Advanced Data Structures and Algorithms

Introduction

Topics – Unit 1

- **Introduction:**
  - **Refresher:** Abstract Data Types, Asymptotic Analysis, Stacks, Queues, Linked Lists, Trees, Searching and Sorting
  - **Recurrence Relation:** Recurrence Relation, Tree Method, Master Theorem, Substitution Method
  - **Sorting:** Radix and Bucket Sort
  - **Balanced Search Trees:** 2-3-4 Tree and Red Black Tree
  - **String Matching and Indexing:** Tries and Suffix Trees

Topics – Unit 2

- **Graph Algorithms:**
  - **Graphs:** Graphs, Data Structures for Graphs
  - **Breadth First Search:** Breadth First Search, Applications of BFS
  - **Depth First Search:** Depth First Search, Applications of DFS
  - DFS in Directed Graphs, Applications of DFS in Directed Graphs,
  - **Minimum Spanning Trees:** Kruskal's and Prim's Algorithm
  - **Shortest Paths:** Single Source Shortest Paths- Dijkstra and Bellman Ford
  - **All Pairs Shortest Paths**- Floyd Warshall

Topics – Unit 3

- **Advanced Algorithmic Design Techniques:**
  - **Dynamic Programming:** Fibonacci Number, Bellman Ford, Floyd Warshall, Longest Common Subsequences, Knapsack, Independent Sets in Trees
  - **Network Flows:** Max flow, Min Cut and Applications
  - **Randomized Algorithms:** Introduction, Quicksort, Min Cut Problem - Karger and Karger-Stein Algorithms
  - **NP-completeness:** NP-completeness, Polytime reductions

Books/References

- *Introduction to Algorithms* by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, Third Edition, The MIT Press
- *Algorithms Design* by Jon Kleinberg and Eva Tardos
- *Data Structures and Algorithm Analysis in C. Second Edition. - Mark Allen Weiss* ***(DSAC)***
- *Data structures and network algorithms*, Robert Endre Tarjan, Society for Industrial and Applied Mathematics Philadelphia, PA, USA, 1983, ISBN:0-89871-187-8
- *Knapsack Problems - Algorithms and Computer Implementations*, Silvano Martello and Paolo Toth, John Wiley & Sons, West Sussex, UK, 1990, ISBN: 0471924202

Course Website

Current Offering (Fall 2020):

https://sites.google.com/site/iiitsadsa/fall2020

Previous Offering (Fall 2019):

https://sites.google.com/site/iiitsadsa/fall2019

Tentative Evaluation Policy

- 15% Mid-Exam-1
- 15% Mid-Exam-2
- 30% End-Exam
- 20% Lab Exams
- 20% Quiz/Assignments

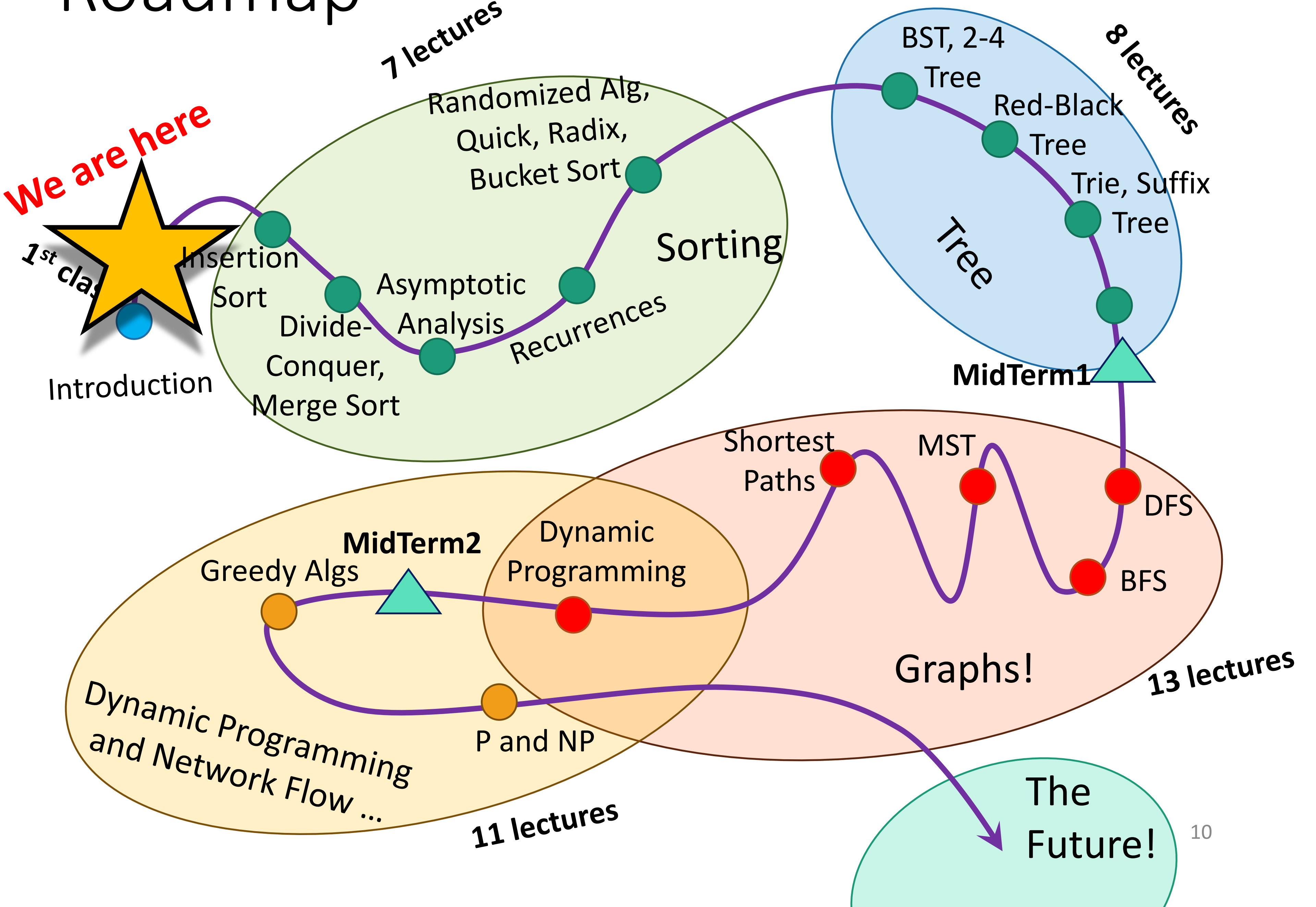
Language

- 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



7 lectures
We are here
1st class
Introduction
Insertion Sort
Divide-Conquer, Merge Sort
Asymptotic Analysis
Recurrences
Randomized Alg, Quick, Radix, Bucket Sort
Sorting
8 lectures
BST, 2-4 Tree
Red-Black Tree
Trie, Suffix Tree
Tree
MidTerm1
Shortest Paths
MST
DFS
BFS
Graphs!
13 lectures
MidTerm2
Greedy Algs
Dynamic Programming
P and NP
Dynamic Programming and Network Flow …
11 lectures
The Future!

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

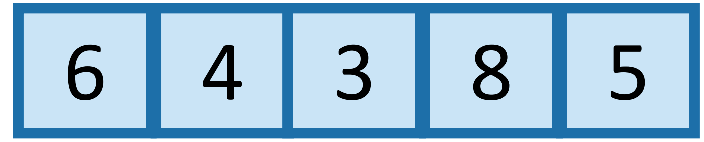
- Important primitive
- For today, we'll pretend all elements are distinct.

6	4	3	8	1	5	2	7

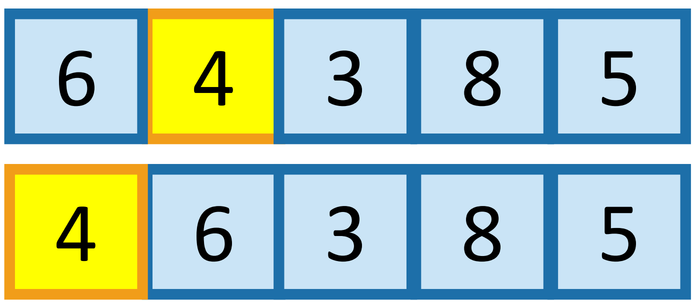
1	2	3	4	5	6	7	8

InsertionSort

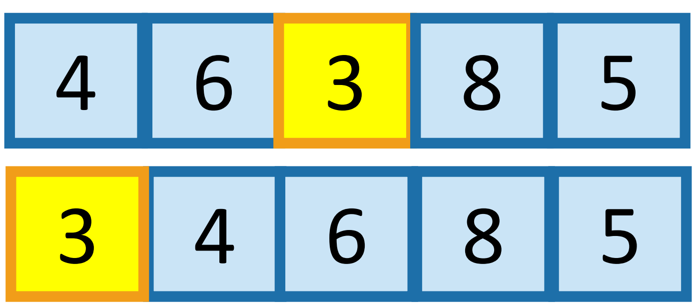
example



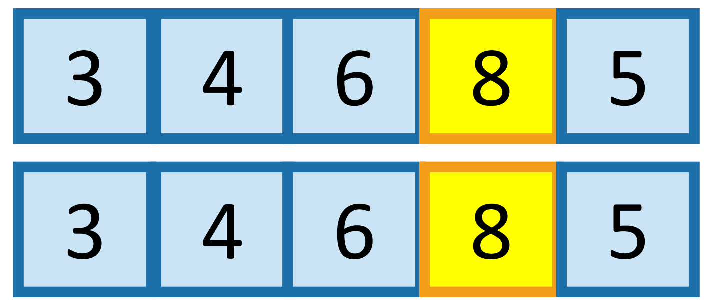
Start by moving A[1] toward the beginning of the list until you find something smaller (or can't go any further):



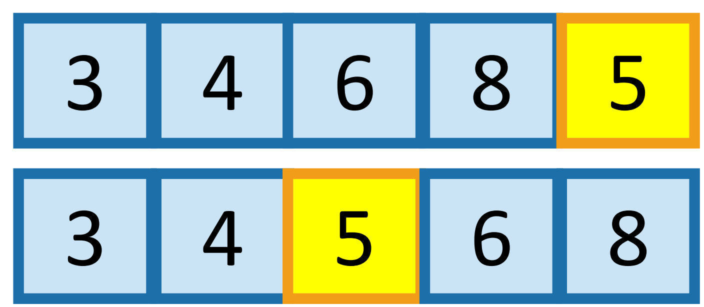
Then move A[2]:



Then move A[3]:



Then move A[4]:



Then we are done!

Insertion Sort

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

Insertion Sort

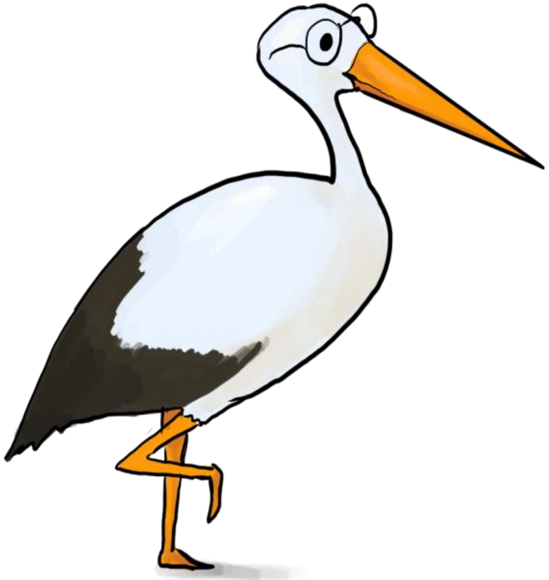
1. Does it work?
2. Is it fast?

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!

Insertion Sort: running time

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

Insertion Sort: running time

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

n-1 iterations of the outer loop

Insertion Sort: running time

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

n-1 iterations of the outer loop

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

Insertion Sort: running time

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

n-1 iterations of the outer loop

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

Running time is $O(n^2)$

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.

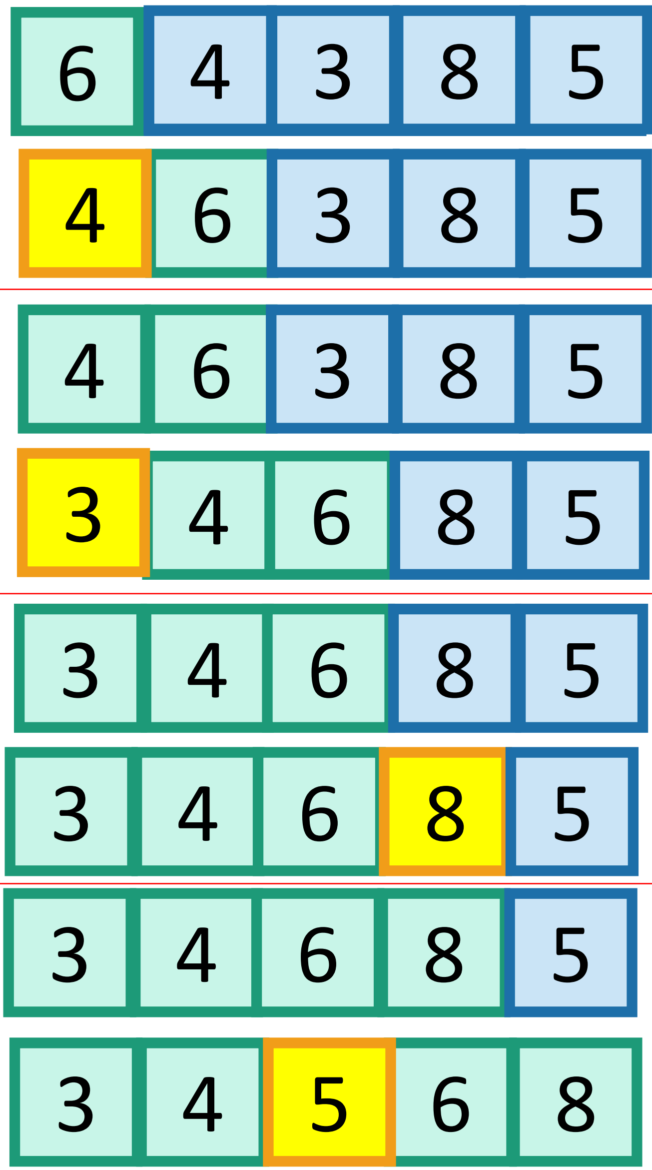
Why does this work?

- 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.



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!

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 i^{th} iteration.
  - Base case: the loop invariant holds before the 1^{st} iteration.
  - Inductive step: If the loop invariant holds after the i^{th} iteration, then it holds after the $(i+1)^{st}$ iteration
  - Conclusion: If the loop invariant holds after the last iteration, then we win.

Formally: induction

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

- Loop invariant(i): A[0:i] is sorted.

- Inductive Hypothesis:
  - The loop invariant(i) holds at the end of the i^{th} iteration (of the outer loop).
- Base case (i=0):
  - Before the algorithm starts, A[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! ✓

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.

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 $A[k] \geq A[j]$ for all $j \leq k$, and so A[i] is larger than everything that is positioned before it.
- Similarly, by the choice of *k* we have $A[i] \leq A[k + 1] \leq A[j]$ for all $j \geq k + 1$, 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

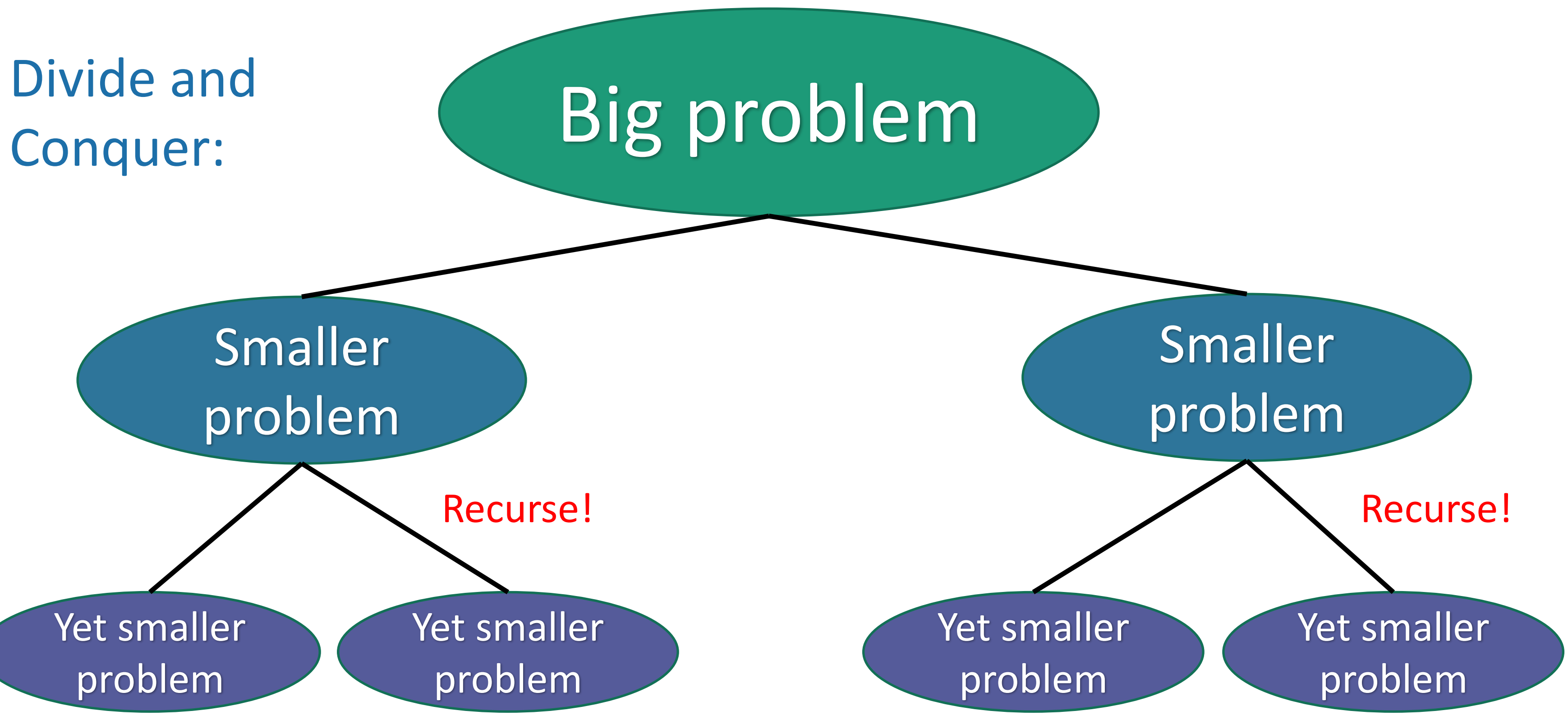
- Sorting Algorithms
  - InsertionSort: does it work and is it fast?
  - MergeSort: does it work and is it fast?

  - Skills:
    - Analyzing correctness of iterative and recursive algorithms.
    - Analyzing running time of recursive algorithms

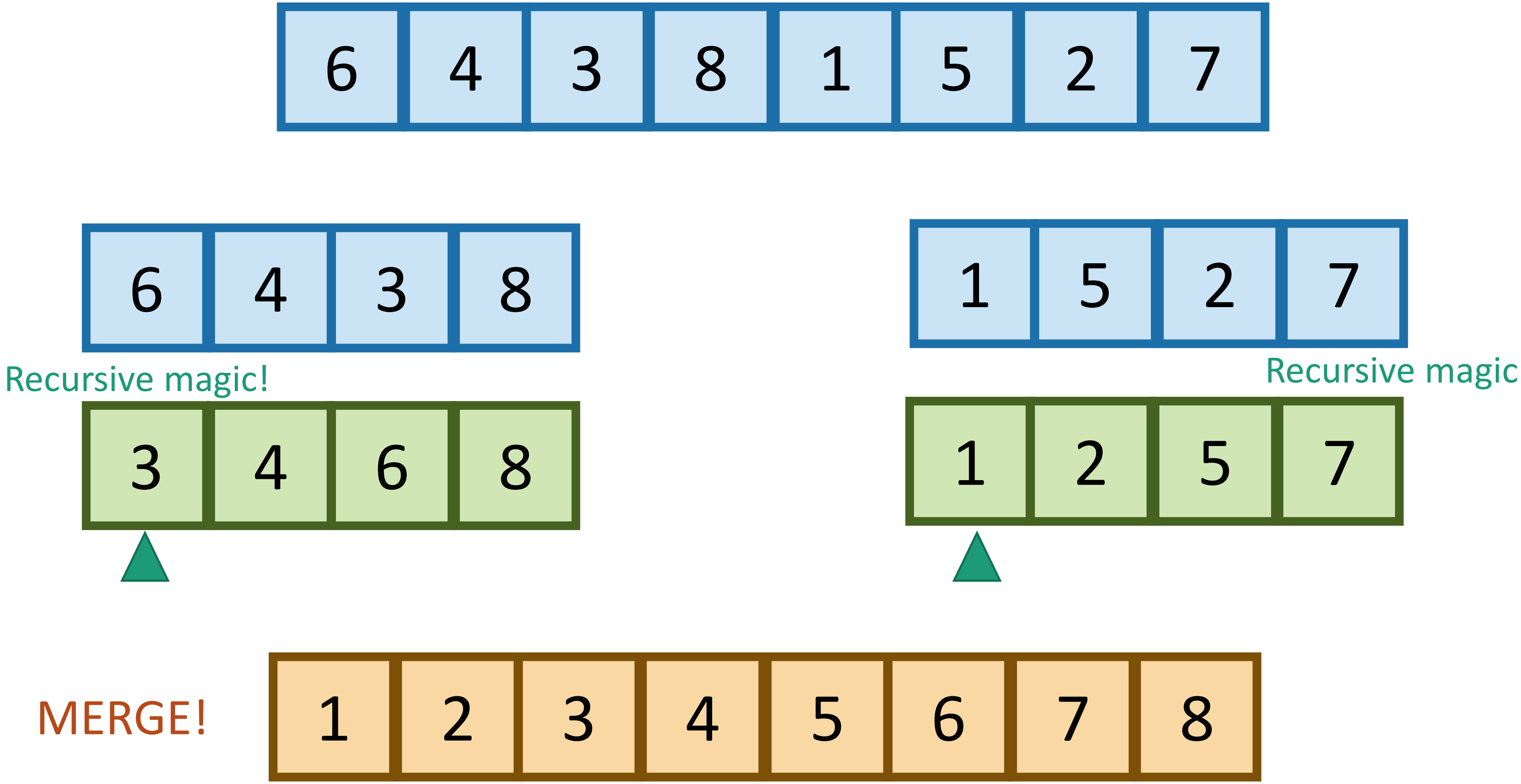
- How do we measure the runtime of an algorithm?
  - Worst-case analysis
  - Asymptotic Analysis

Can we do better?

- MergeSort: a divide-and-conquer approach



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

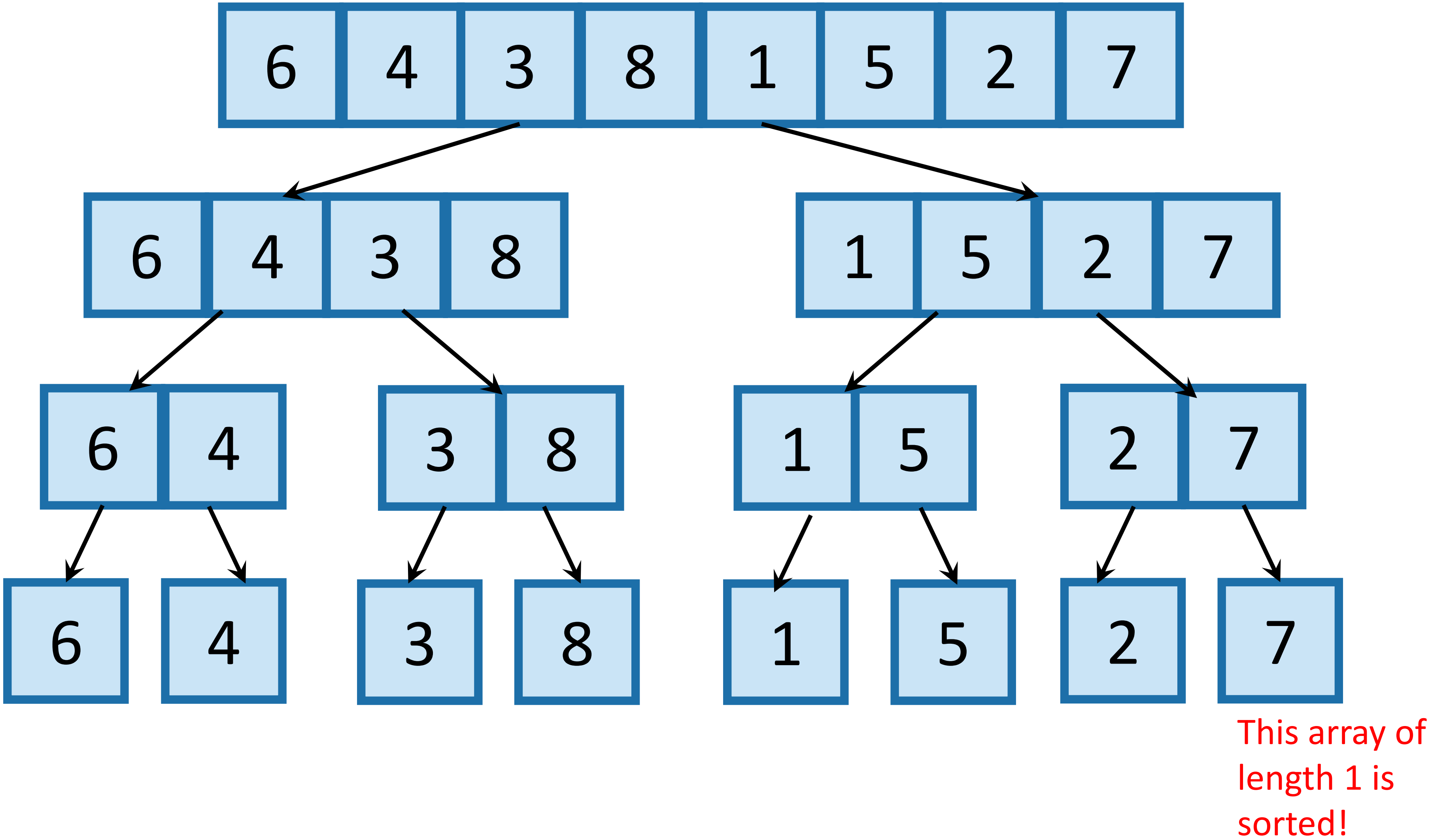
MERGESORT(A):

- n = length(A)
- **if** n ≤ 1:
  - **return** A

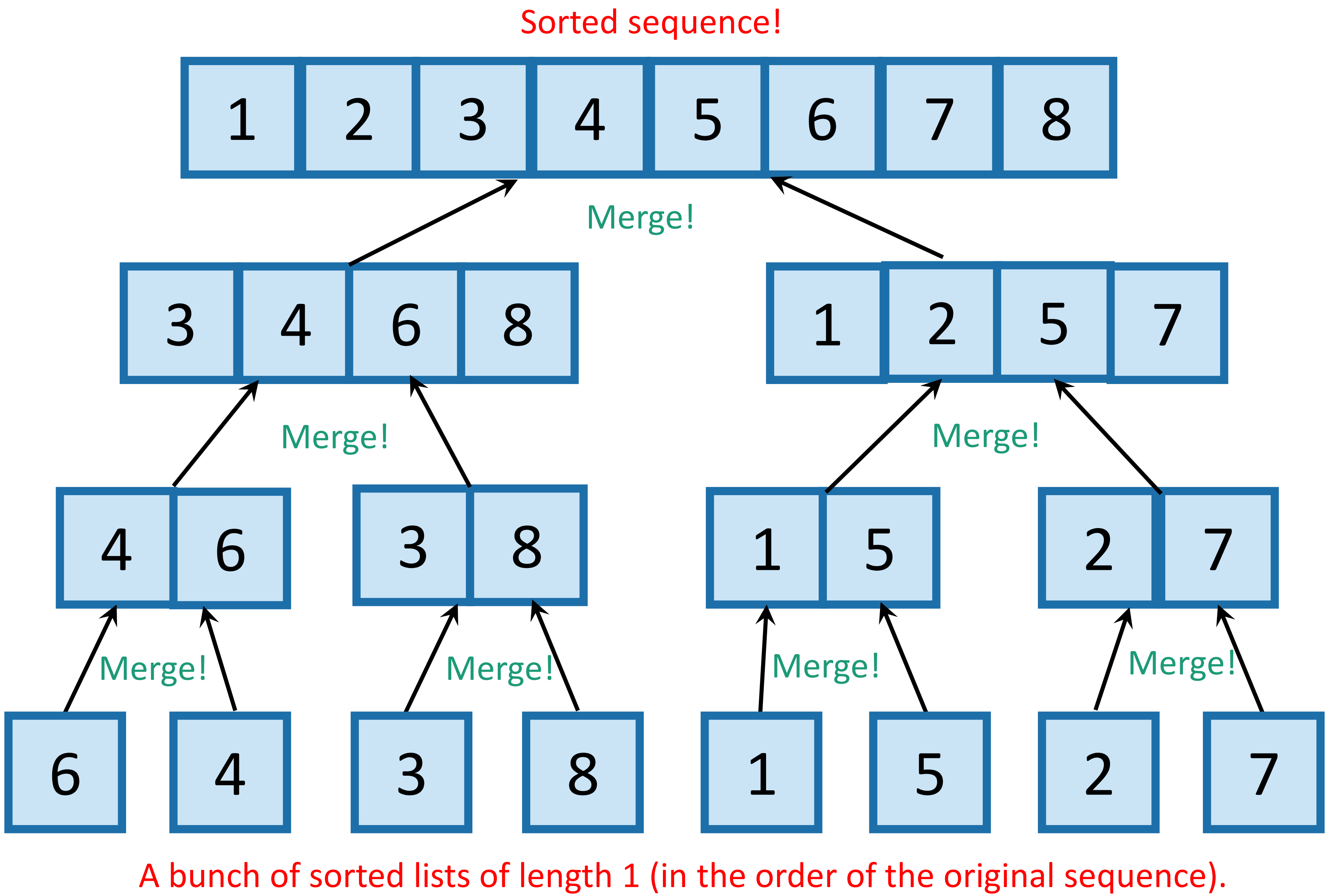
  If A has length 1,
  It is already sorted!
- L = MERGESORT(A[0 : (n/2)-1]) Sort the left half
- R = MERGESORT(A[n/2 : n-1]) Sort the right half
- **return** MERGE(L,R) Merge the two halves

What actually happens?

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



Then, merge them all back up!



Two questions

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

Empirically:

1. Seems to work.
2. Seems fast.

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)
```

Assume that n is a power of 2 for convenience.

It's fast

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.

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

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

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

- log(n) grows very slowly!

Assume that n is a power of 2 for convenience.

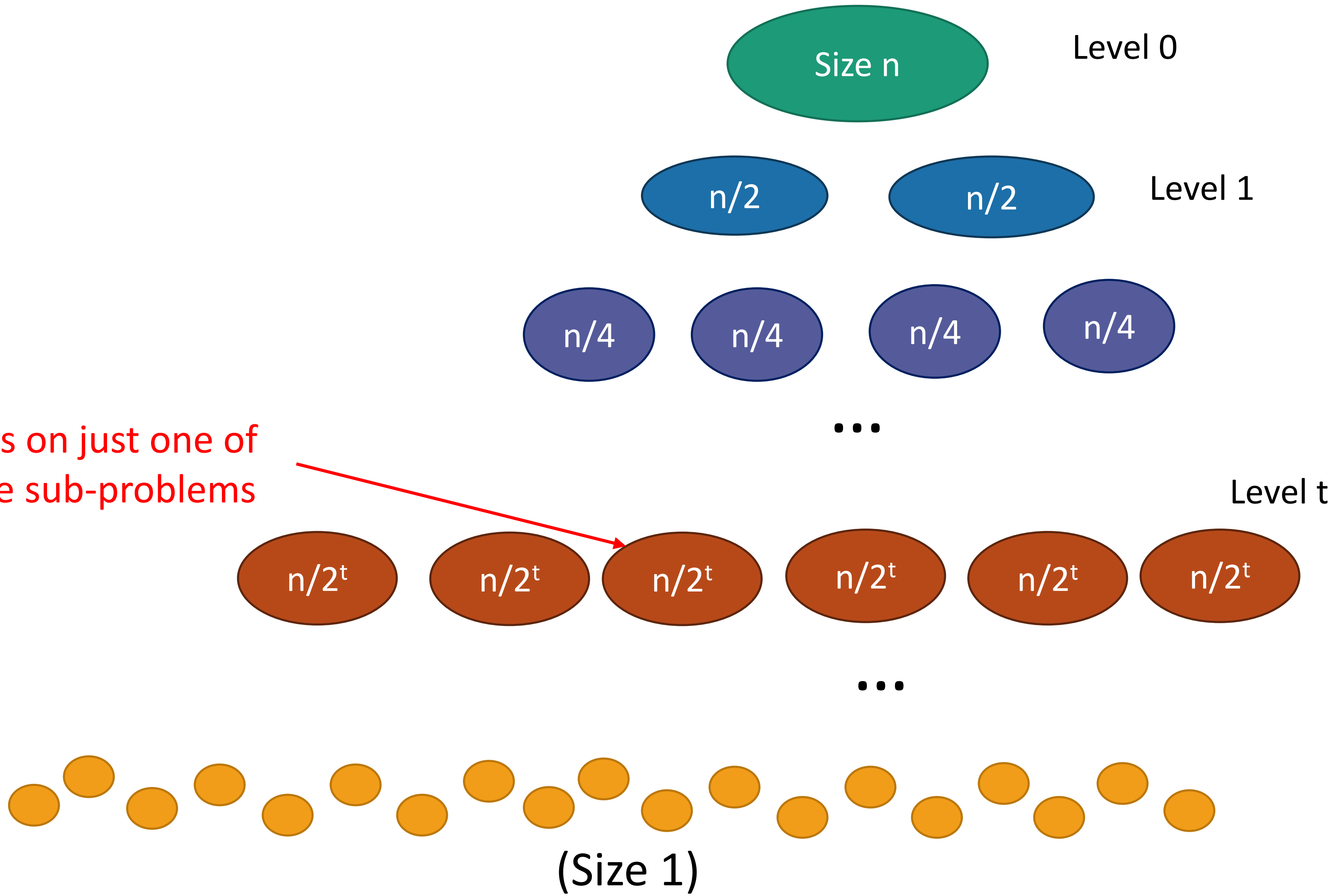
Now let's prove the claim

CLAIM:

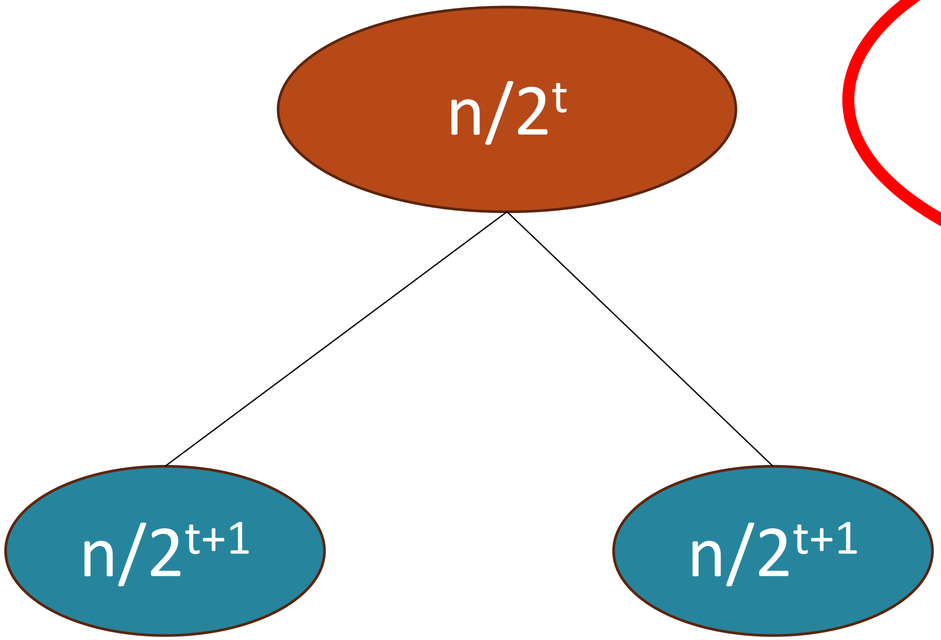
MergeSort requires at most $c*n\,(\log(n) + 1)$ operations to sort n numbers.

Let’s prove the claim

Focus on just one of these sub-problems



How much work in this sub-problem?



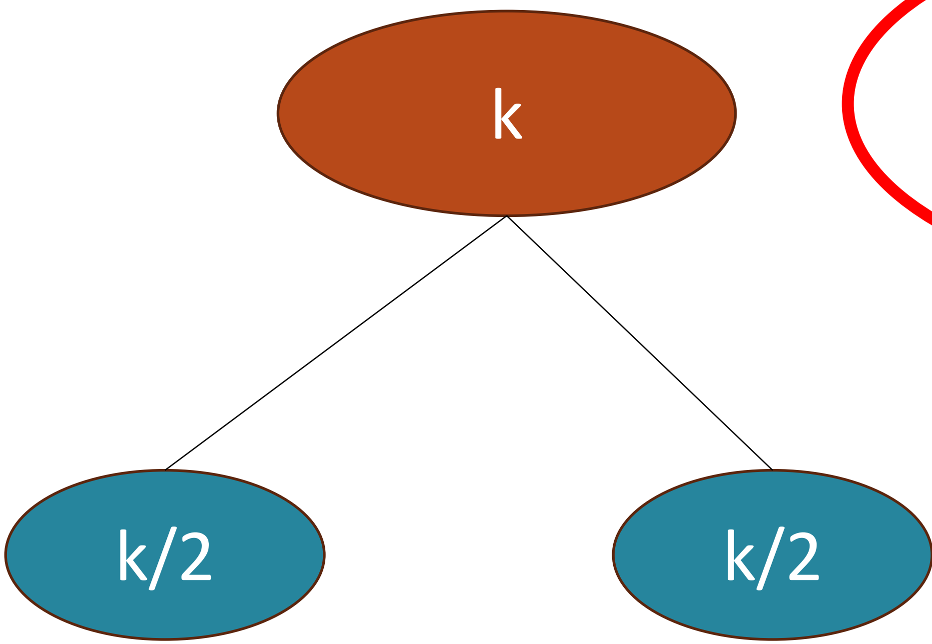
Time spent MERGE-ing the two subproblems

+

Time spent within the two sub-problems

How much work in this sub-problem?

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

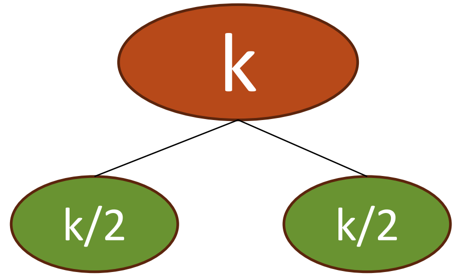


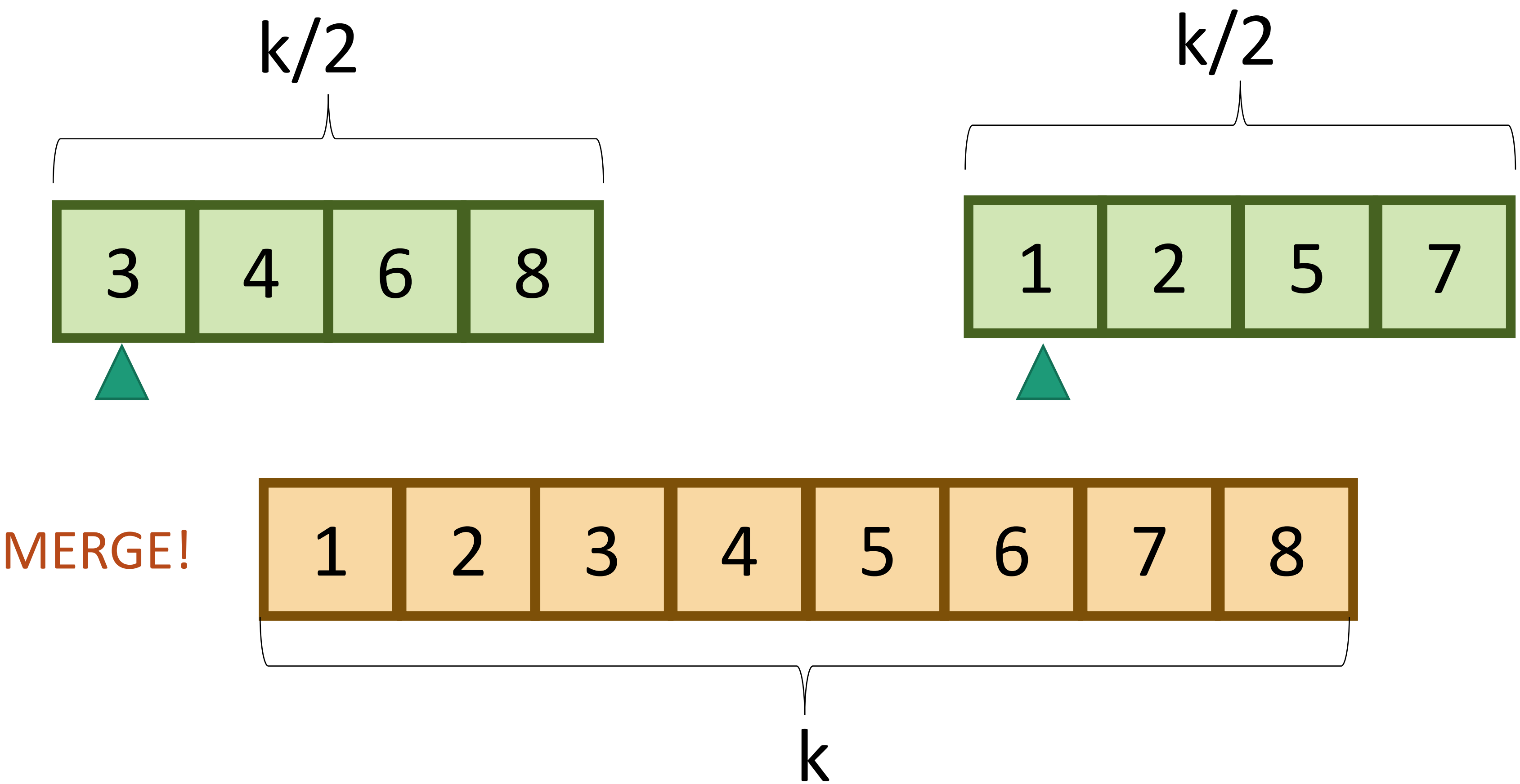
Time spent MERGE-ing the two subproblems

+

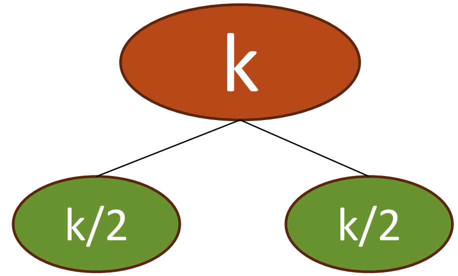
Time spent within the two sub-problems

How long does it take to MERGE?

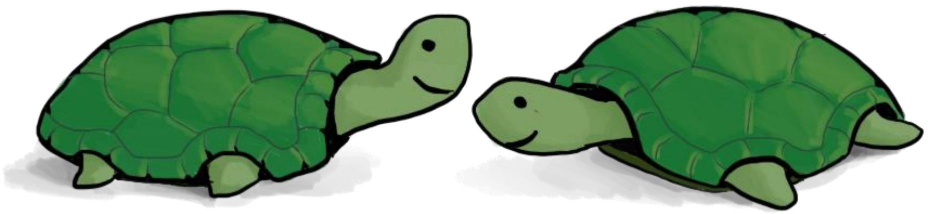




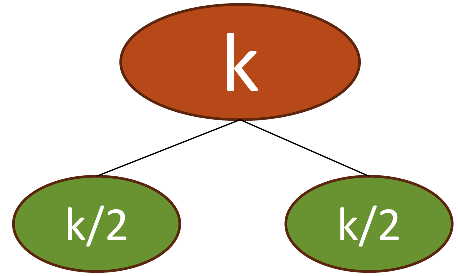
How long does it take to MERGE?



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

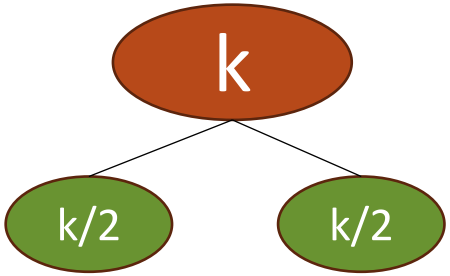


How long does it take to MERGE?



- 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…

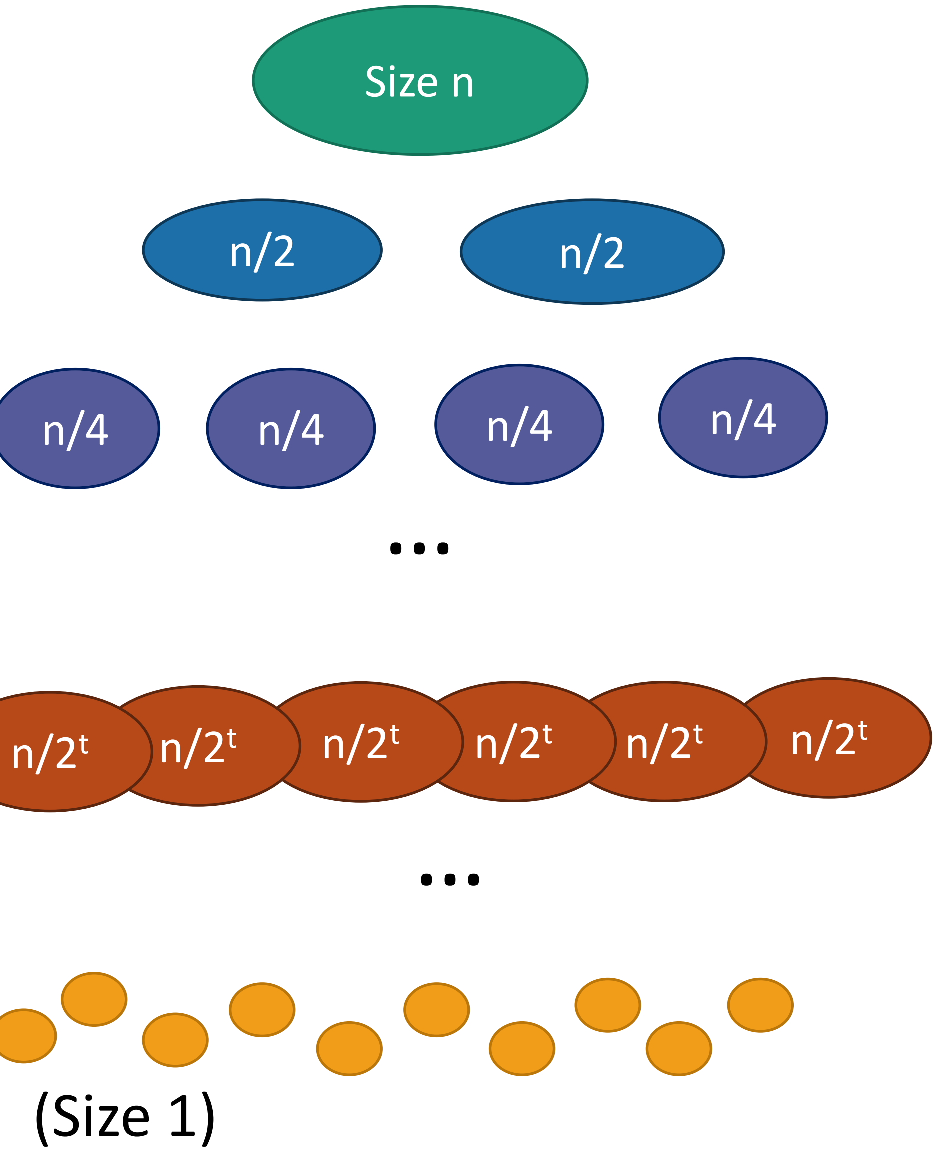
How long does it take to MERGE?

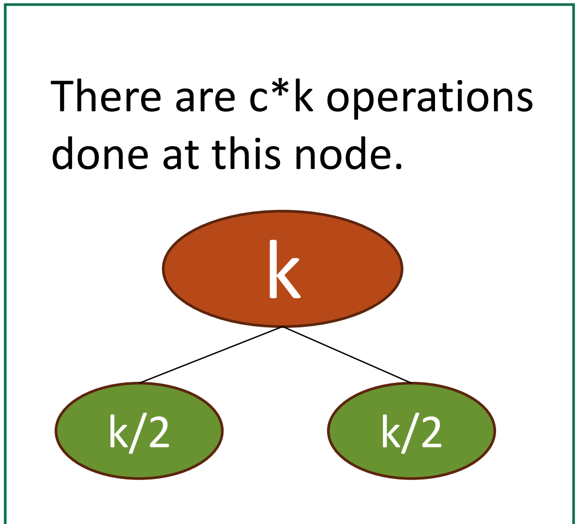


- Time to initialize an array of size k
- Plus the time to initialize three counters
- Plus the time to increment two of those counters k/2 times each
- Plus the time to compare two values at least k times
- Plus the time to copy k values from the existing array to the big array.
- Plus…

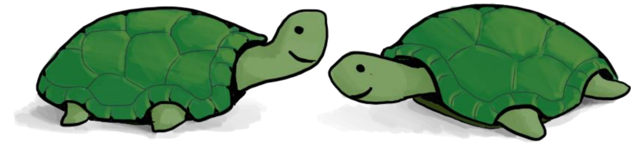
Let's say no more than c*k operations.

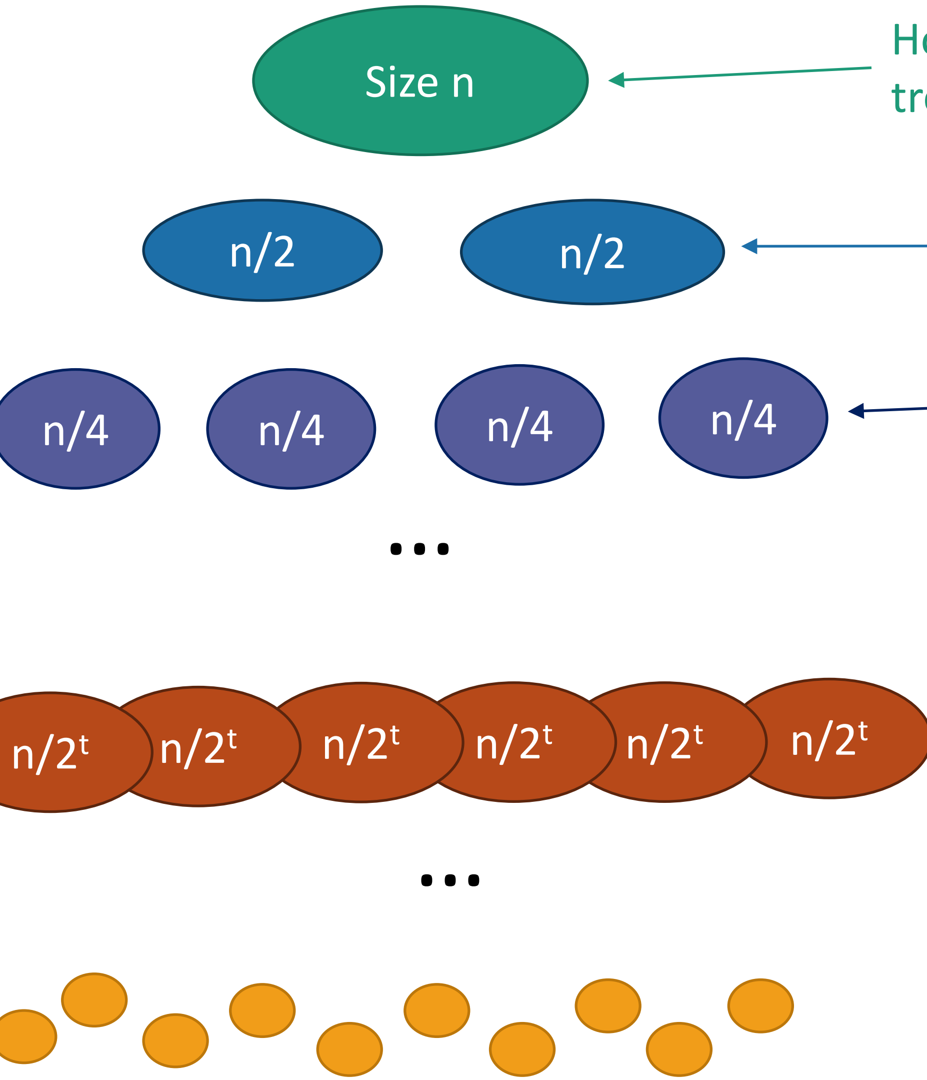
Recursion tree





Recursion tree





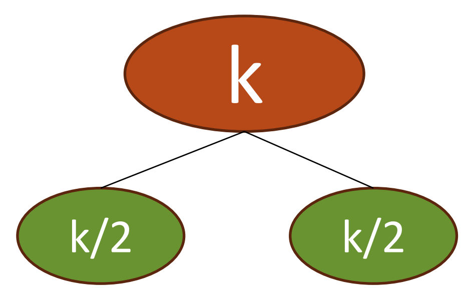
How many operations are done at this level of the tree? (Just MERGE-ing subproblems).

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

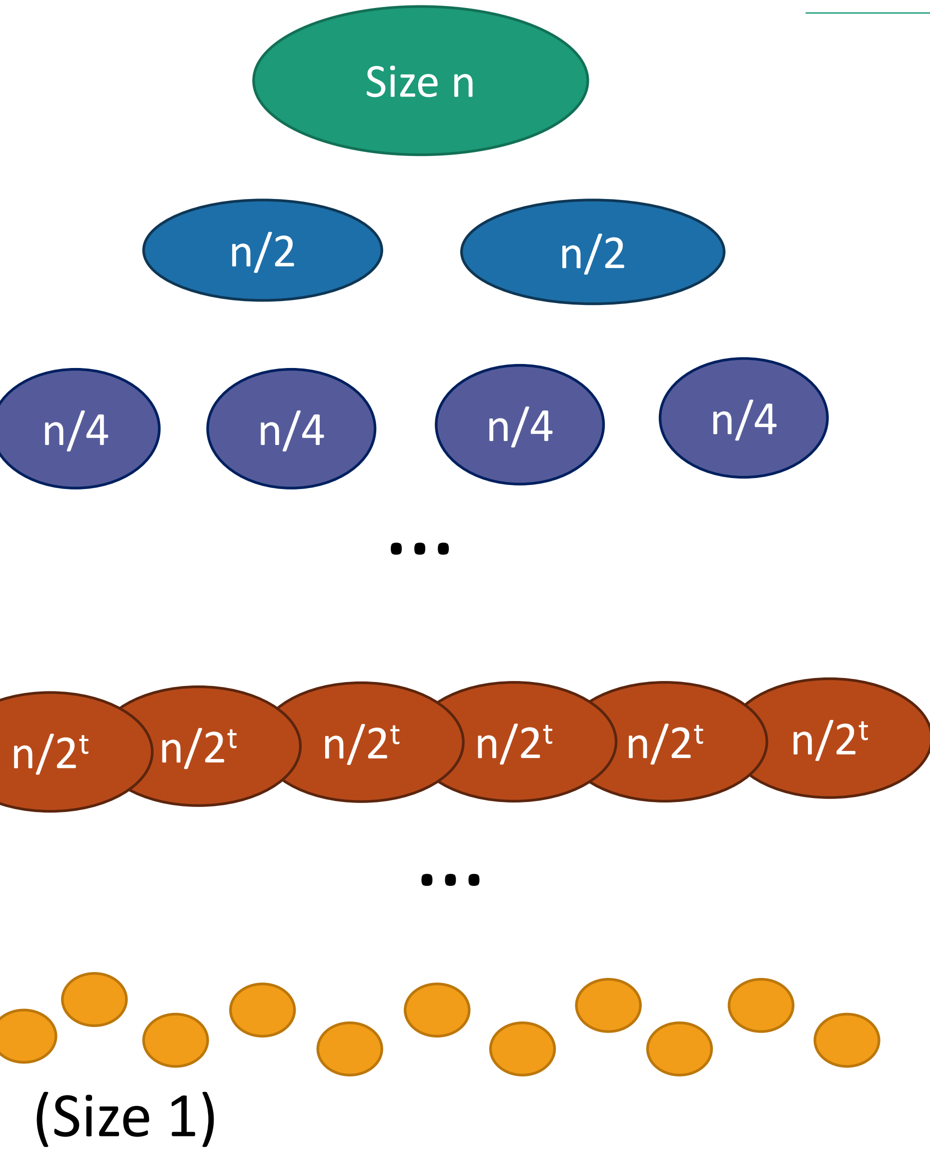
This level?

This level?

There are c*k operations done at this node.



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	$2*n \cong c*n$

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