Indian Institute of Information Technology Allahabad

Data Structures and Algorithms

Introduction

Dr. Shiv Ram Dubey
Associate Professor
Department of Information Technology
Indian Institute of Information Technology, Allahabad

Email: srdubey@iiita.ac.in Web: https://profile.iiita.ac.in/srdubey/

DISCLAIMER

Books

1. Data Structures and Algorithm Analysis in C (DSAC) by Mark Allen Weiss, Second Edition
2. Data Structures, S. Lipschutz, Schaum’s Outline Series
3. Introduction to Algorithms by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, Third Edition, The MIT Press

… and many more options

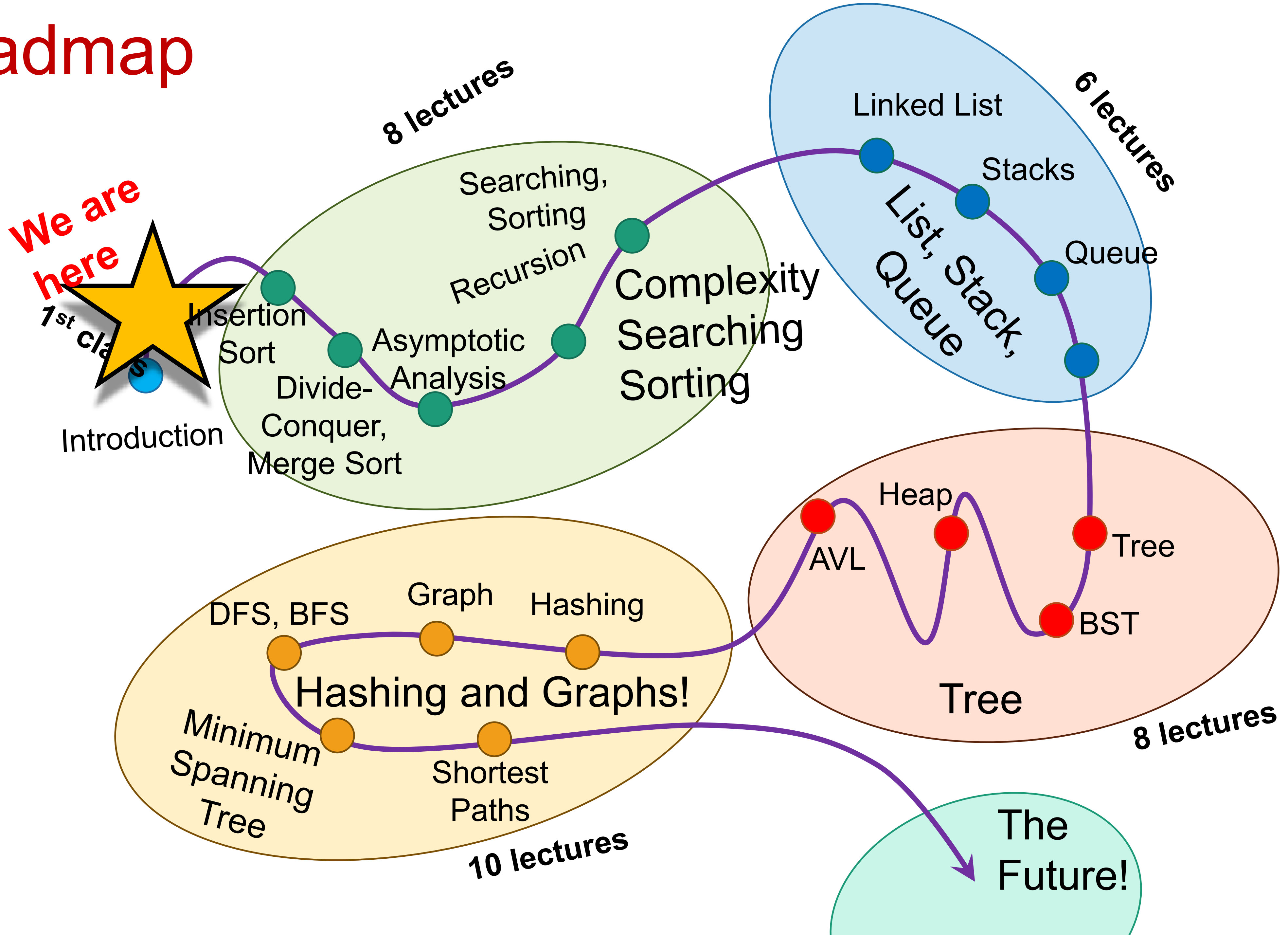
Language

- Only C language is allowed
  - It facilitates the development of better programming skills

Course Ethics

- All class work is to be done independently.
- It is best to try to solve problems on your own, since problem solving is an important component of the course, and exam problems are often based on the outcome of the assignment problems.
- You are allowed to discuss class material, assignment problems, and general solution strategies with your classmates. But, when it comes to formulating or writing solutions or writing codes, you must work alone.
- You are not allowed to take the codes from any source, including online, books, your classmate, etc. in the home works and exams.
- You may use free and publicly available sources (at idea level only), such as books, journal and conference publications, and web pages, as research material for your answers. (You will not lose marks for using external sources.)
- You may not use any paid service and you must clearly and explicitly cite all outside sources and materials that you made use of.
- I consider the use of uncited external sources as portraying someone else's work as your own, and as such it is a violation of the Institute's policies on academic dishonesty.
- Instances will be dealt with harshly and typically result in a failing course grade.
- Cheating cases will attract severe penalties.

Roadmap


We are here
1st class
Introduction
8 lectures
Searching, Sorting
Recursion
Insertion Sort
Asymptotic Analysis
Divide-Conquer, Merge Sort
Complexity Searching Sorting
6 lectures
Linked List
Stacks
Queue
List, Stack, Queue
Heap
AVL
Tree
BST
Tree
8 lectures
DFS, BFS
Graph
Hashing
Hashing and Graphs!
Minimum Spanning Tree
Shortest Paths
10 lectures
The Future!

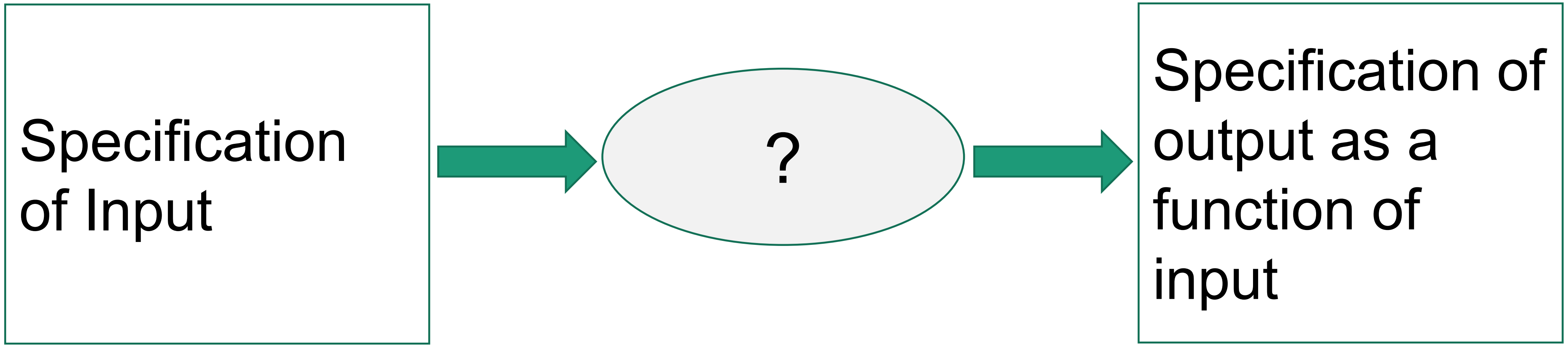
Data Structure and Algorithms

- **Algorithm:**
  - Outline, the essence of a computational procedure, step-by-step instructions

- **Program:**
  - An implementation of an algorithm in some programming language

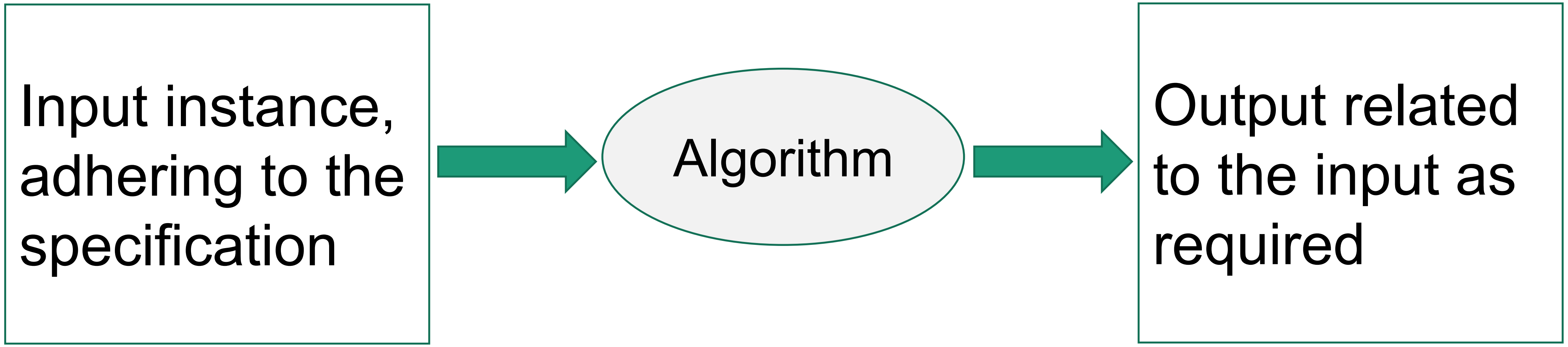
- **Data Structure:**
  - **Organization** of data needed to solve the problem

Algorithmic problem



- Infinite number of input instances satisfying the input specification.
- For eg: A sorted, non-decreasing sequence of natural numbers of non-zero, finite length:
  - 1, 15, 20, 300, 845, 9876
  - 56, 67, 100
  - 88

Algorithmic solution



Input instance, adhering to the specification
Algorithm
Output related to the input as required

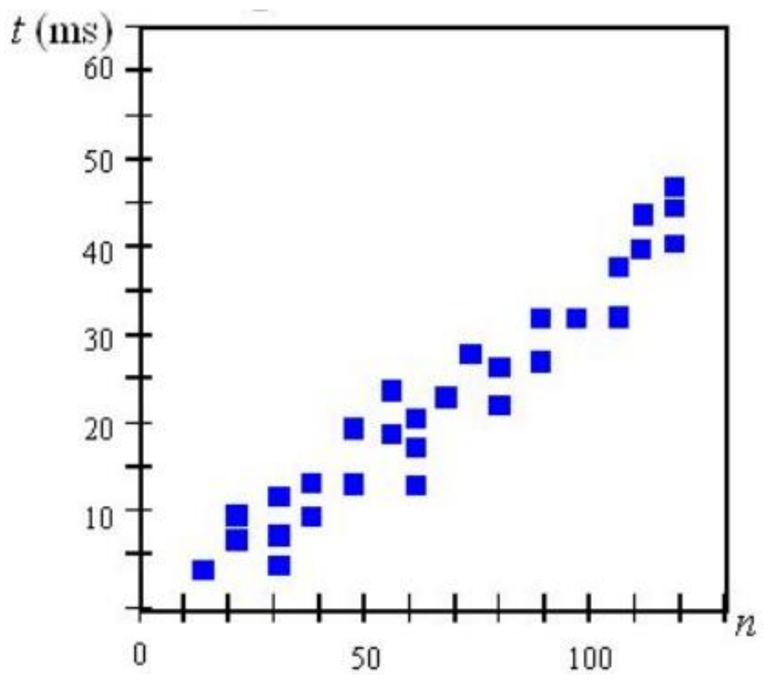
- Algorithm describes actions on the input instance
- Infinitely many correct algorithms for the same algorithmic problem

What is a good algorithm

- Efficient
  - Running time
  - Spaces used

- Efficiency as a function of input size
  - The number of bits in an input number
  - Number of data elements (numbers, points)

Measuring the Running Time



- How should we measure the running time of an algorithm?

- Experimental study
  - Write a program that implement the algorithm
  - Run the program with data sets of varying size and composition
  - Use the system time clock method to get an accurate measure of the actual running time

Limitations of Experimental Studies

- It is necessary to implement and test the algorithm in order to determine its running time.
- Experiments can be done on a limited set of inputs, and may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare to algorithms, the same hardware and software environments should be used.

Beyond Experimental Studies

We will develop a general methodology for analyzing running time of algorithms. This approach

- Uses a high-level description of the algorithm instead of testing its one of the implementations.
- Takes into account all possible inputs.
- Allows one to evaluate the efficiency of any algorithm in a way that is independent of the hardware and software environment.

Pseudo-Code

- A mixture of natural language and high-level programming concepts that describes the main ideas behind a generic implementation of a data structure or algorithm.
- Eg: Algorithm ArrayMax(A, n):

```
Input: An array A storing n integers.
Output: The maximum element in A.
currentMax = A[0]
for i = 1 to n-1 do
   if currentMax < A[i] then currentMax = A[i]
return currentMax
```

Pseudo-Code

It is more structured than usual prose, but less formal than a programming language.

- Expressions:
    - Use standard mathematical symbols to describe numeric and boolean expressions
- Method Declarations:
    - **Algorithm** name(param1, param2)

Pseudo-Code

- Programming Constructs:
  - decision structures: **if …. then …. [else ….]**
  - while-loops: **while …. do ….**
  - repeat-loops: **repeat …. until ….**
  - for-loop: **for … do ….**
  - array indexing: **A[i]**, **A[i,j]**
- Methods:
  - calls: object method(args)
  - returns: **return** value

[Only indicative, might be different at different places]

Analysis of Algorithms

- Primitive Operation: Low-level operation independent of programming language.
- Can be identified in pseudo-code. For eg:
  - Data movement (assignment)
  - Control (branch, subroutine call, return)
  - Arithmetic and logical operations (e.g., addition, comparison, etc.)
- By inspecting the pseudo-code, we can count the number of primitive operations executed by an algorithm.

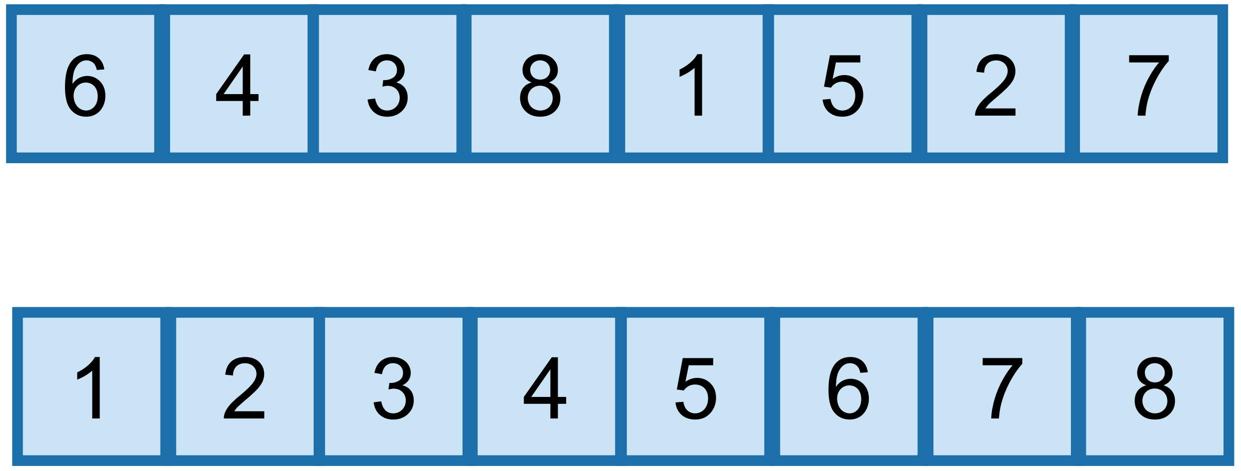
The plan

- Sorting Algorithms
    - InsertionSort: does it work and is it fast?
    - MergeSort: does it work and is it fast?
    - Skills:
        - Analyzing correctness of iterative and recursive algorithms.
        - Analyzing running time of recursive algorithms

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

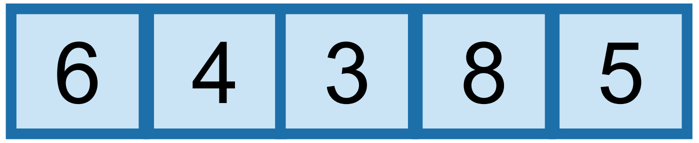
Sorting

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

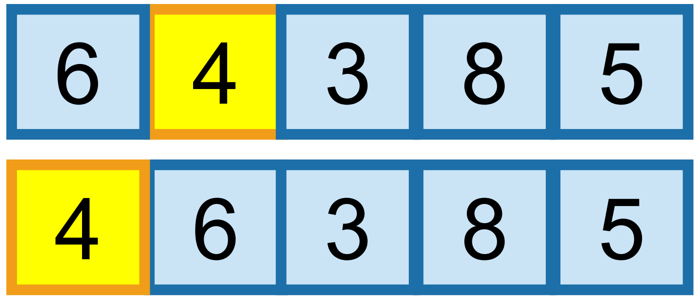


Insertion Sort

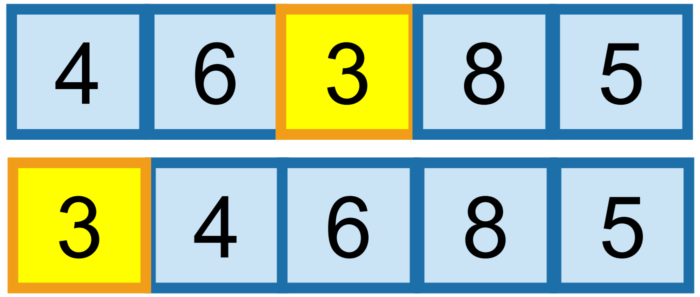
example



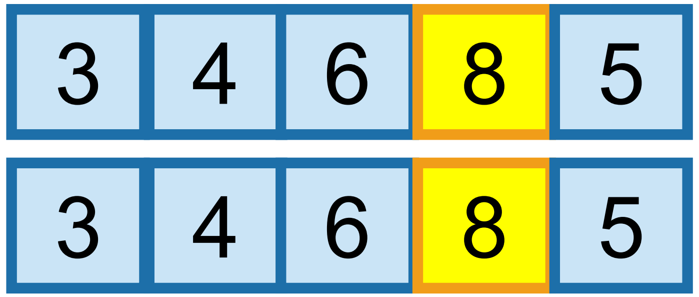
Start by moving A[1] toward the beginning of the list until you find something smaller (or can't go any further):



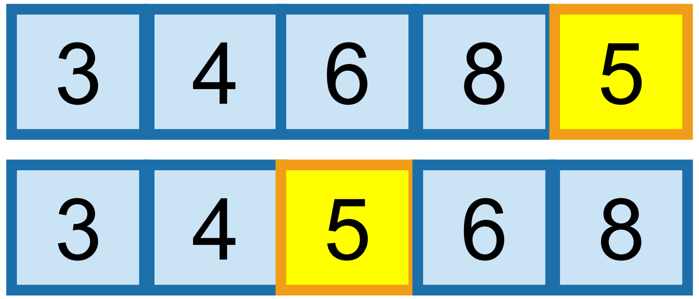
Then move A[2]:



Then move A[3]:



Then move A[4]:



Then we are done!

Insertion Sort

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

Insertion Sort

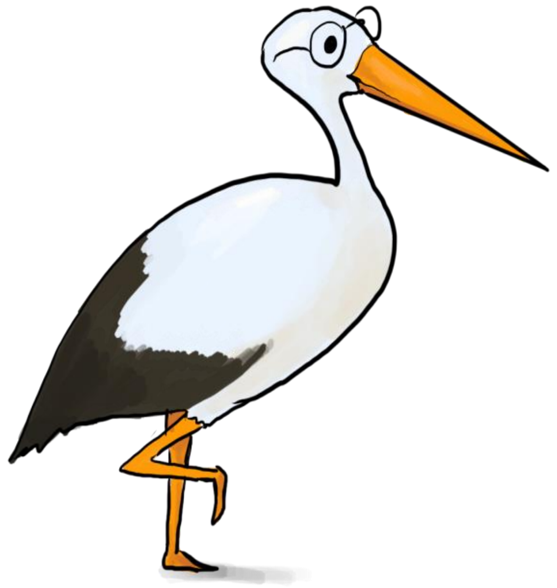
1. Does it work?
2. Is it fast?

Insertion Sort: Running Time

- Claim:
    - The running time is $O(n^2)$

Insertion Sort: Running Time

- Claim:
    - The running time is $O(n^2)$



Verify this!

Analysis of Insertion Sort

```
for j = 1 to n-1 do
    key = A[j]
    //Insert A[j] into the sorted
    //Sequence A[0,….,j-1]
    i = j-1
    while i>=0 and A[i]>key
        do A[i+1] = A[i]
             i--
    A[i+1] = key
```

Analysis of Insertion Sort

```
for j = 1 to n-1 do
    key = A[j]
    //Insert A[j] into the sorted
    //Sequence A[0,…,j-1]
    i = j-1
    while i>=0 and A[i]>key
        do A[i+1] = A[i]
             i--
    A[i+1] = key
```

n-1 iterations of the outer loop

Analysis of Insertion Sort

```
for j = 1 to n-1 do
    key = A[j]
    //Insert A[j] into the sorted
    //Sequence A[0,…,j-1]
    i = j-1
    while i>=0 and A[i]>key
        do A[i+1] = A[i]
             i--
    A[i+1] = key
```

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

n-1 iterations of the outer loop

Analysis of Insertion Sort

```
for j = 1 to n-1 do
    key = A[j]
    //Insert A[j] into the sorted
    //Sequence A[0,…,j-1]
    i = j-1
    while i>=0 and A[i]>key
        do A[i+1] = A[i]
            i--
    A[i+1] = key
```

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

n-1 iterations of the outer loop

Running time is $O(n^2)$

Analysis of Insertion Sort

	Cost	**times**
for j = 1 **to** n-1 **do**	c_1	n
key = A[j]	c_2	n-1
//Insert A[j] into the sorted	0	n-1
//Sequence A[0,…,j-1]	0	n-1
i = j-1	c_3	n-1
while i>=0 and A[i]>key	c_4	$\sum_{j=1}^{n-1} t_j$
do A[i+1] = A[i]	c_5	$\sum_{j=1}^{n-1}(t_j-1)$
i--	c_6	$\sum_{j=1}^{n-1}(t_j-1)$
A[i+1] = key	c_7	n-1

Total time = $n(c_1+c_2+c_3+c_7)+\sum_{j=1}^{n-1} t_j(c_4+c_5+c_6) - (c_2+c_3+c_5+c_6+c_7)$

Analysis of Insertion Sort

Total time = $n(c_1+c_2+c_3+c_7)+\sum_{j=1}^{n-1} t_j(c_4+c_5+c_6) - (c_2+c_3+c_5+c_6+c_7)$

Best case:

elements already sorted; $t_j = 1$, running time = f(n), i.e., *linear* time.

Worst case:

elements are sorted in inverse order; $t_j = j$, running time = $f(n^2)$, i.e., *quadratic* time.

Average case:

$t_j = j/2$, running time = $f(n^2)$, i.e., *quadratic* time.

Insertion Sort

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

Insertion Sort

1. Does it work? ⬅
2. Is it fast?

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

Insertion Sort

1. Does it work? ⬅
2. Is it fast?

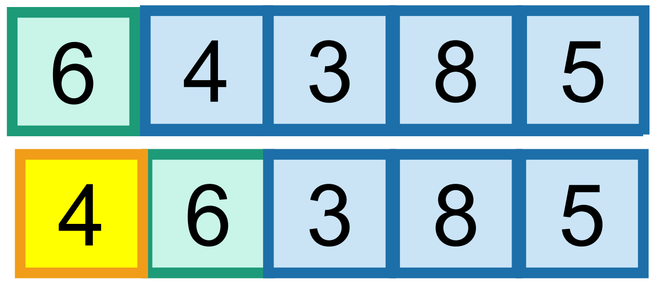
- Okay, so it's pretty obvious that it works.

- 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.

Insertion Sort: Why does this work?

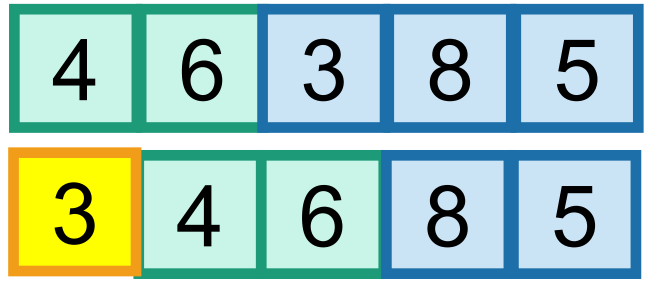
- Say you have a sorted list, $\boxed{3}\boxed{4}\boxed{6}\boxed{8}$, and another element $\boxed{5}$.
- Insert $\boxed{5}$ right after the largest thing that's still smaller than $\boxed{5}$. (Aka, right after $\boxed{4}$).
- Then you get a sorted list: $\boxed{3}\boxed{4}\boxed{5}\boxed{6}\boxed{8}$

Insertion Sort: So just use this logic at every step



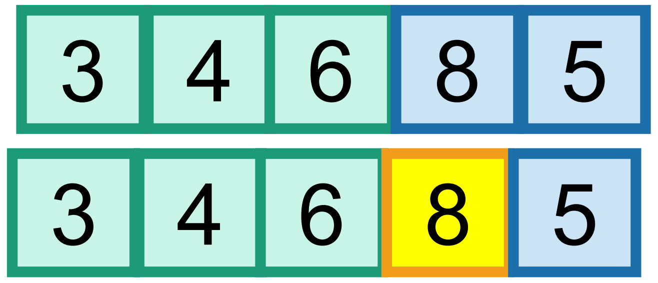
The first element, [6], makes up a sorted list.

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



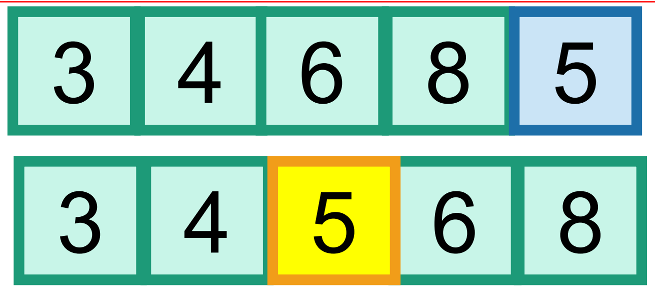
The first two elements, [4,6], make up a sorted list.

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!

Insertion Sort

Proof By Induction!

Recall: proof by induction

- Maintain a loop invariant.
- Proceed by induction.

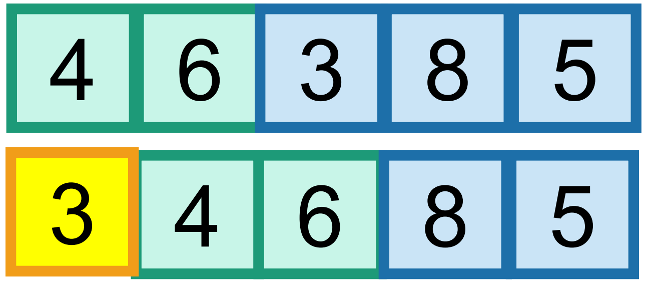
A loop invariant is something that should be true at every iteration.

- **Four steps in the proof by induction:**
  - Inductive Hypothesis: The loop invariant holds after the ith iteration.
  - Base case: the loop invariant holds before the 1st iteration.
  - Inductive step: If the loop invariant holds after the ith 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[0:i] is sorted.
- Inductive Hypothesis:
  - The loop invariant(i) holds at the end of the ith iteration (of the outer loop).
- Base case (i=0):
  - Before the algorithm starts, A[0] is sorted. ✓
- Inductive step:
  - If the inductive hypothesis holds at step i-1, it holds at step i
  - Aka, if A[0:i-1] is sorted at step i-1, then A[0:i] is sorted at step i
- Conclusion:
  - At the end of the n-1'st iteration (aka, at the end of the algorithm), A[0:n-1] = A is sorted.
  - That's what we wanted! ✓



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.

This was iteration i=2.

Correctness of Insertion Sort

- **Inductive hypothesis.** After iteration *i* of the outer loop, A[0:i] is sorted.
- **Base case.** After iteration 0 of the outer loop (aka, before the algorithm begins), the list A[0] contains only one element, and this is sorted.
- **Inductive step.** Suppose that the inductive hypothesis holds for i-1, so A[0:i-1] is sorted after the i-1'st iteration. We want to show that A[0:i] is sorted after the i'th iteration.
- Suppose that *k*th element is the largest integer in $\{0, . . . , i - 1\}$ such that $A[k] < A[i]$. Then the effect of the inner loop is to turn

$$[A[0], A[1], . . . , A[k], . . . , A[i - 1], A[i]]$$

into

$$[A[0], A[1], . . . , A[k], A[i], A[k + 1], . . . , A[i - 1]]$$

Correctness of Insertion Sort

- We claim that the following list is sorted:

  [A[0], A[1], . . . , A[k], A[i], A[k + 1], . . . , A[i − 1]]
- This is because A[i] > A[k], and by the inductive hypothesis, we have A[k] ≥ A[j] for all j ≤ k , and so A[i] is larger than everything that is positioned before it.
- Similarly, by the choice of *k* we have A[i] ≤ A[k + 1] ≤ A[j] for all j ≥ k + 1, so A[i] is smaller than everything that comes after it. Thus, A[i] is in the right place. All of the other elements were already in the right place, so this proves the claim.
- Thus, after the i'th iteration completes, A[0:i] is sorted, and this establishes the inductive hypothesis for i.

Correctness of Insertion Sort

- **Conclusion.** By induction, we conclude that the inductive hypothesis holds for all $i \leq n - 1$. In particular, this implies that after the end of the n−1’st iteration (after the algorithm ends) A[0:n-1] is sorted.
- Since A[0:n-1] is the whole list, this means the whole list is sorted when the algorithm terminates, which is what we were trying to show.

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?

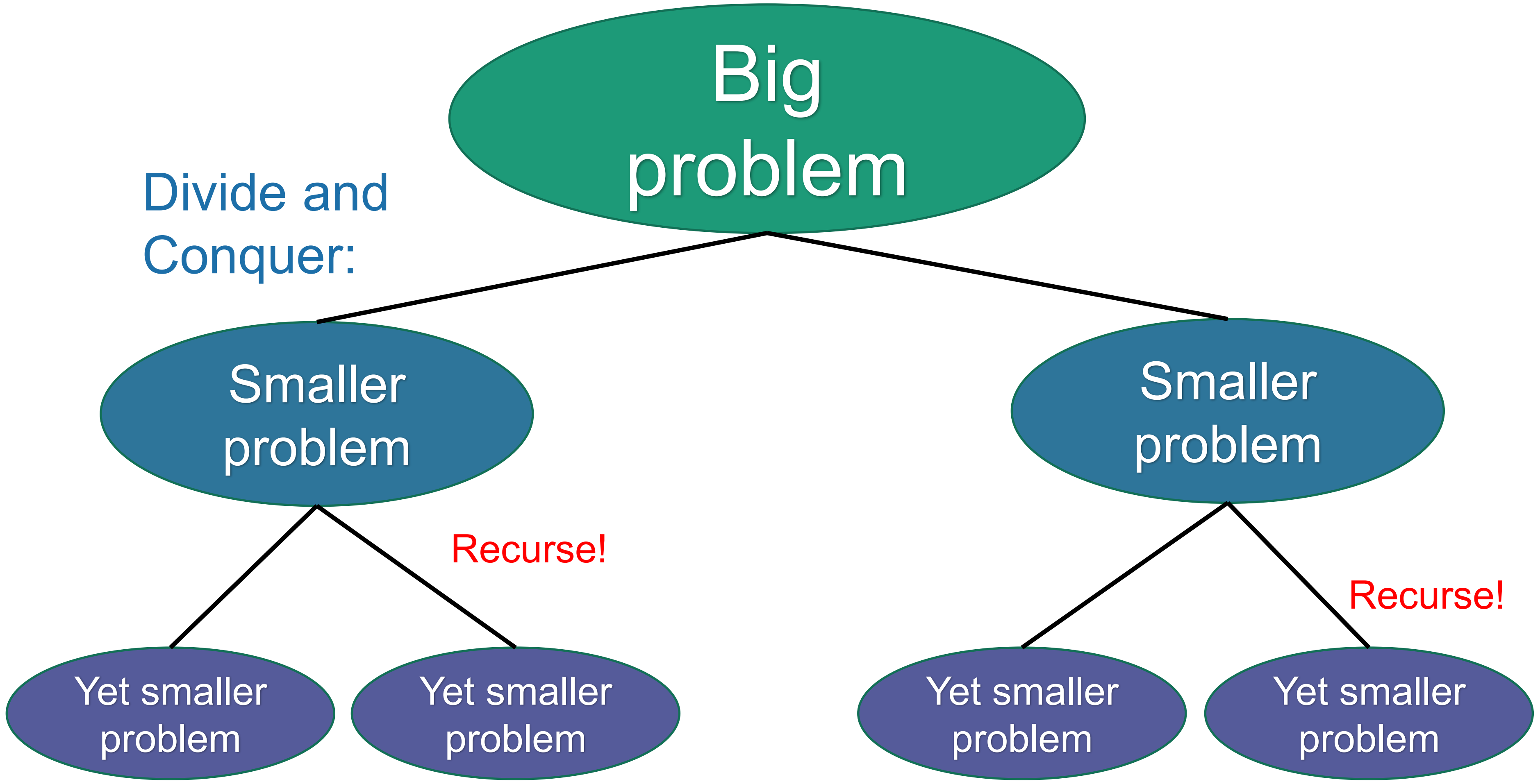
The plan

- Sorting Algorithms
  - InsertionSort: does it work and is it fast?
  - MergeSort: does it work and is it fast?
  - Skills:
    - Analyzing correctness of iterative and recursive algorithms.
    - Analyzing running time of recursive algorithms

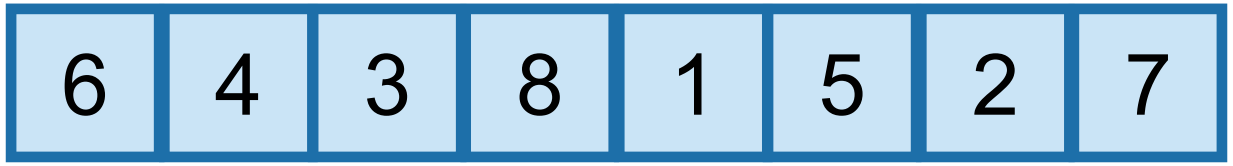
- How do we measure the runtime of an algorithm?
  - Worst-case analysis
  - Asymptotic Analysis

Can we do better?

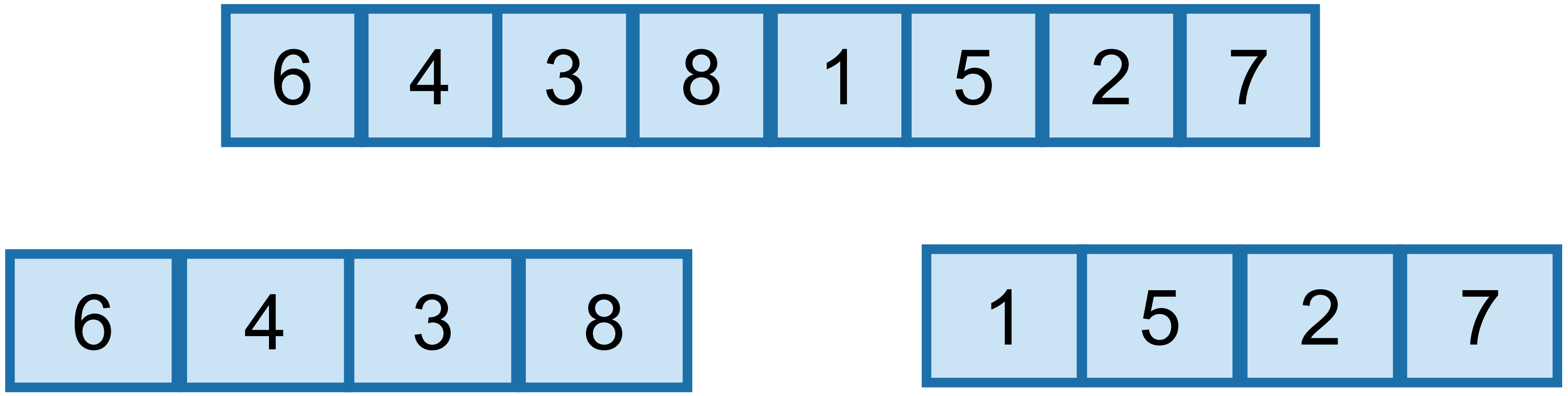
- MergeSort: a divide-and-conquer approach



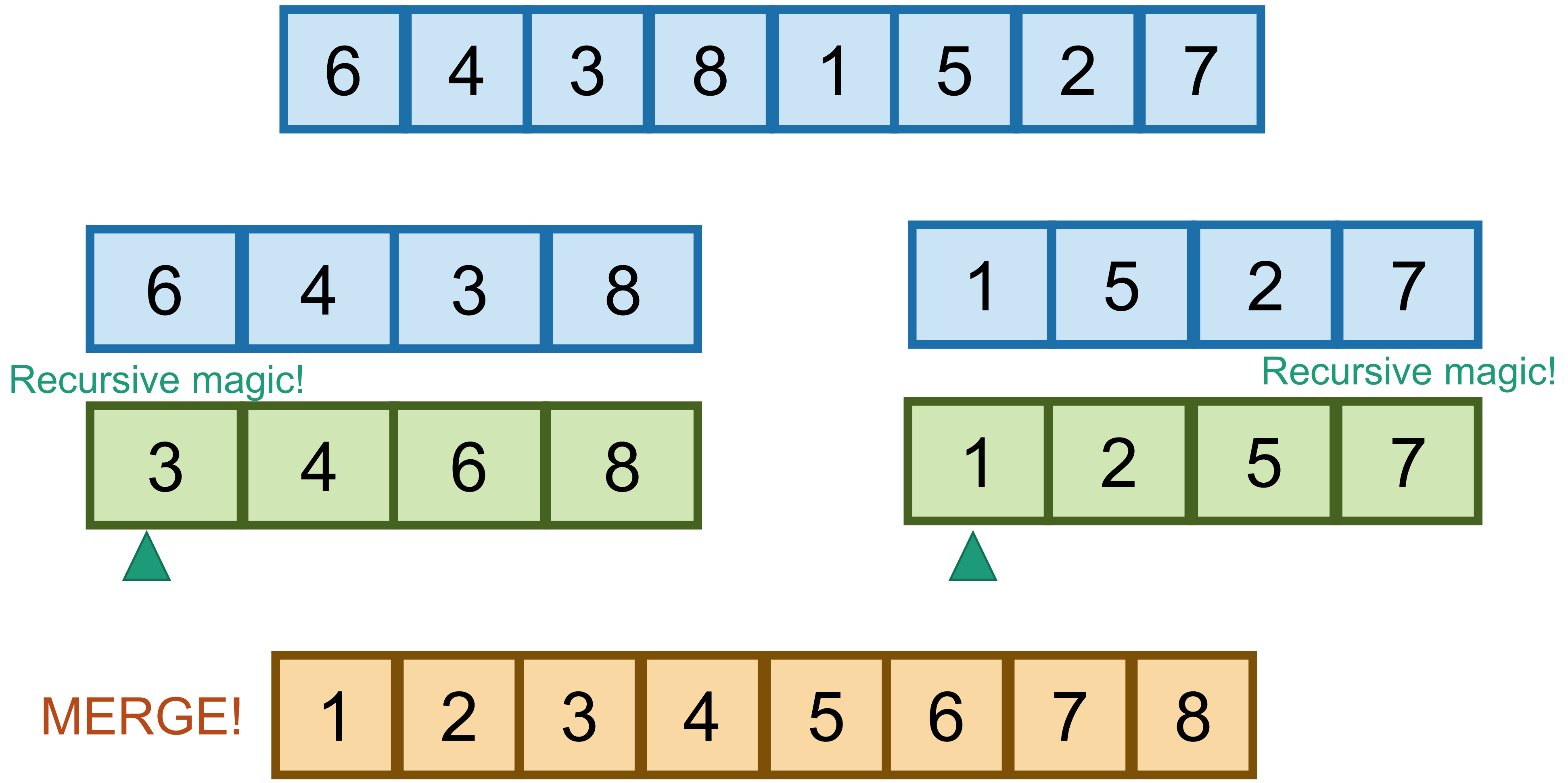
Can we do better?



Can we do better?



Can we do better?

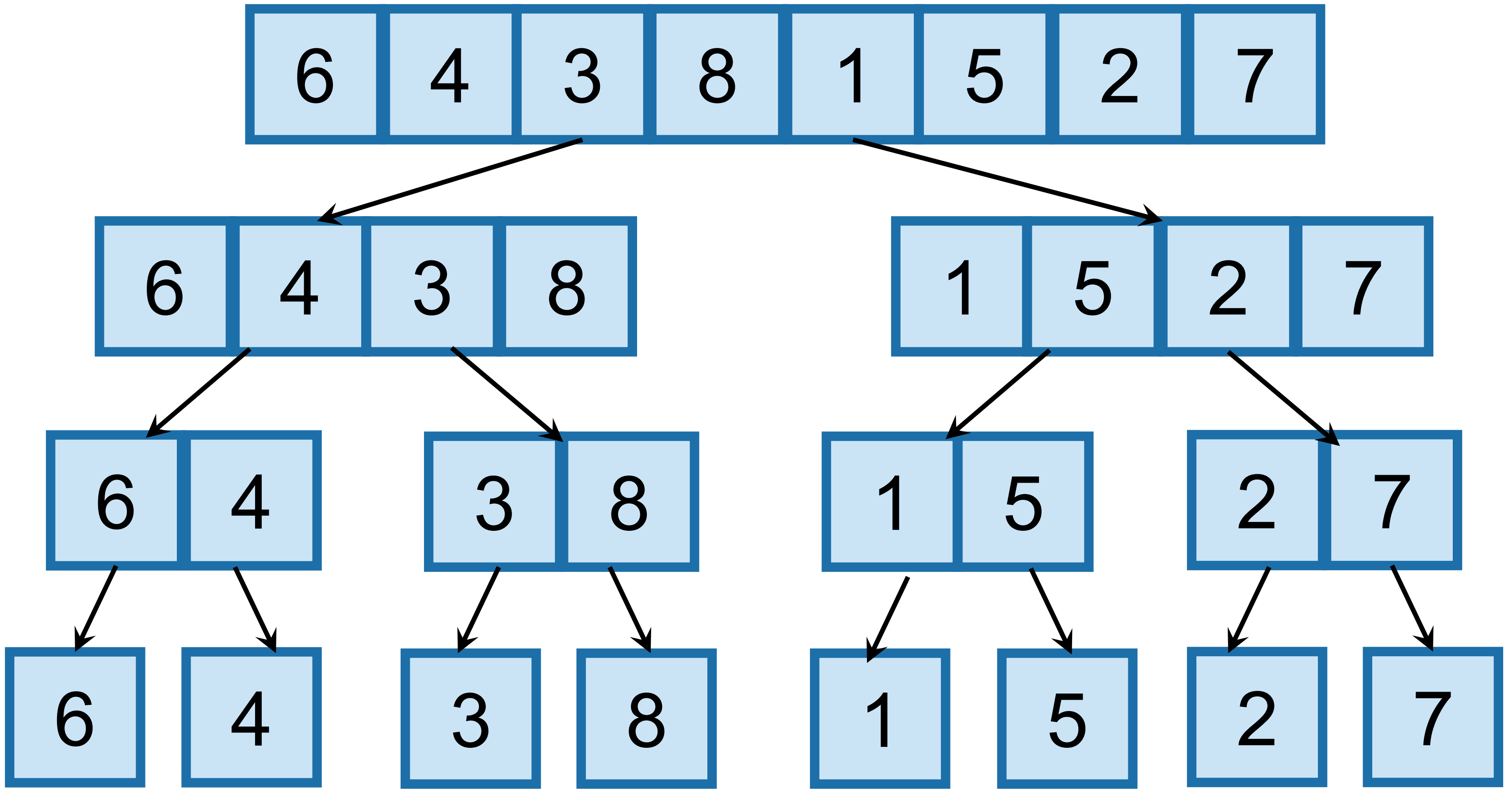


MergeSort Pseudocode

```
MERGESORT(A):
  • n = length(A)
  • if n ≤ 1:                               If A has length 1,
      • return A                            It is already sorted!
  • L = MERGESORT(A[0 : (n/2)-1])           Sort the left half
  • R = MERGESORT(A[n/2 : n-1])             Sort the right half
  • return MERGE(L,R)                       Merge the two halves
```

What actually happens?

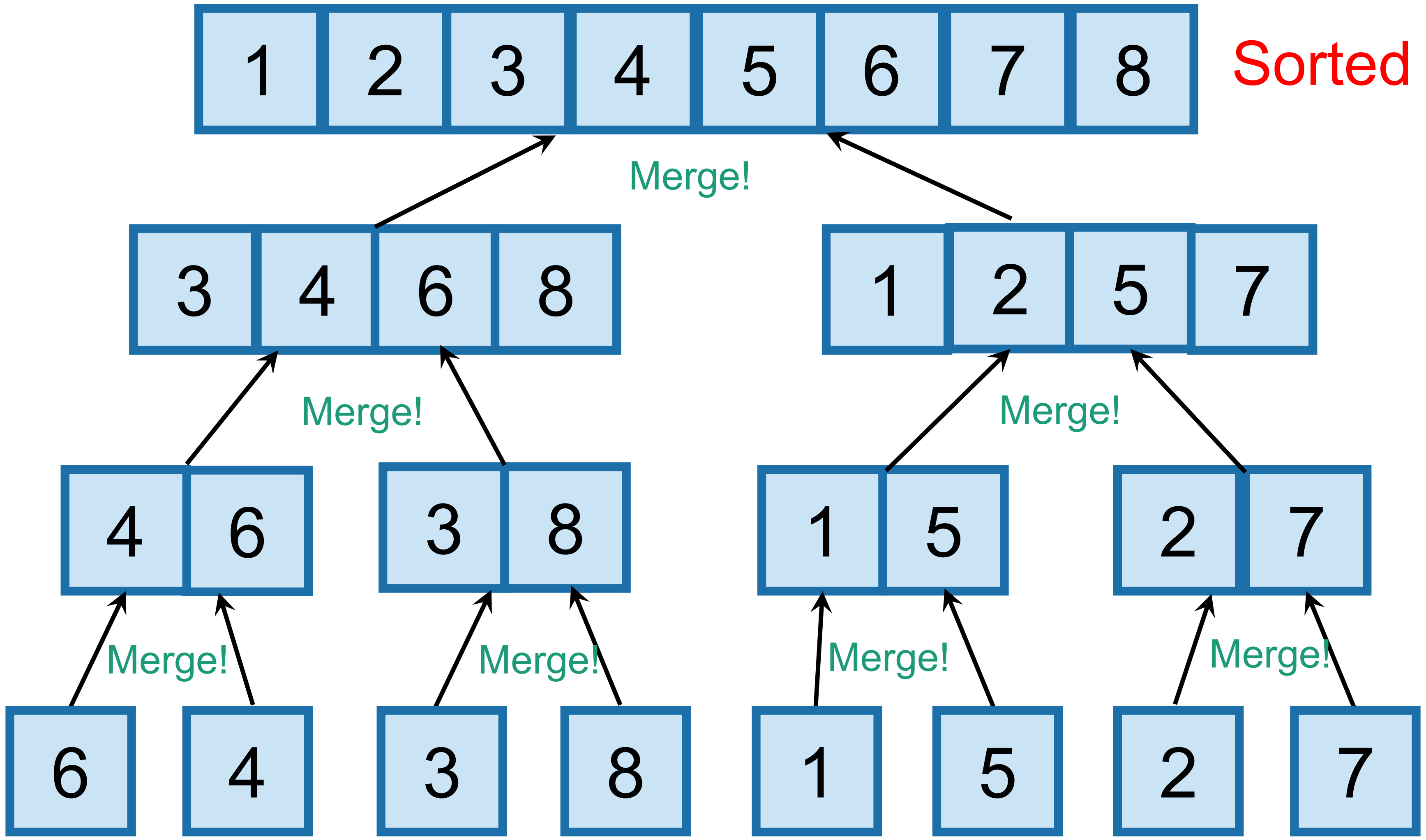
First, recursively break up the array all the way down to the base cases



This array of length 1 is sorted!

What actually happens?

Then, merge them all back up!



Sorted sequence!

A bunch of sorted lists of length 1 (in the order of the original sequence).

MergeSort

Two questions

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

Empirically:

1. Seems to work.
2. Seems fast.

MergeSort: It works

Assume that n is a power of 2 for convenience.

- **Inductive hypothesis:** "In every recursive call on an array of length at most i, MERGESORT returns a sorted array."
- **Base case (i=1):** a 1-element array is always sorted.
- **Inductive step:** Need to show: If L and R are sorted, then MERGE(L,R) is sorted.
- **Conclusion:** In the top recursive call, MERGESORT returns a sorted array.

```
• MERGESORT(A):
    • n = length(A)
    • if n ≤ 1:
        • return A
    • L = MERGESORT(A[0 : (n/2)-1])
    • R = MERGESORT(A[n/2 : n-1])
    • return MERGE(L,R)
```

MergeSort: It’s fast

Assume that n is a power of 2 for convenience.

CLAIM:

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

- How does this compare to InsertionSort?
  - Recall InsertionSort used on the order of n^2 operations.

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

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

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

Aside:

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.

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

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

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

- log(n) grows very slowly!

MergeSort: It’s fast

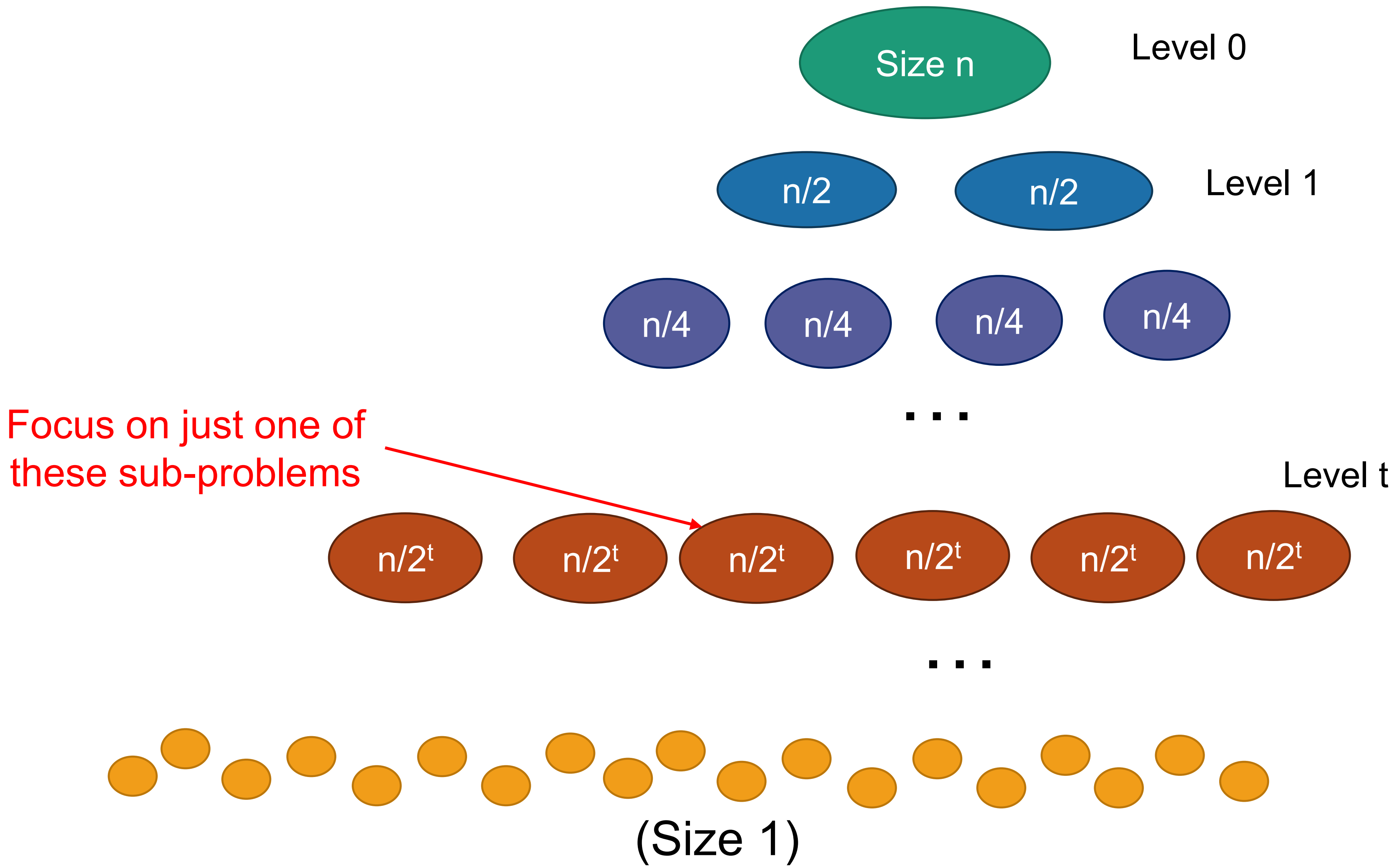
Assume that n is a power of 2 for convenience.

Now let’s prove the claim

CLAIM:

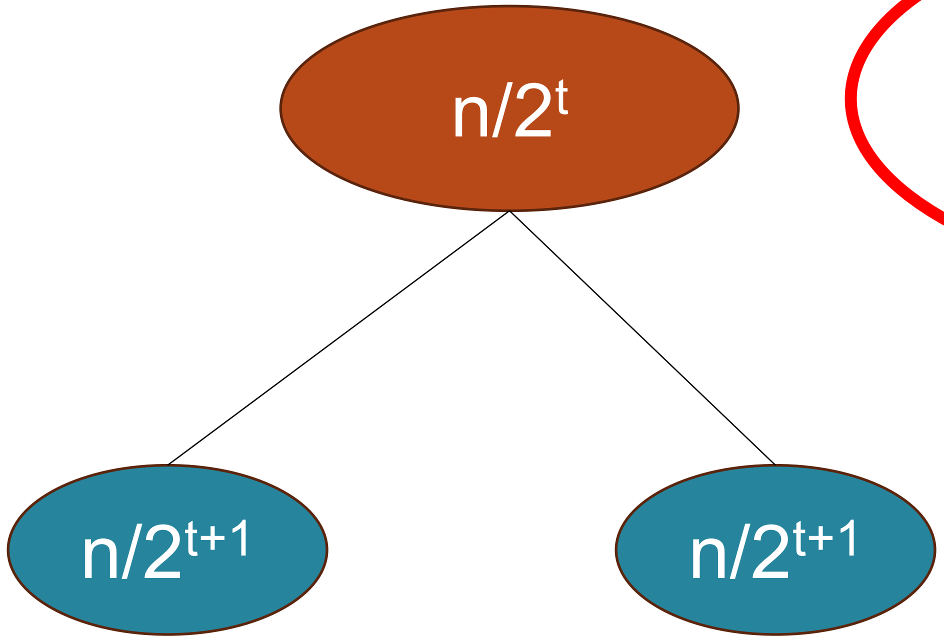
MergeSort requires at most c*n (log(n) + 1) operations to sort n numbers.

MergeSort: Let's prove the claim



MergeSort: Let’s prove the claim

How much work in this sub-problem?



Time spent MERGE-ing the two subproblems

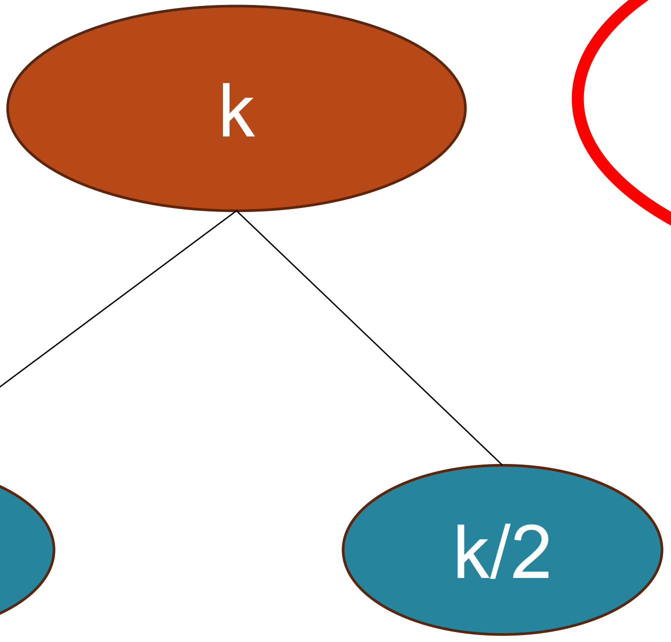
+

Time spent within the two sub-problems

MergeSort: Let's prove the claim

How much work in this sub-problem?

Let $k=n/2^t$…



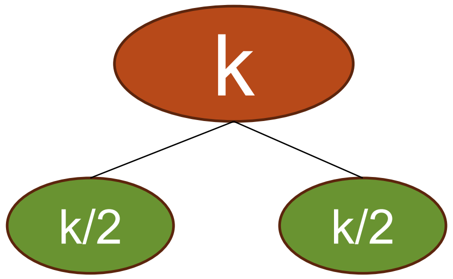
Time spent MERGE-ing the two subprojects

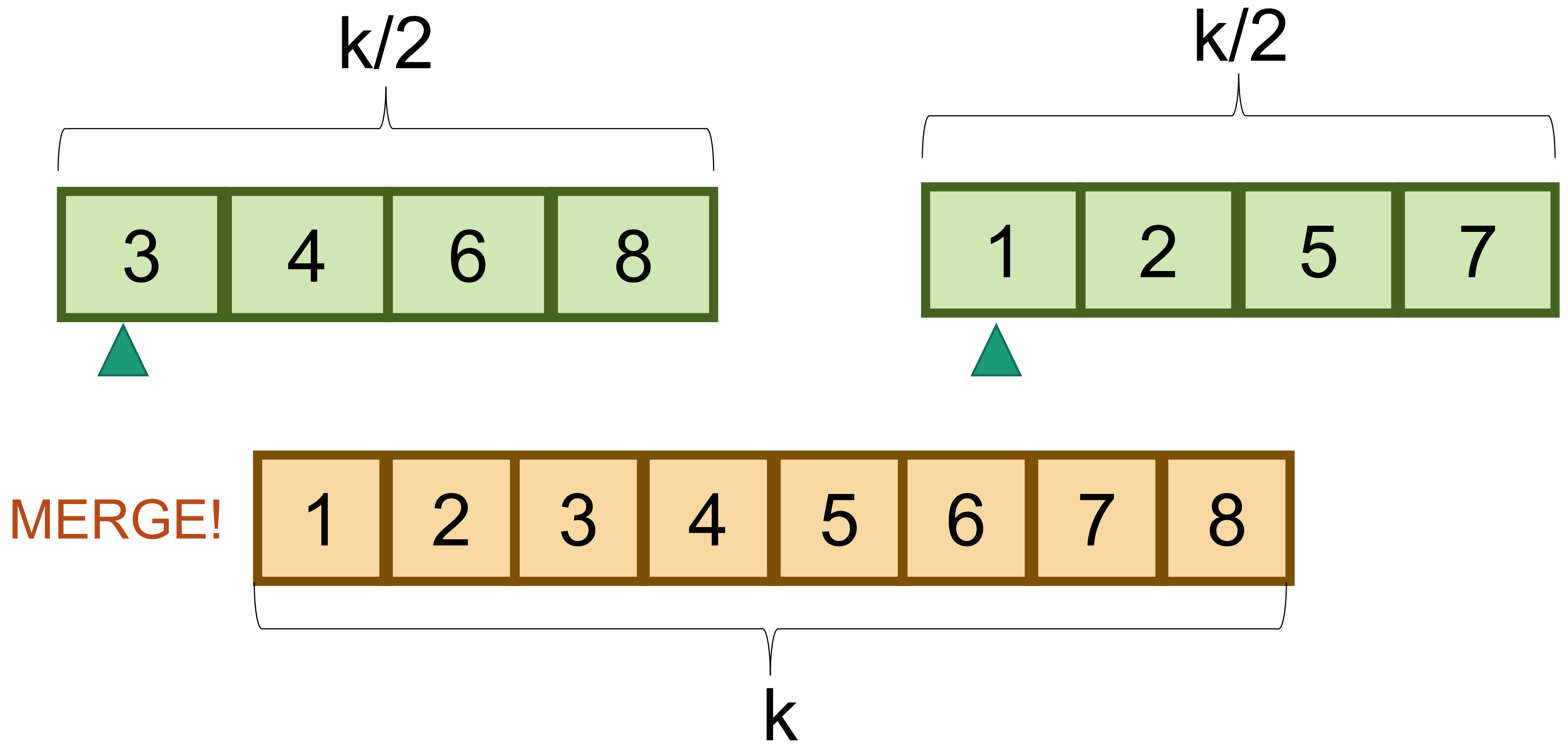
+

Time spent within the two sub-problems

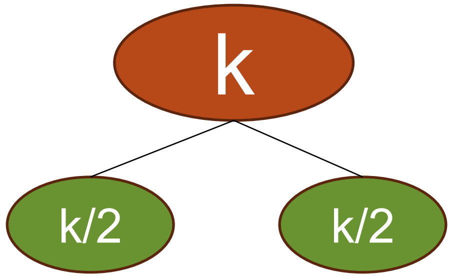
MergeSort

How long does it take to MERGE?



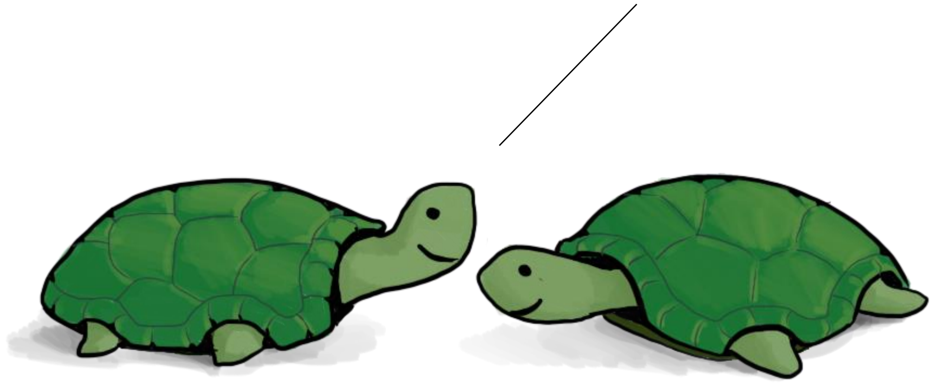


MergeSort

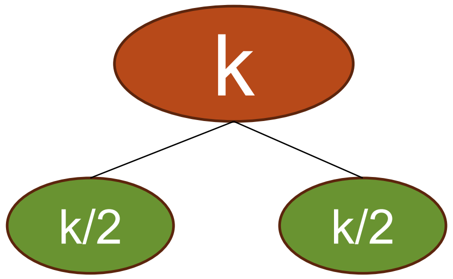


How long does it take to MERGE?

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



MergeSort

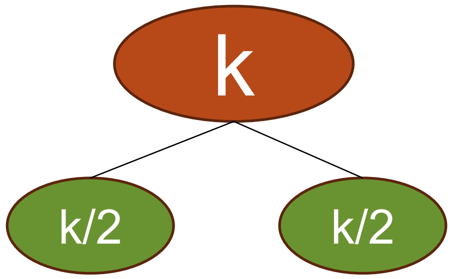


How long does it take to MERGE?

- 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…

MergeSort

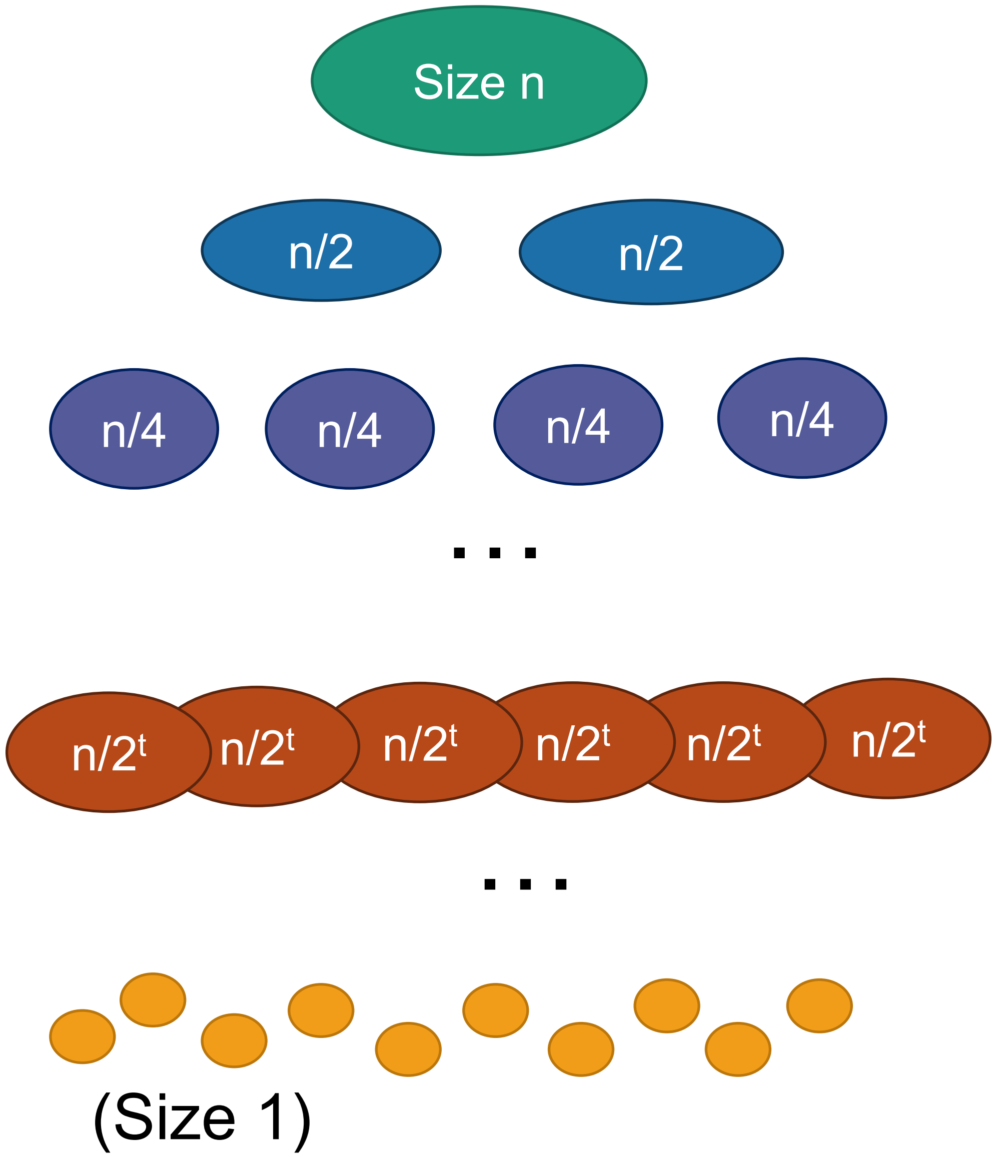
How long does it take to MERGE?



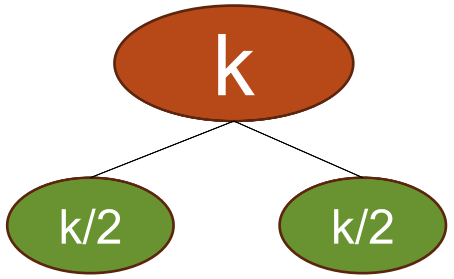
- Time to initialize an array of size k
- Plus the time to initialize three counters
- Plus the time to increment two of those counters k/2 times each
- Plus the time to compare two values at least k times
- Plus the time to copy k values from the existing array to the big array.
- Plus…

Let's say no more than c*k operations.

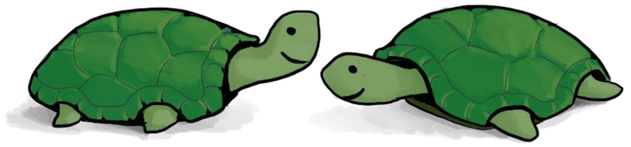
MergeSort: Recursion tree

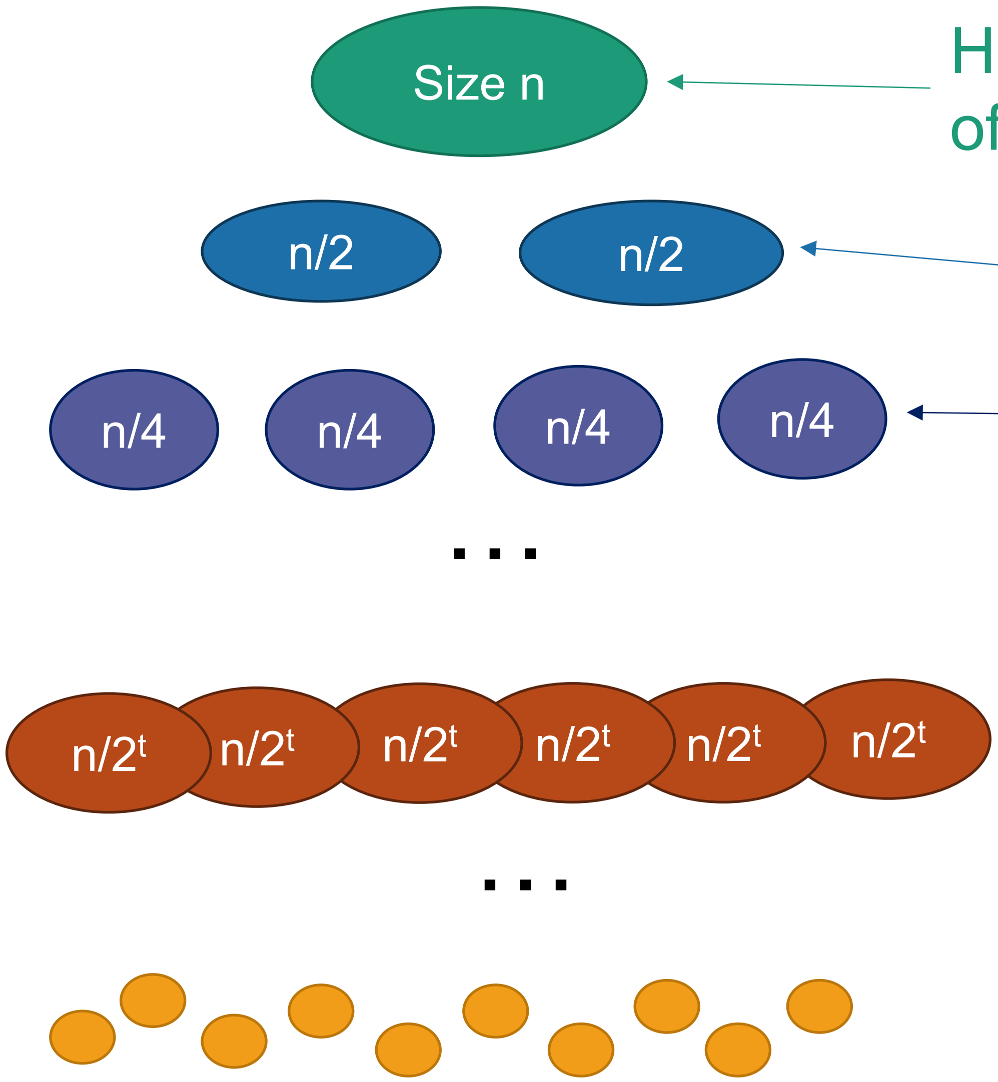


There are c*k operations done at this node.



MergeSort: Recursion tree





(Size 1)

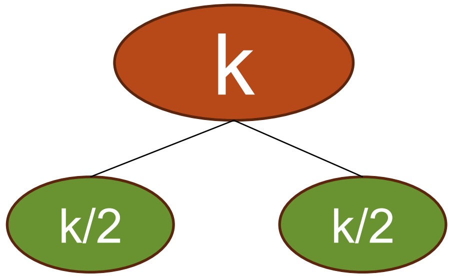
How many operations are done at this level of the tree? (Just MERGE-ing subproblems).

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

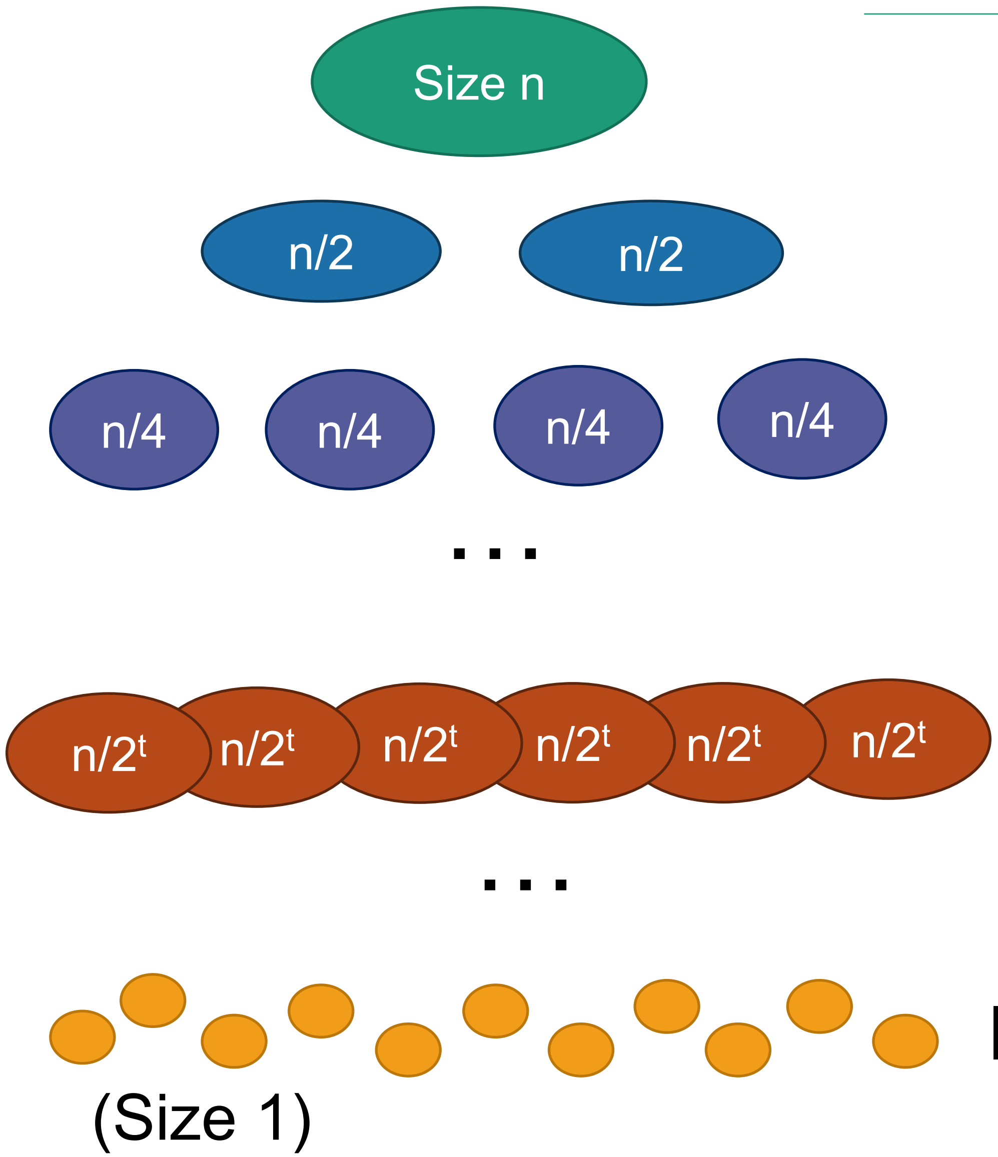
This level?

This level?

There are c*k operations done at this node.



MergeSort: Recursion tree



Level	# problems	Size of each problem	Amount of work at this level
0	1	n	c*n
1	2	n/2	c*n
2	4	n/4	c*n
. . .			
t	2^t	$n/2^t$	c*n
. . .			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	$2*n \tilde{=} c*n$

MergeSort: Total runtime…

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

That was the claim!

What have we learned?

- MergeSort correctly sorts a list of n integers in at most c*n(log(n) + 1) operations.
  - c is roughly 11

A few reasons to be grumpy

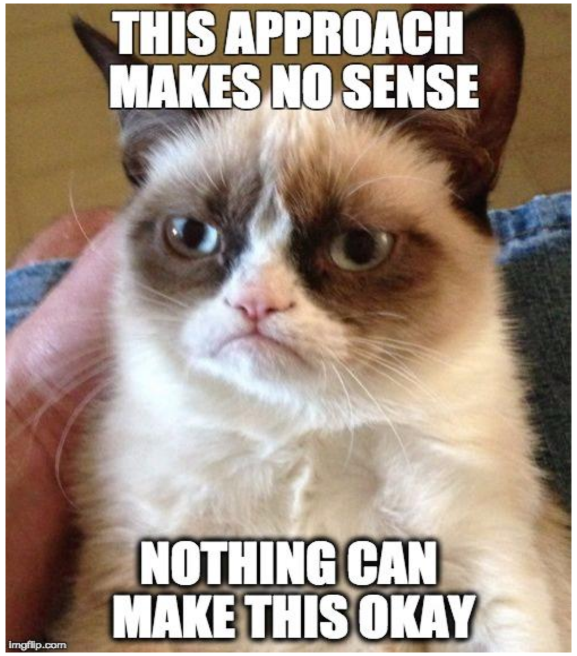
- Sorting

1	2	3	4	5	6	7	8

should take zero steps…

How we will deal with **grumpiness**

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



Acknowledgement

- Stanford University
- IIT Delhi

Thank You