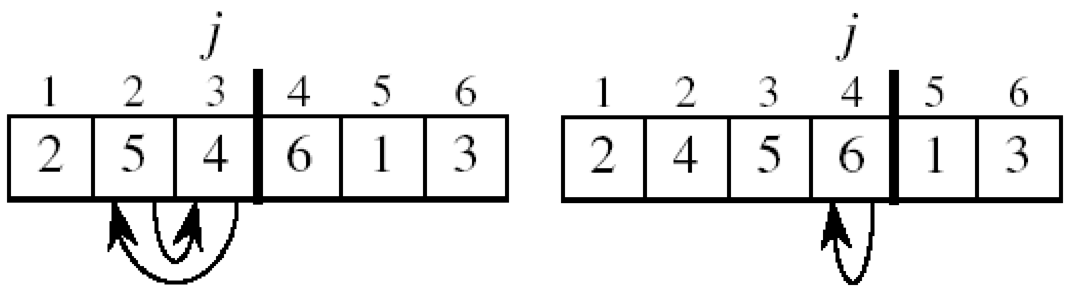
- **Initialization:**
  - Just before the first iteration, j = 2:

    the subarray $A[1 \,..\, j\text{-}1] = A[1]$, (the element originally in $A[1]$) – is sorted



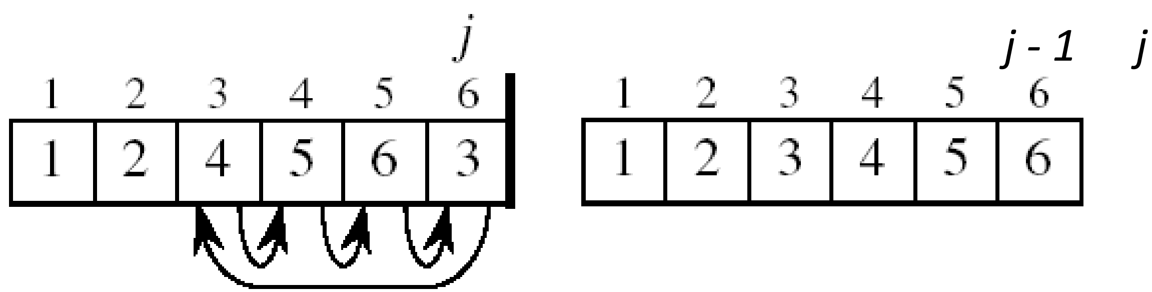
Loop Invariant for Insertion Sort

- **Maintenance:**
  - the **while** inner loop moves A[j -1], A[j -2], A[j -3], and so on, by one position to the right until the proper position for key (which has the value that started out in A[j]) is found
  - At that point, the value of key is placed into this position.



Loop Invariant for Insertion Sort

- **Termination:**
  - The outer **for** loop ends when $j = n + 1 \Rightarrow j\text{-}1 = n$
  - Replace n with $j\text{-}1$ in the loop invariant:
    - the subarray $A[1 \,..\, n]$ consists of the elements originally in $A[1 \,..\, n]$, but in sorted order



- The entire array is sorted!

Invariant: at the start of the **for** loop the elements in $A[1 \,..\, j\text{-}1]$ are in sorted order

Analysis of Insertion Sort

INSERTION-SORT*(A)*

	cost	times
for j ← 2 **to** n	c_1	n
do key ← A[j]	c_2	$n-1$
▷Insert A[j] into the sorted sequence A[1 . . j -1]	0	$n-1$
i ← j - 1	c_4	$n-1$
while i > 0 and A[i] > key	c_5	$\sum_{j=2}^{n} t_j$
do A[i + 1] ← A[i]	c_6	$\sum_{j=2}^{n} (t_j - 1)$
i ← i – 1	c_7	$\sum_{j=2}^{n} (t_j - 1)$
A[i + 1] ← key	c_8	$n-1$

t_j: # of times the while statement is executed at iteration j

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} \left(t_j - 1\right) + c_7 \sum_{j=2}^{n} \left(t_j - 1\right) + c_8(n-1)$$

Best Case Analysis

- The array is already sorted

"**while** i > 0 and A[i] > key"

  - $A[i] \leq key$ upon the first time the **while** loop test is run

    (when $i = j - 1$)

  - $t_j = 1$

- $T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) =$

  $(c_1 + c_2 + c_4 + c_5 + c_8)n + (c_2 + c_4 + c_5 + c_8)$

  $= an + b = \Theta(n)$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

Worst Case Analysis

- The array is in reverse sorted order **"while** i > 0 and A[i] > key"
  - Always A[i] > key in **while** loop test
  - Have to compare key with all elements to the left of the j-th position $\Rightarrow$ compare with j-1 elements $\Rightarrow$ $t_j = j$

using $\sum_{j=1}^{n} j = \frac{n(n+1)}{2} \Rightarrow \sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1 \Rightarrow \sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$ we have:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6 \frac{n(n-1)}{2} + c_7 \frac{n(n-1)}{2} + c_8(n-1)$$

$$= an^2 + bn + c$$ a quadratic function of n

- $T(n) = \Theta(n^2)$ order of growth in n^2

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

Comparisons and Exchanges in Insertion Sort

INSERTION-SORT*(A)*

	cost	times
for j ← 2 **to** n	c_1	n
do key ← A[j]	c_2	n-1
Insert A[j] into the sorted sequence A[1 . . j -1]	0	n-1
i ← j - 1	c_4	n-1
while i > 0 and A[i] > key	c_5	$\sum_{j=2}^{n} t_j$
do A[i + 1] ← A[i]	c_6	$\sum_{j=2}^{n} (t_j - 1)$
i ← i – 1	c_7	$\sum_{j=2}^{n} (t_j - 1)$
A[i + 1] ← key	c_8	n-1

$\approx n^2/2$ comparisons

$\approx n^2/2$ exchanges

Insertion Sort - Summary

- Advantages
  - Good running time for “almost sorted” arrays $\Theta(n)$
- Disadvantages
  - $\Theta(n^2)$ running time in worst and average case
  - $\approx n^2/2$ comparisons and exchanges

Insertion Sort

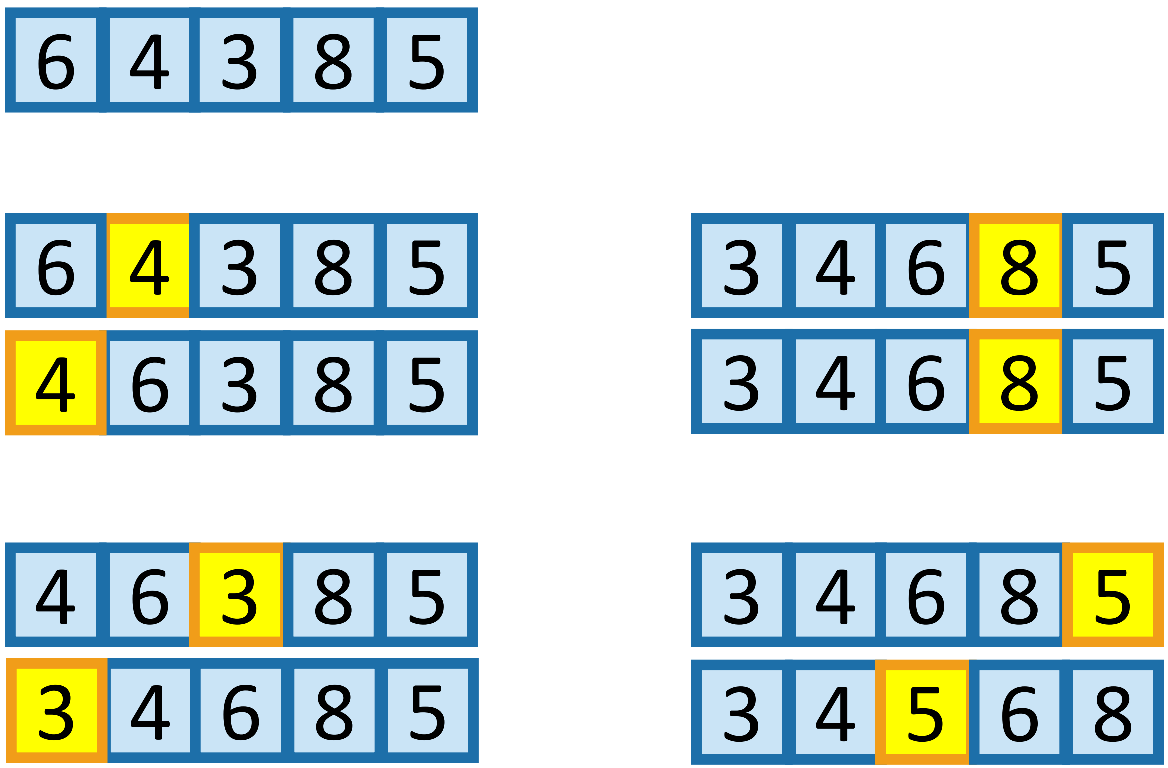
1. Does it work?
2. Is it fast?

What does that mean???

Plucky the
Pedantic Penguin

Claim: InsertionSort “works”

- “Proof:” It just worked in this example:



Sorted!

Claim: InsertionSort “works”

- “Proof:” I did it on a bunch of random lists and it always worked:

```
A = [1,2,3,4,5,6,7,8,9,10]
for trial in range(100):
    shuffle(A)
    InsertionSort(A)
    if is_sorted(A):
        print('YES IT IS SORTED!')
```

```
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
```

What does it mean to "work"?

- Is it enough to be correct on only one input?
- Is it enough to be correct on most inputs?

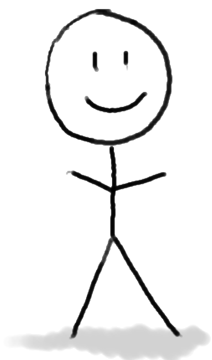
- In this class, we will use **worst-case analysis**:
  - An algorithm must be correct on **all possible** inputs.
  - The running time of an algorithm is the worst possible running time over all inputs.

Worst-case analysis

Think of it like a game:

Worst-case analysis guarantee: `Algorithm` should work (and be fast) on that worst-case input.

Here is my algorithm!



Algorithm designer

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

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

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

Insertion Sort

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

- Okay, so it's pretty obvious that it works.

- HOWEVER! In the future it won't be so obvious, so let's take some time now to see how we would prove this rigorously.

Why does this work?

- Say you have a sorted list, $\boxed{3}\boxed{4}\boxed{6}\boxed{8}$, and another element $\boxed{5}$.

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

- Then you get a sorted list: $\boxed{3}\boxed{4}\boxed{5}\boxed{6}\boxed{8}$

So just use this logic at every step.

6	4	3	8	5

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

4	6	3	8	5

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

4	6	3	8	5

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

3	4	6	8	5

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

3	4	6	8	5

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

3	4	6	8	5

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

3	4	6	8	5

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

3	4	5	6	8

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!

What have we learned?

- In this class we will use worst-case analysis:
  - We assume that a “random guy” comes up with a worst-case input for our algorithm, and we measure performance on that worst-case input.

The Plan

- InsertionSort recap
- Worst-case Analysis
  - Back to InsertionSort: Does it work?
- Asymptotic Analysis
  - Back to InsertionSort: Is it fast?
- MergeSort
  - Does it work?
  - Is it fast?

In this class we will use…

- **Big-Oh notation!**
- Gives us a meaningful way to talk about the running time of an algorithm, independent of programming language, computing platform, etc., without having to count all the operations.

Main idea:

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

Some examples…

(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.

Why is this a good idea?

- Suppose the running time of an algorithm is:

$$T(n) = 10n^2 + 3n + 7 \quad \text{ms}$$

This constant factor of 10 depends a lot on my computing platform…

These lower-order terms don't really matter as n gets large.

We're just left with the n^2 term! That's what's meaningful.

Pros and Cons of Asymptotic Analysis

Pros:

- Abstracts away from hardware- and language-specific issues.
- Makes algorithm analysis much more tractable.
- Allows us to meaningfully compare how algorithms will perform on large inputs.

Cons:

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

1000000000 n
is “better” than n^2 ?!?!

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

Informal definition for O(…)

- 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:

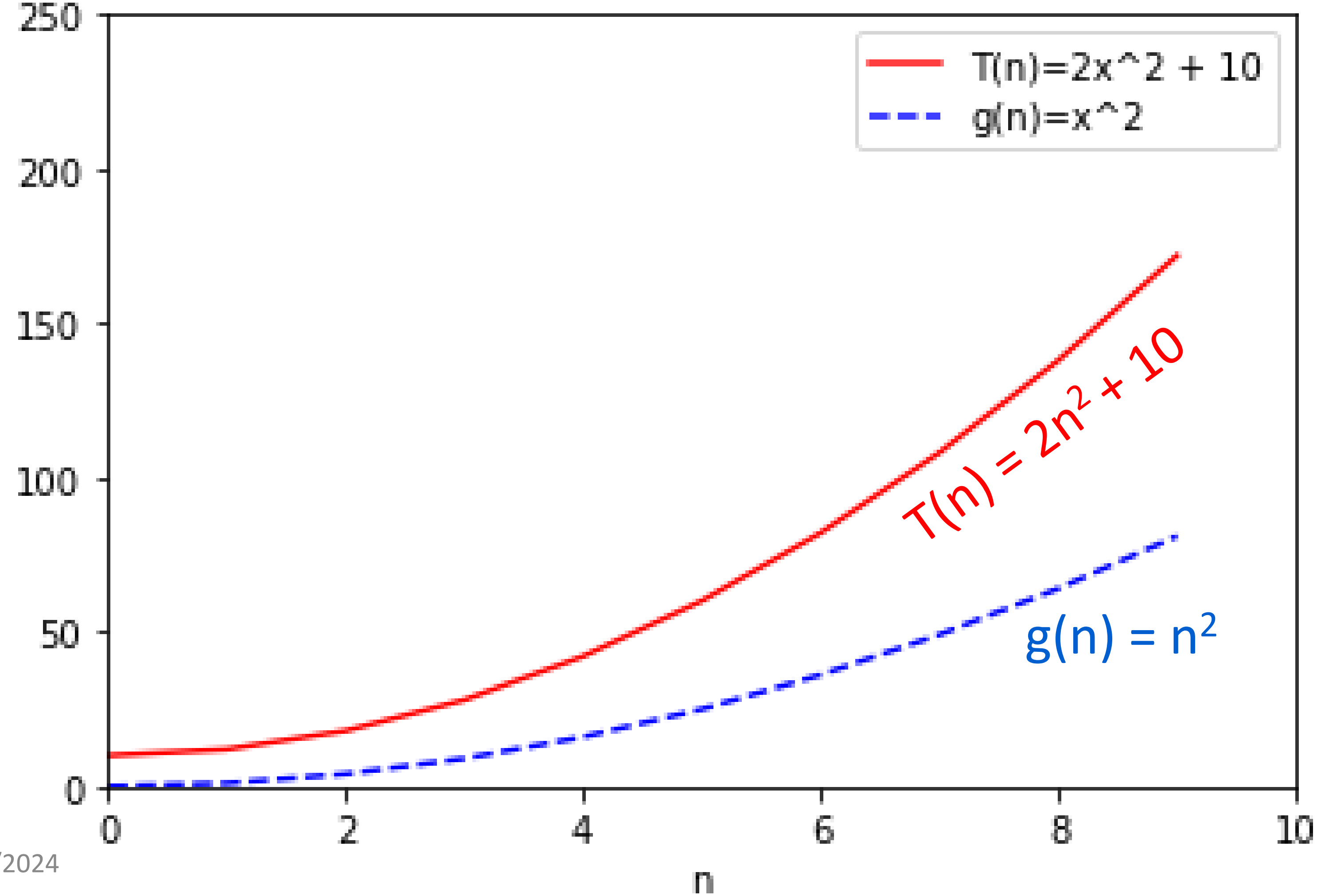
for large enough n,

$T(n)$ is at most some constant multiple of $g(n)$.

Here, “constant” means “some number that doesn’t depend on n.”

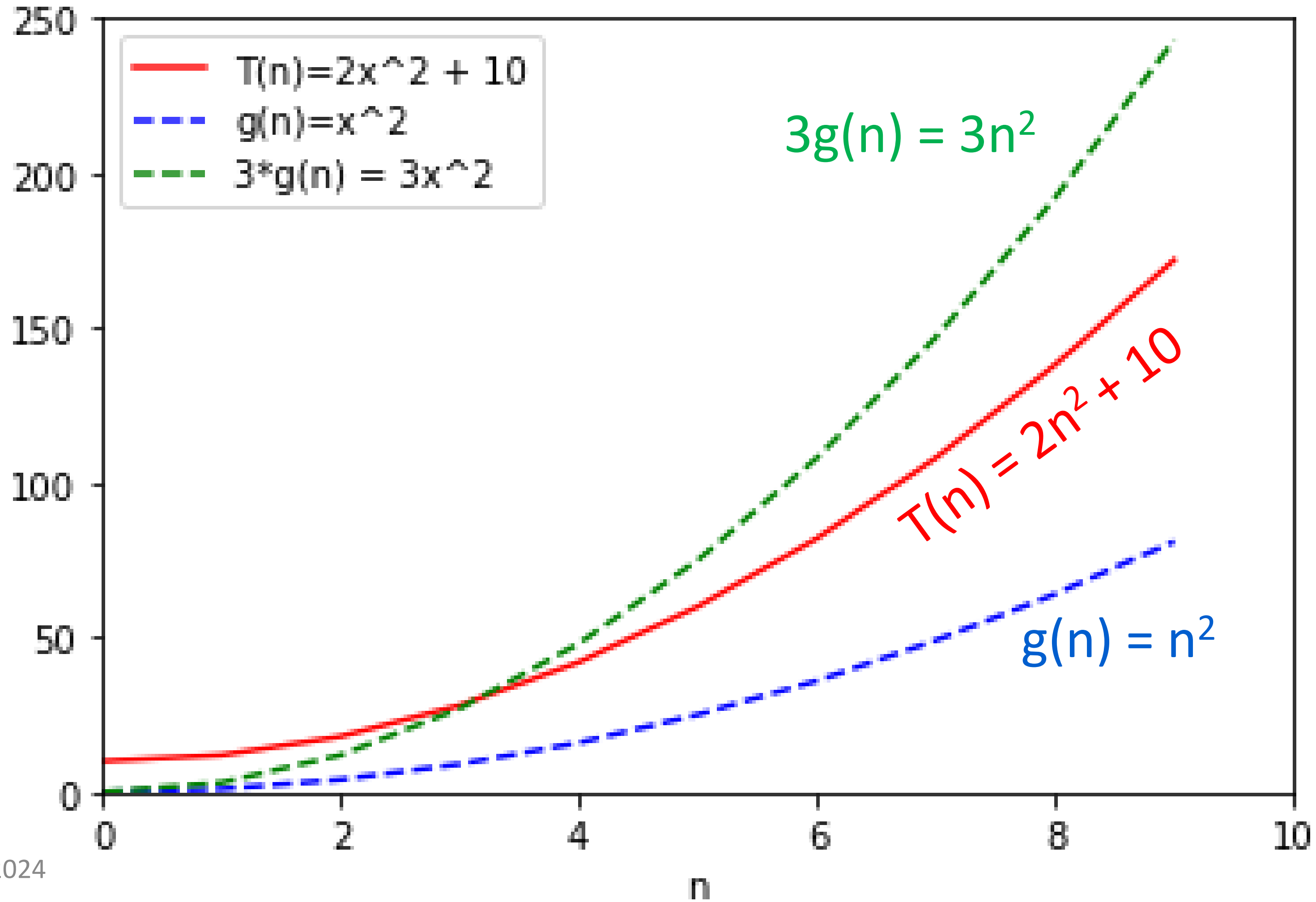
Example
$2n^2 + 10 = O(n^2)$

for large enough n,
$T(n)$ is at most some constant multiple of $g(n)$.



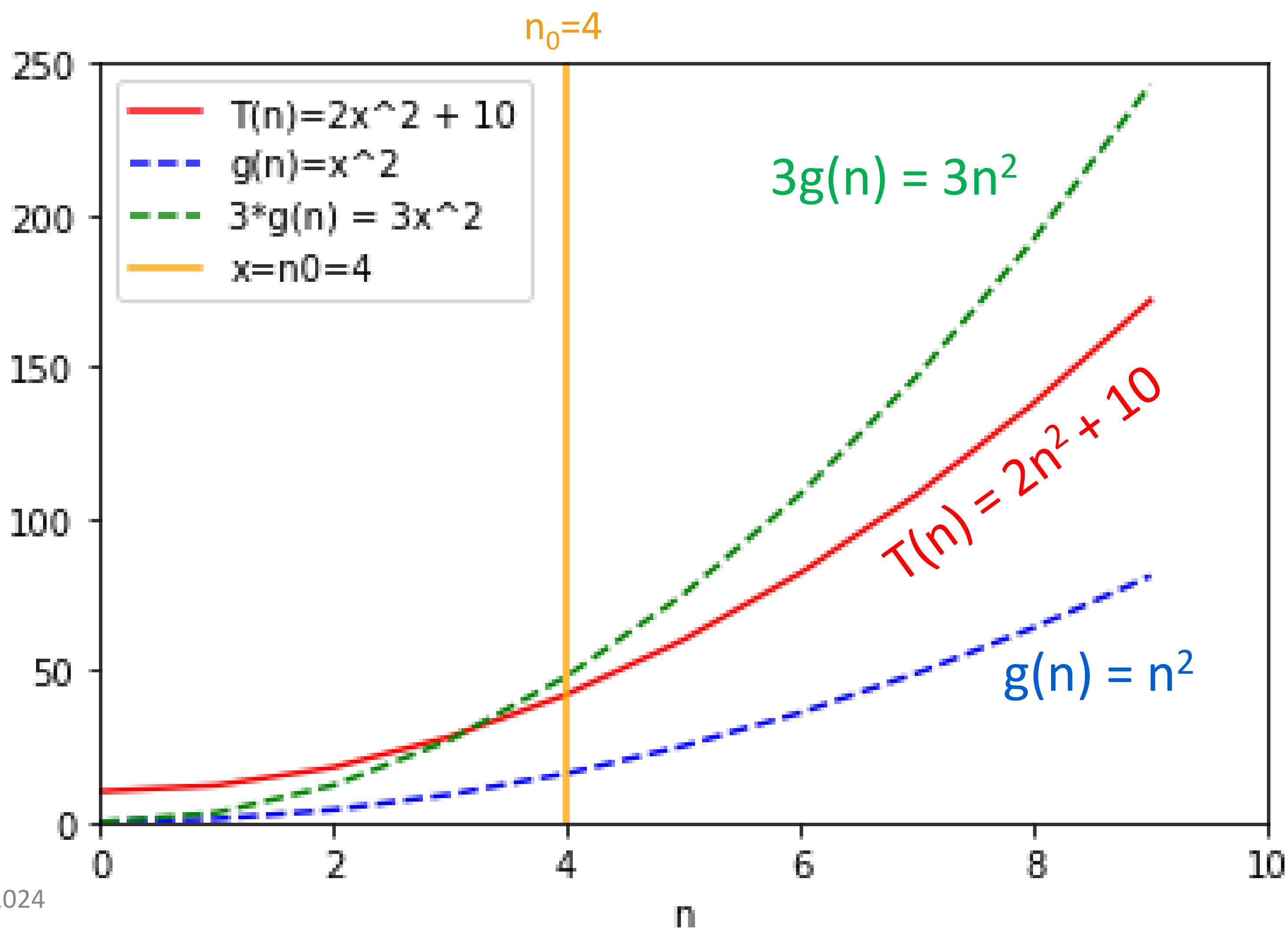
Example
$2n^2 + 10 = O(n^2)$

for large enough n,
$T(n)$ is at most some constant
multiple of $g(n)$.



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

for large enough n,
$T(n)$ is at most some constant
multiple of $g(n)$.



Formal definition of O(...)

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

- Formally,

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

"If and only if"

$$\Longleftrightarrow$$

"For all"

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

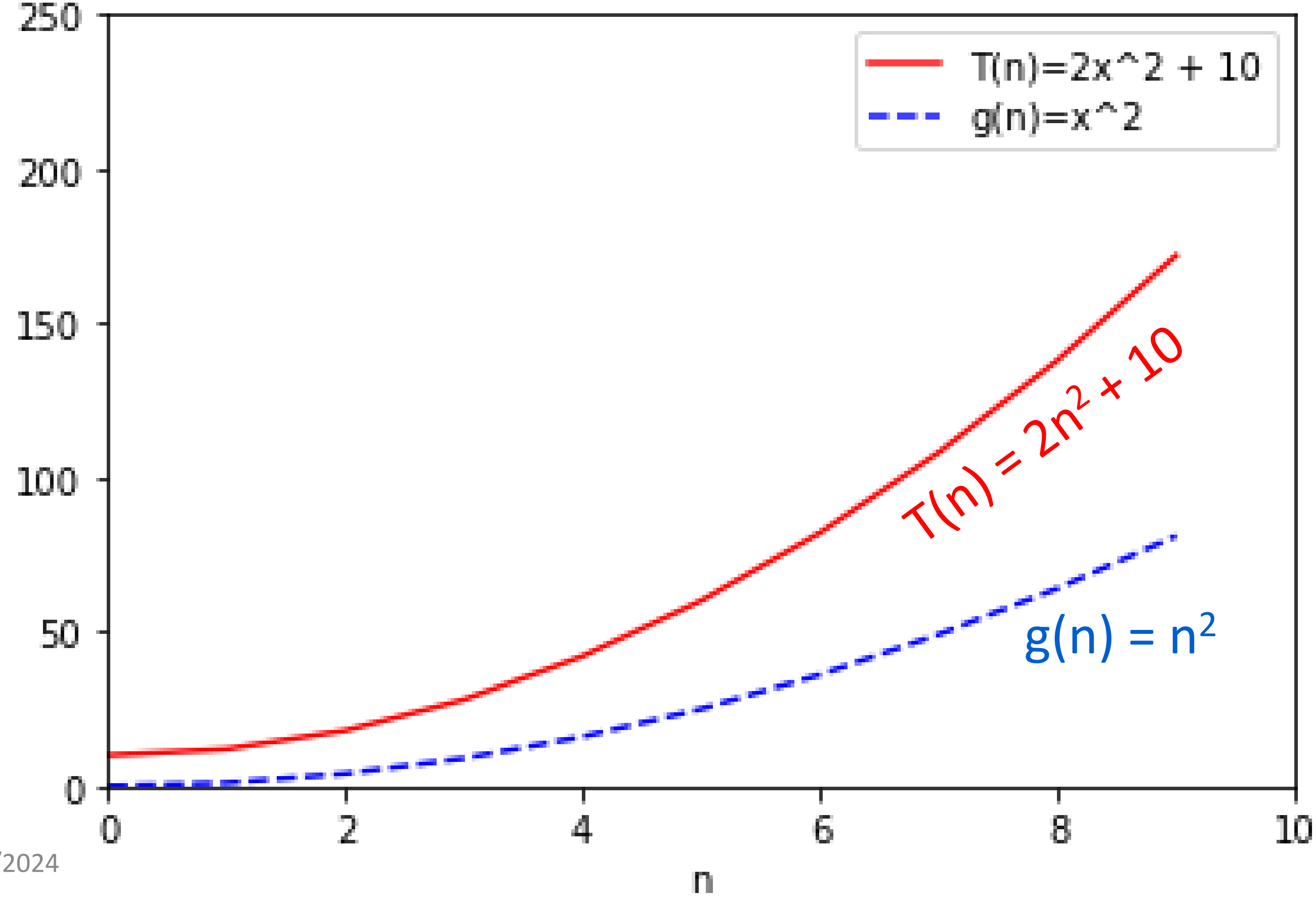
"There exists"

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

"such that"

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

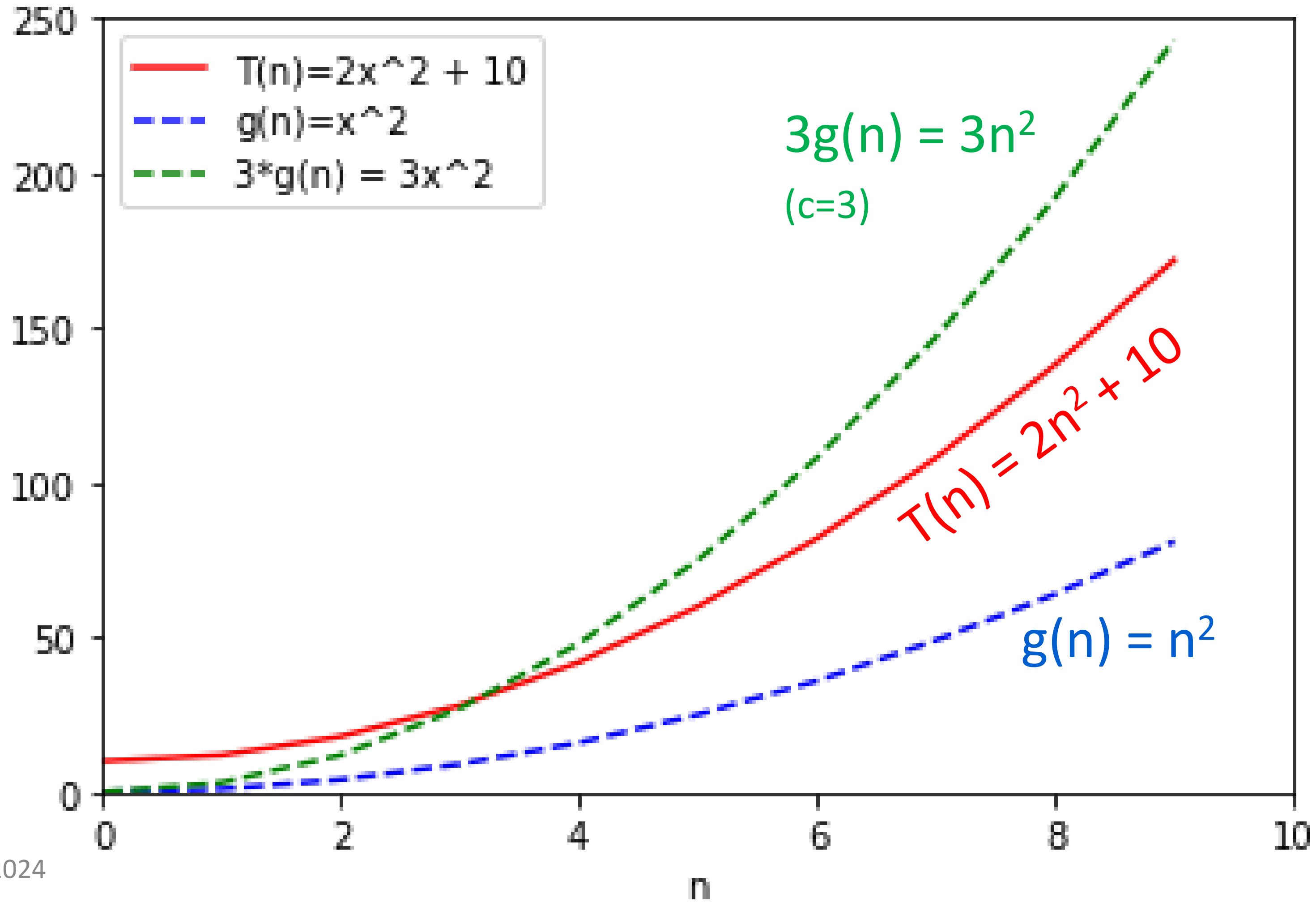
$$T(n) = O\big(g(n)\big)$$
$$\Leftrightarrow$$
$$\exists c, n_0 > 0 \;\; s.t. \;\; \forall n \geq n_0,$$
$$T(n) \leq c \cdot g(n)$$



Example

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

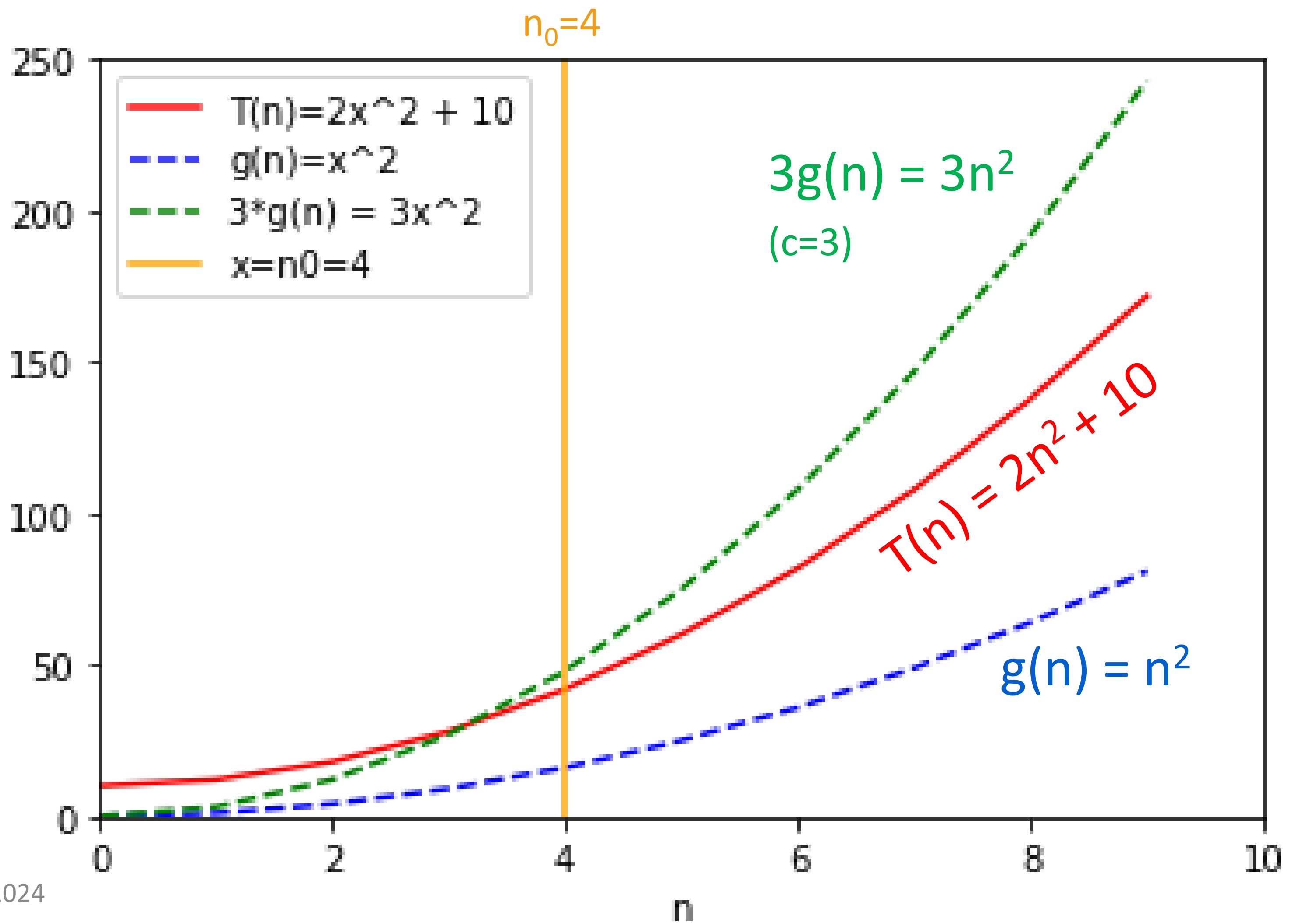
$$T(n) = O(g(n)) \Leftrightarrow \exists c, n_0 > 0 \;\; s.t. \;\; \forall n \geq n_0, \; T(n) \leq c \cdot g(n)$$



Example

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

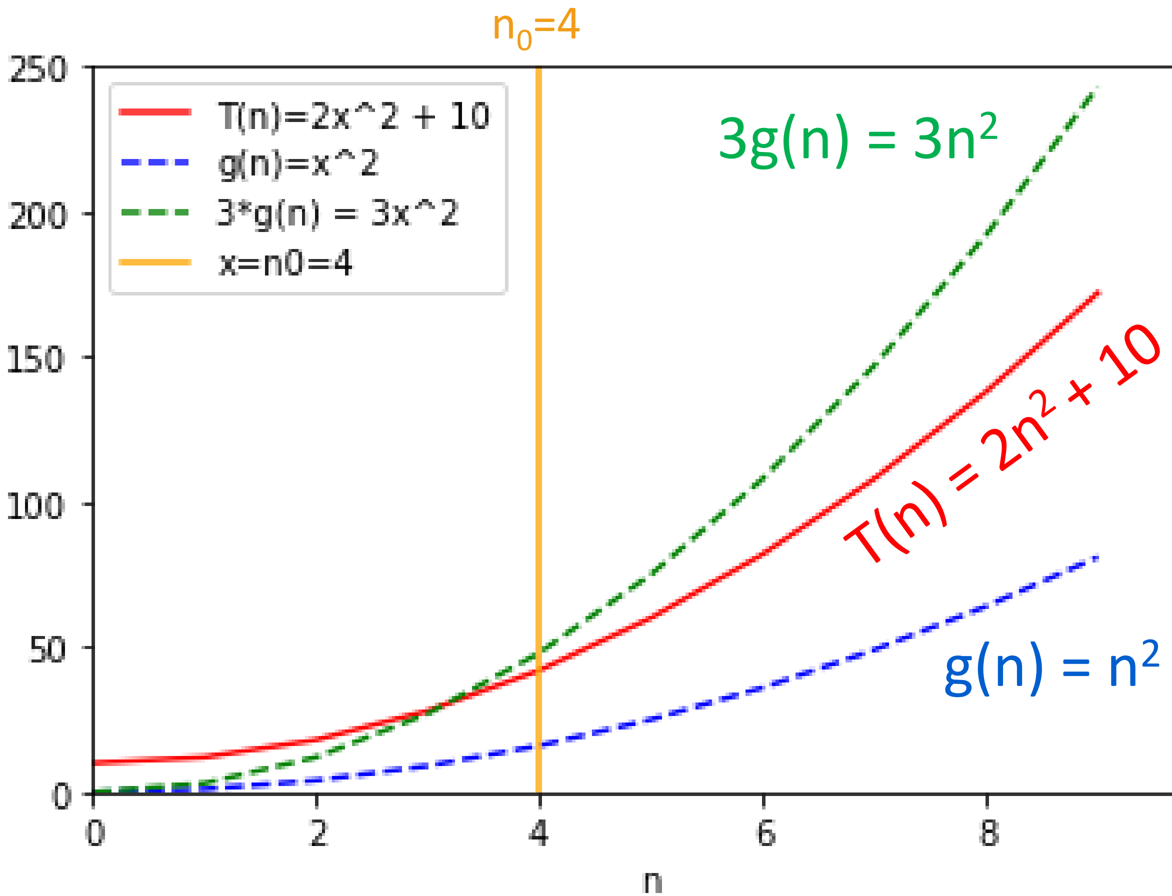
$$T(n) = O(g(n)) \iff \exists c, n_0 > 0 \;\; s.t. \;\; \forall n \geq n_0, \; T(n) \leq c \cdot g(n)$$



Example

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

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



Formally:

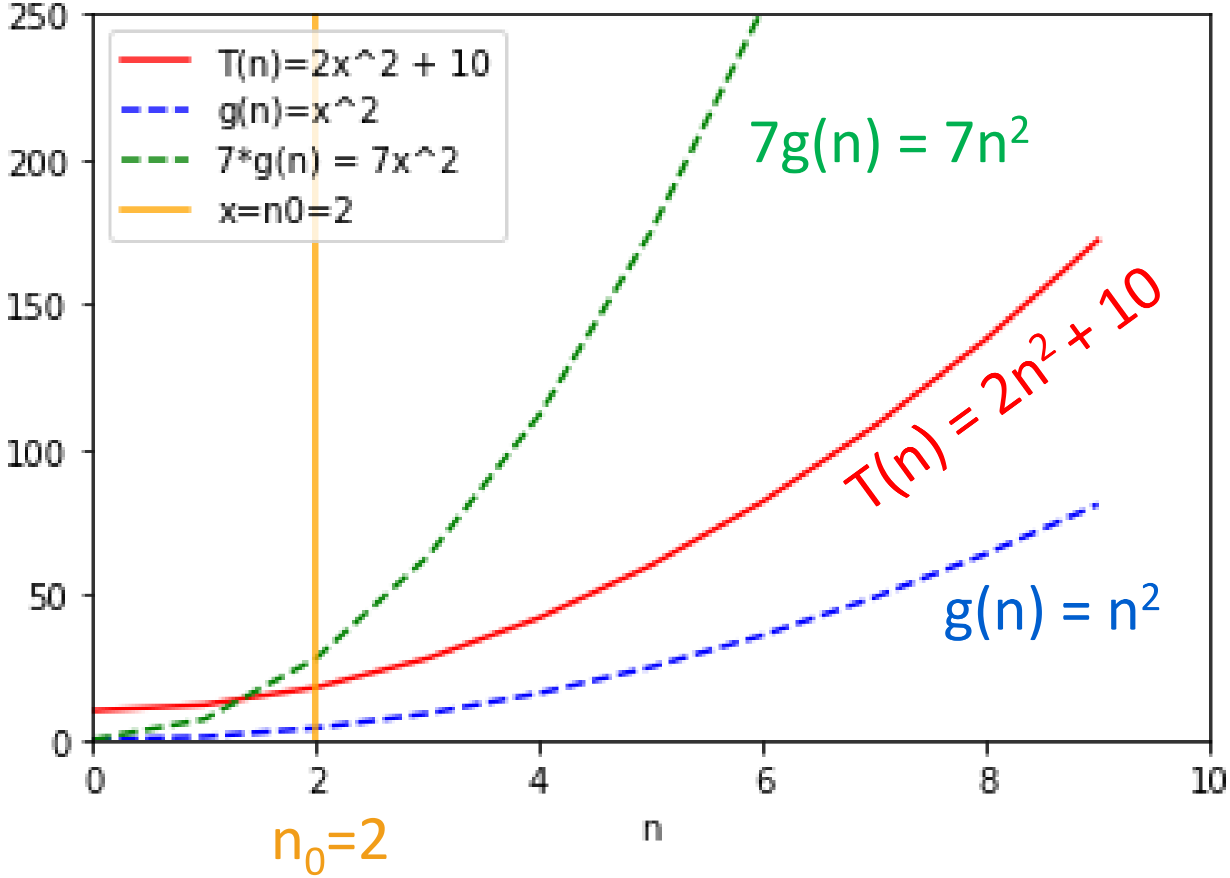
- Choose c = 3
- Choose n_0 = 4
- Then:

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

Same example

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

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



Formally:

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

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

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

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.

Recap: Asymptotic Notation

- 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.

- 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 $n \geq n_0 = 2^{10000000}$.

Insertion Sort: running time

As you get more used to this, you won't have to count up operations anymore. For example, just looking at the pseudocode below, you might think…

```
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

"There's O(1) stuff going on inside the inner loop, so each time the inner loop runs, that's O(n) work. Then the inner loop is executed O(n) times by the outer loop, so that's O(n^2)."

What have we learned?

InsertionSort is an algorithm that correctly sorts an arbitrary n-element array in time $O(n^2)$.

Can we do better?

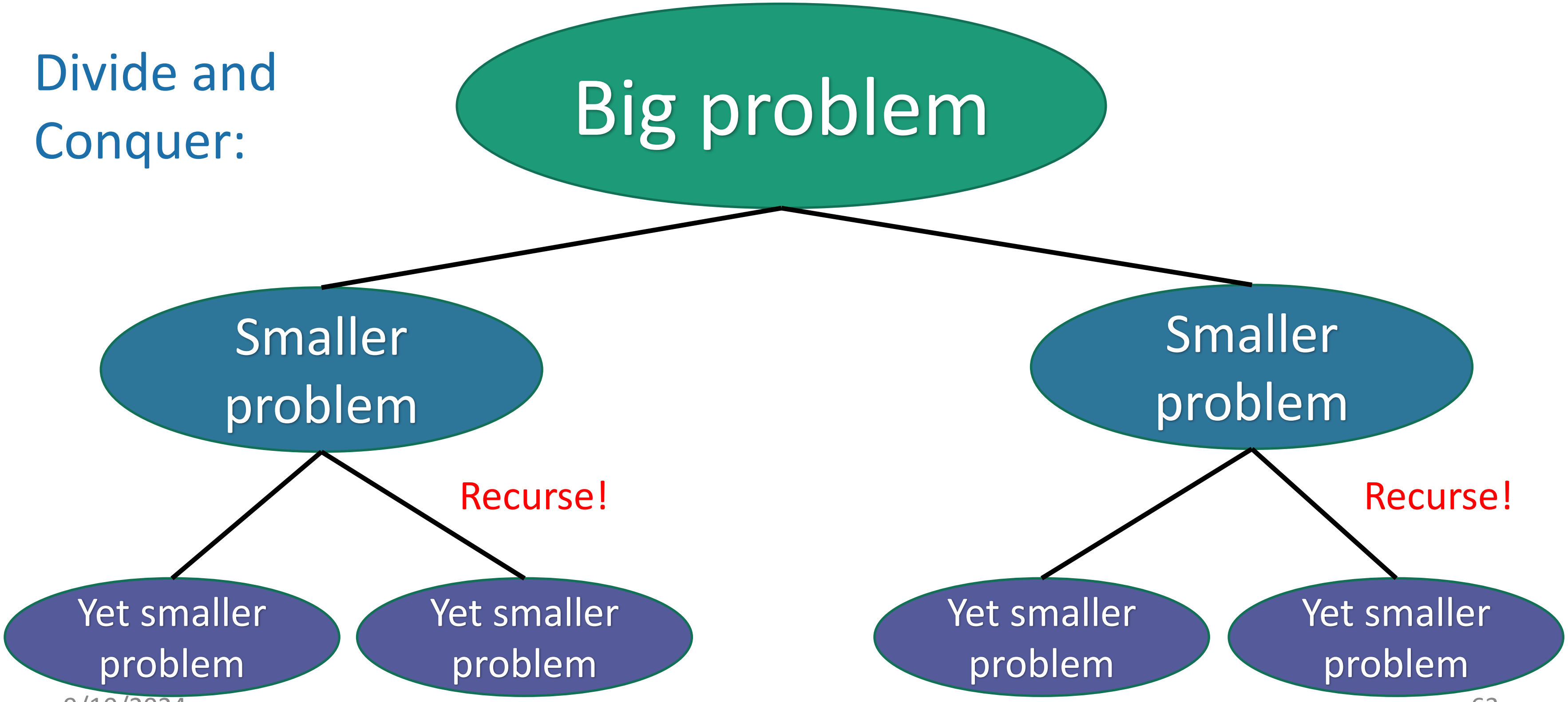
The Plan

- InsertionSort recap
- Worst-case analyisis
  - Back to InsertionSort: Does it work?
- Asymptotic Analysis
  - Back to InsertionSort: Is it fast?
- MergeSort
  - Does it work?
  - Is it fast?

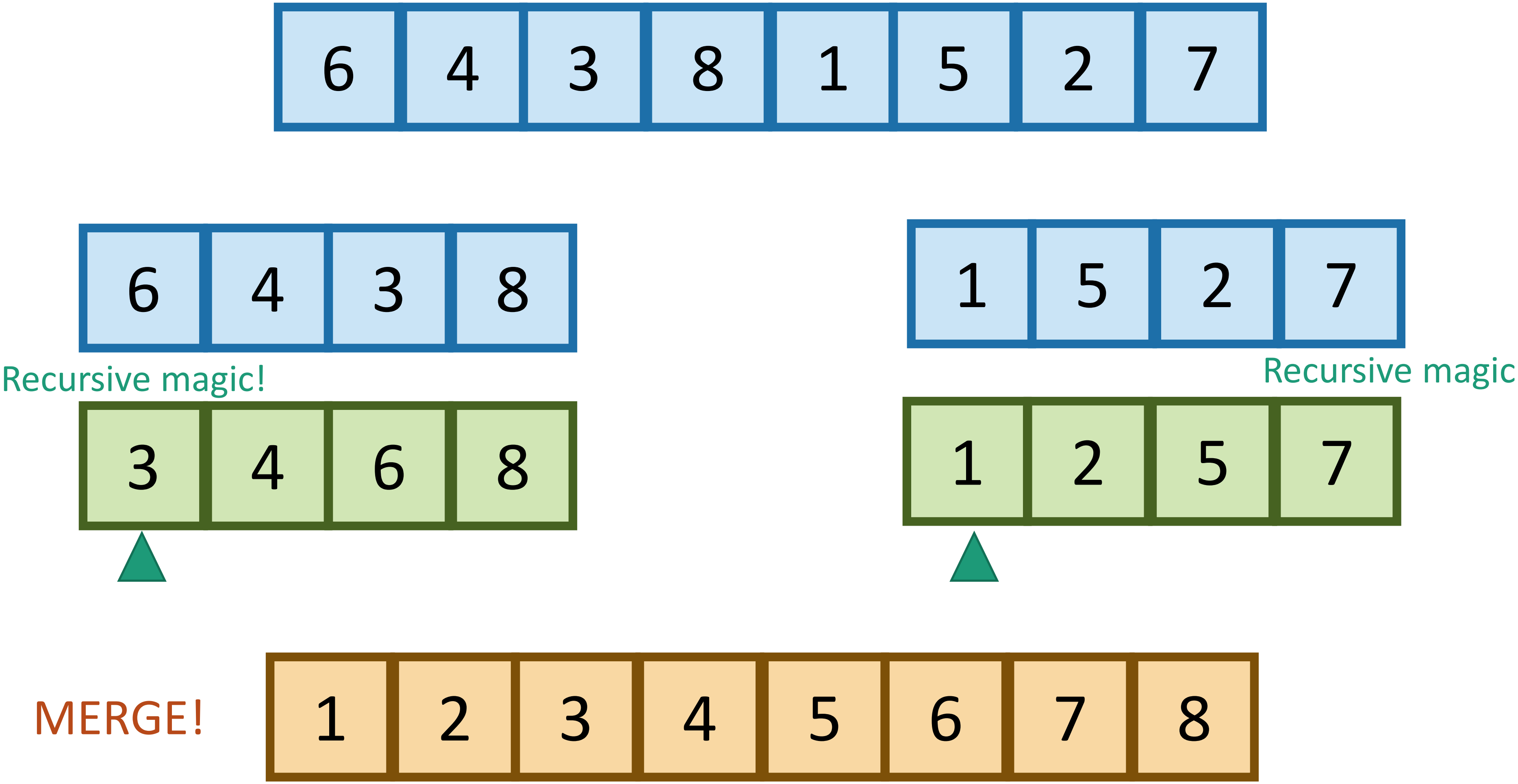
Can we do better?

- MergeSort: a divide-and-conquer approach
- 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

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

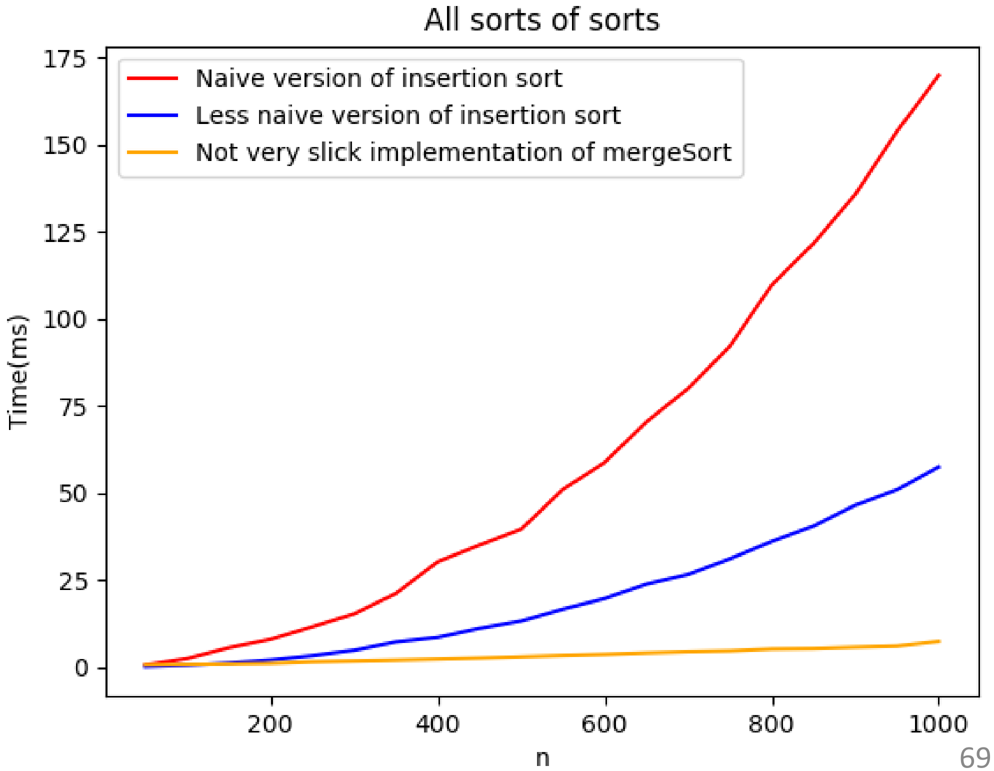
MergeSort Pseudocode

```
MERGE(A, p, q, r)
n_L = q – p + 1          // length of A[p : q]
n_R = r – q              // length of A[q + 1 : r]
let L[0 : n_L – 1] and R[0 : n_R – 1] be new arrays
for i = 0 to n_L – 1     // copy A[p : q] into L[0 : n_L – 1]
    L[i] = A[p + i]
for j = 0 to n_R – 1     // copy A[q + 1 : r] into R[0 : n_R – 1]
    R[j] = A[q + j + 1]
i = 0                    // i indexes the smallest remaining element in L
j = 0                    // j indexes the smallest remaining element in R
k = p                    // k indexes the location in A to fill
// As long as each of the arrays L and R contains an unmerged element,
//     copy the smallest unmerged element back into A[p : r].
while i < n_L and j < n_R
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
    k = k + 1
```

```
while i < n_L
    A[k] = L[i]
    i = i + 1
    k = k + 1
while j < n_R
    A[k] = R[j]
    j = j + 1
    k = k + 1
```

Two questions

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



Empirically:

1. Seems to work.
2. Seems fast.

Assume that n is a power of 2 for convenience.

It’s fast

CLAIM:

MergeSort runs in time $O(n \log(n))$

- Proof coming soon.
- But first, how does this compare to InsertionSort?
  - Recall InsertionSort ran in time $O(n^2)$.

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

Aside:

All logarithms in this course are base 2

Quick log refresher

- Def: log(n) is the number so that $2^{\log(n)} = n$.
- 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!

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

- $\log(n)$ grows much more slowly than n
- $n \log(n)$ grows much more slowly than n^2

Punchline: A running time of O(n log n) is a lot better than O(n^2)!

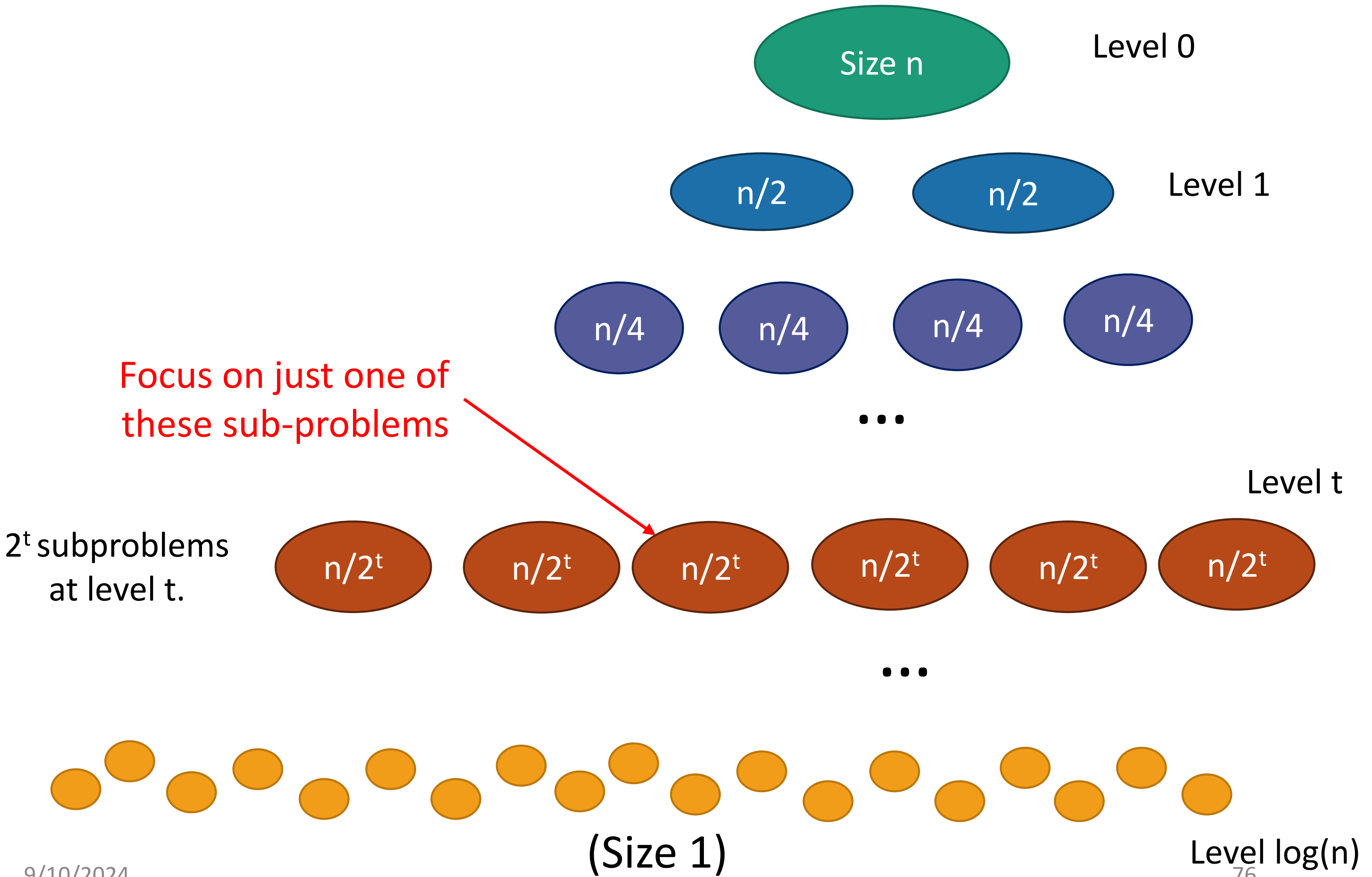
Assume that n is a power of 2 for convenience.

Now let's prove the claim

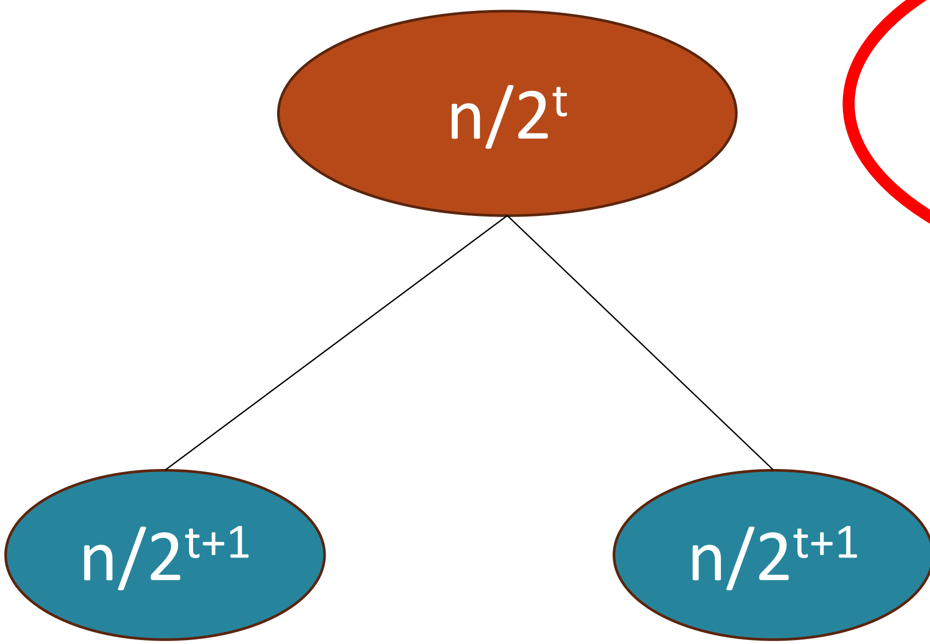
CLAIM:

MergeSort runs in time $O(n\log(n))$

Let’s prove the claim



How much work in this sub-problem?



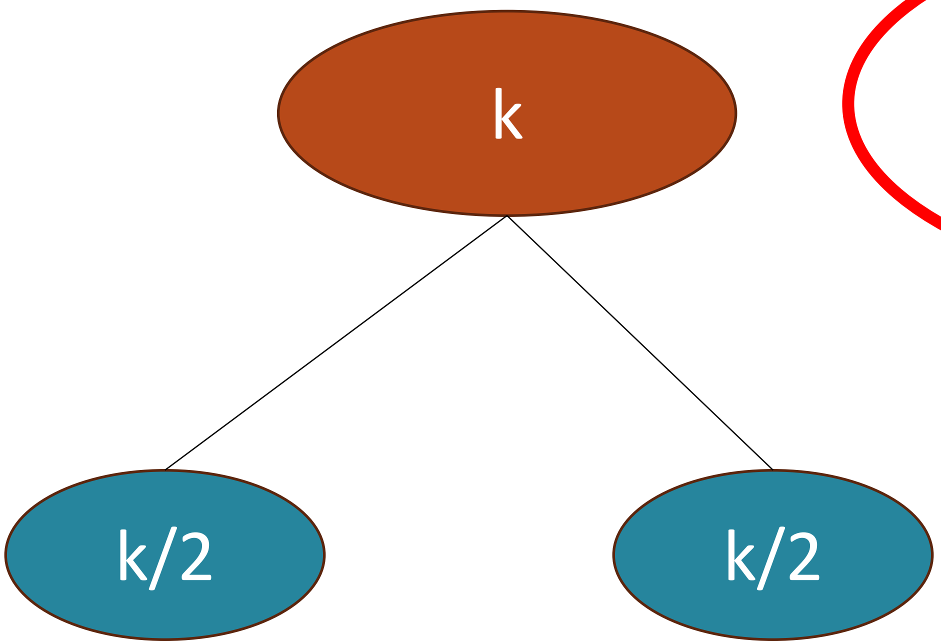
Time spent MERGE-ing the two subproblems

+

Time spent within the two sub-problems

How much work in this sub-problem?

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

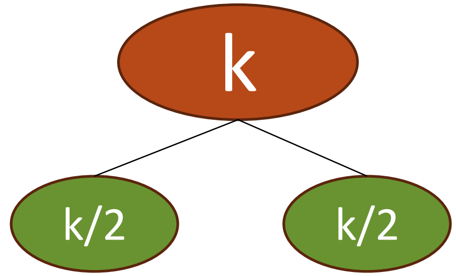


Time spent MERGE-ing the two subproblems

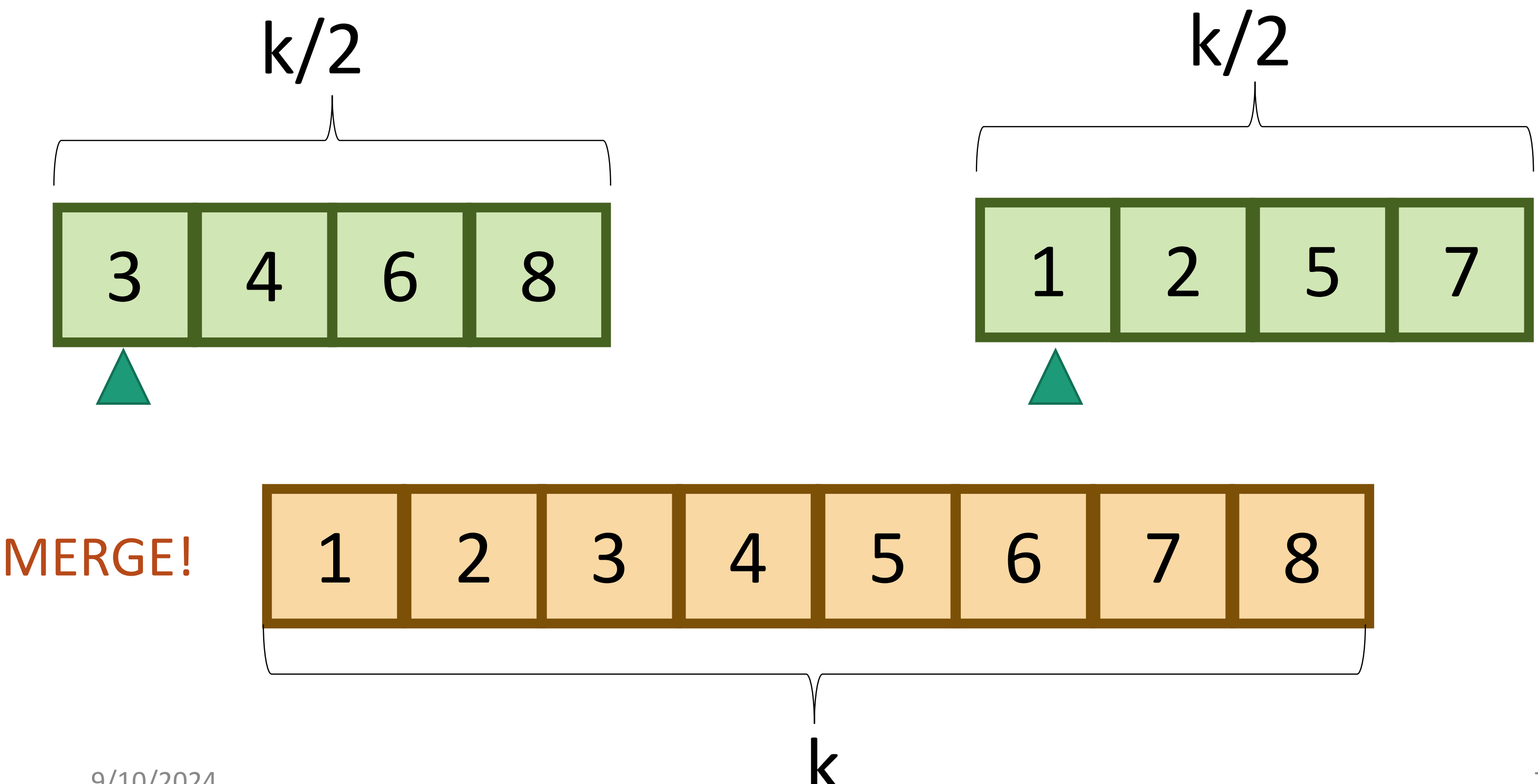
+

Time spent within the two sub-problems

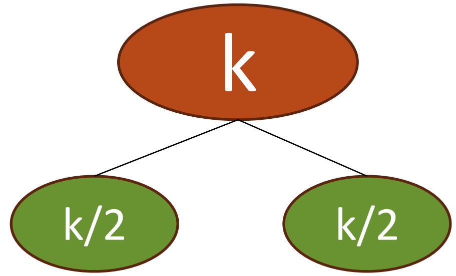
How long does it take to MERGE?



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



How long does it take to MERGE?

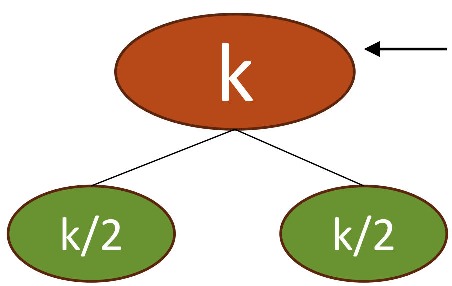


Code for the MERGE step is given in the Lecture2 notebook.

Question: in big-Oh notation, how long does it take to run MERGE on two lists of size k/2?

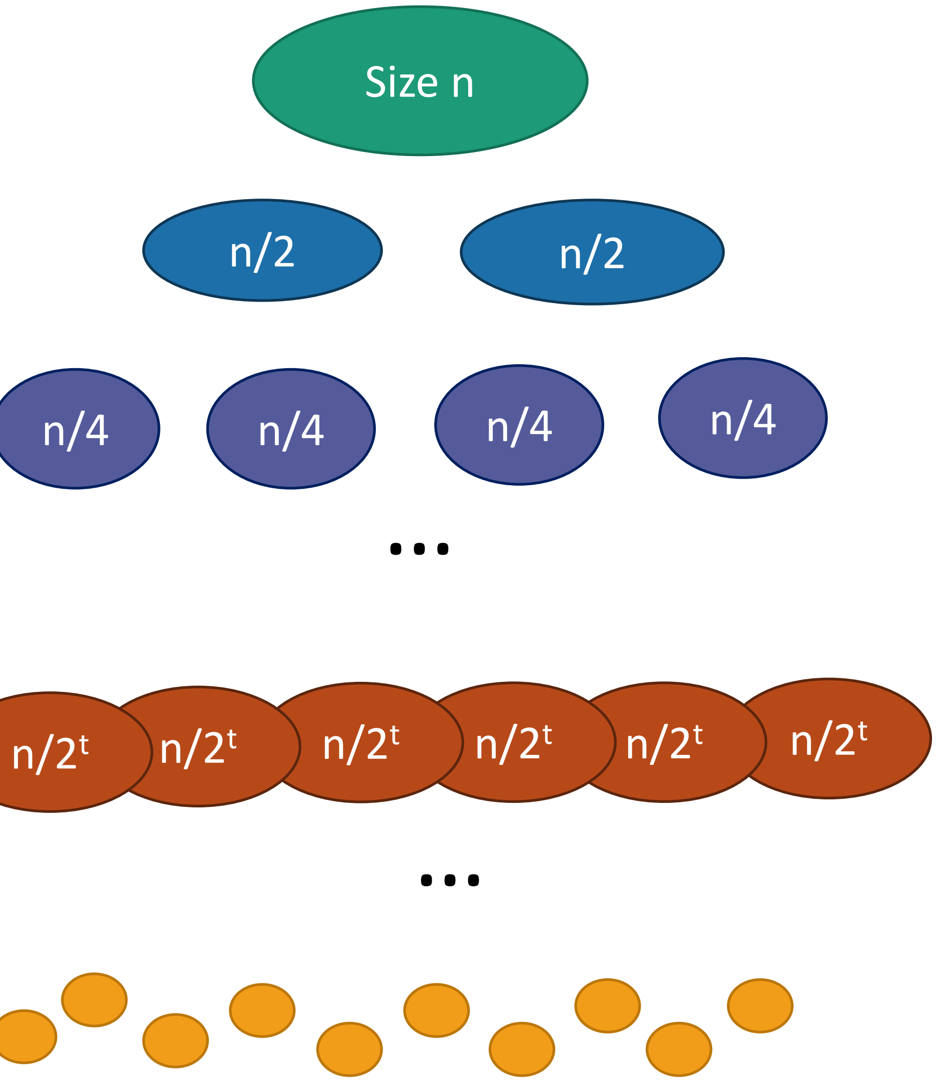
Answer: It takes time O(k), since we just walk across the list once.

Take-away:



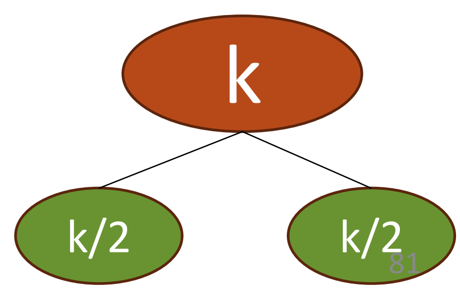
There are O(k) operations done at this node.
(Not including work at recursive calls).

Recursion tree

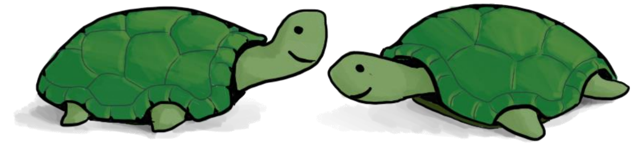


(Size 1)

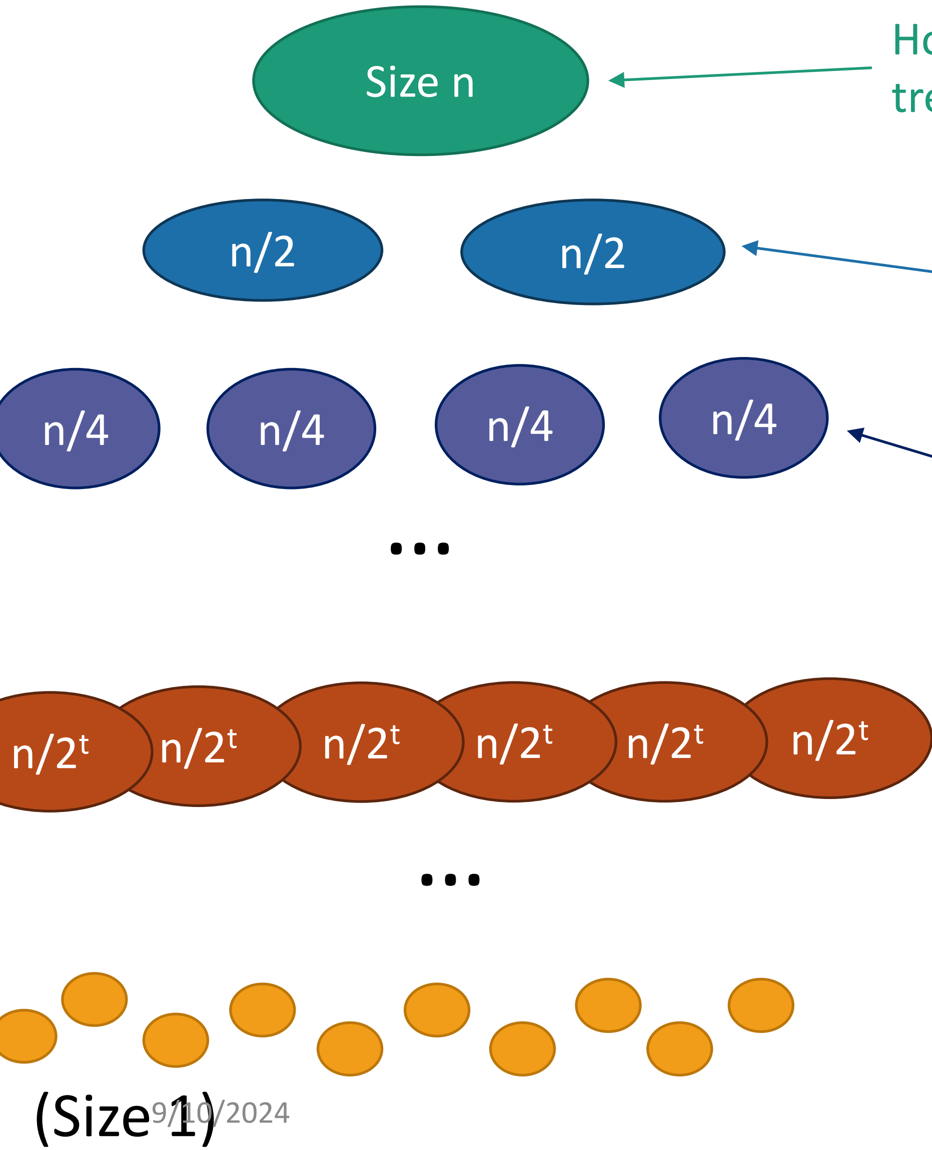
There are O(k) operations done at this node.



Recursion tree



Think, Pair, Share!



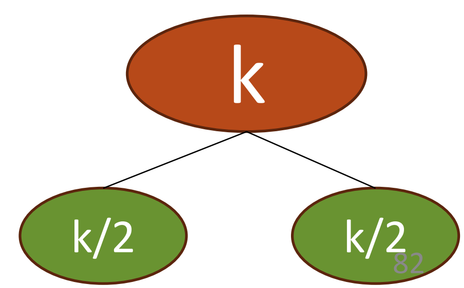
How many operations are done at this level of the tree? (Just MERGE-ing subproblems).

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

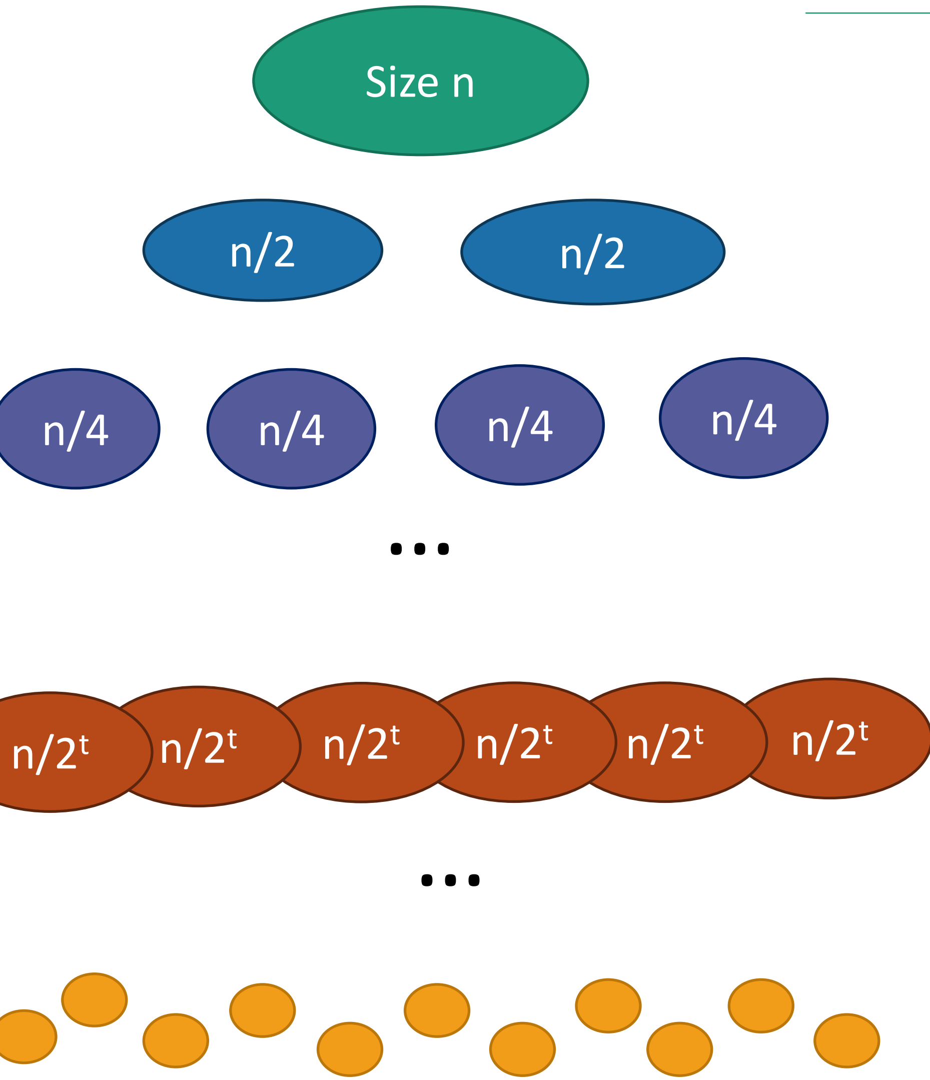
This level?

This level?

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



Recursion tree



(Size 1)

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)

Explanation for this table done on the board!

Total runtime…

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

That was the claim!

What have we learned?

- MergeSort correctly sorts a list of n integers in time O(n log(n)).
- That’s (asymptotically) better than InsertionSort!

The Plan

- InsertionSort recap
- Worst-case analyisis
    - Back to InsertionSort: Does it work?
- Asymptotic Analysis
    - Back to InsertionSort: Is it fast?
- MergeSort
    - Does it work?
    - Is it fast?

Wrap-Up

Recap

- InsertionSort runs in time $O(n^2)$
- MergeSort is a divide-and-conquer algorithm that runs in time $O(n \log(n))$

- How do we measure the runtime of an algorithm?
  - Worst-case analysis
  - Asymptotic analysis
- How do we analyze the running time of a recursive algorithm?
  - One way is to draw a recursion tree.