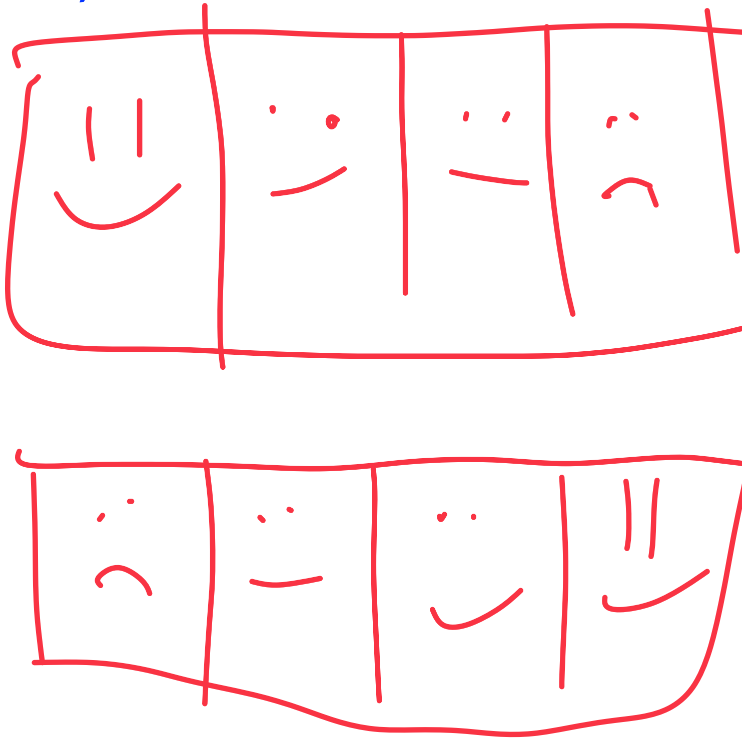
Algorithms and Data Structures for Data Science
Sorting

CS 277

Brad Solomon

April 15, 2024



Department of Computer Science

Exam Information

Exam 3 (4/23 — 4/25)

Covering all material up to last Wednesday (April 10th)

↳ All Trees
↳ Graphs
↳ Hash tables

↳ MP_ Mosaic
Binary Search
↳ Trees

Final Exam (05/02 — 05/06)

Must take

Expected time: 1 hour exam in 1 hour, 50 minute time block

3 makeup exams
Take one of them

50 minute makeup exams *during* final exam time!

Submit topics or concepts you want reviewed

Google form linked through Prairielearn

↳ Concepts to review

↳ Topics not seen that you want to see

We’ve seen most core data structures

Lists

Graphs

Trees

Hash Tables

But we haven’t seen a great deal of **algorithms!**

For the rest of the class, review core concepts…

And apply them to new problems!

Learning Objectives

Review fundamentals of lists and Big O

Introduce common sorting algorithms

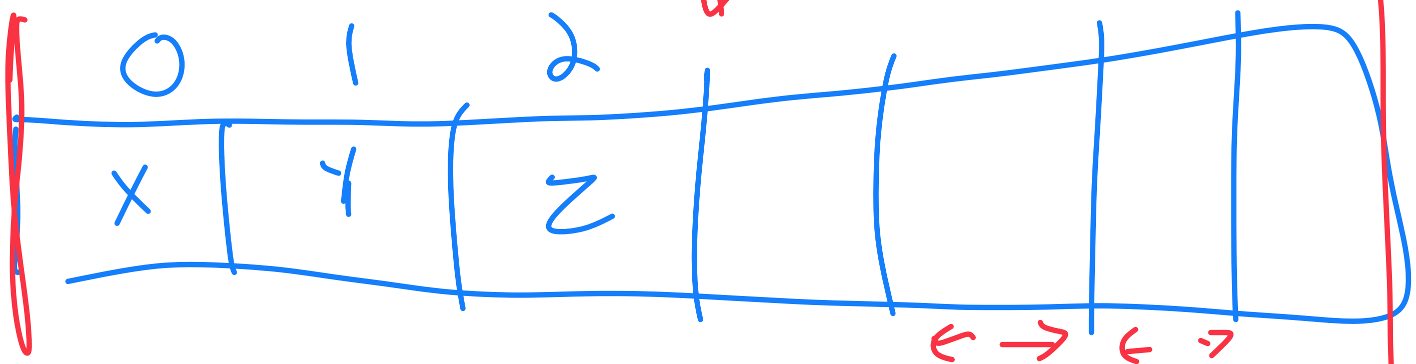
Practice analyzing the Big O of different methods

List Implementations

Array List

of items in list

Specific size

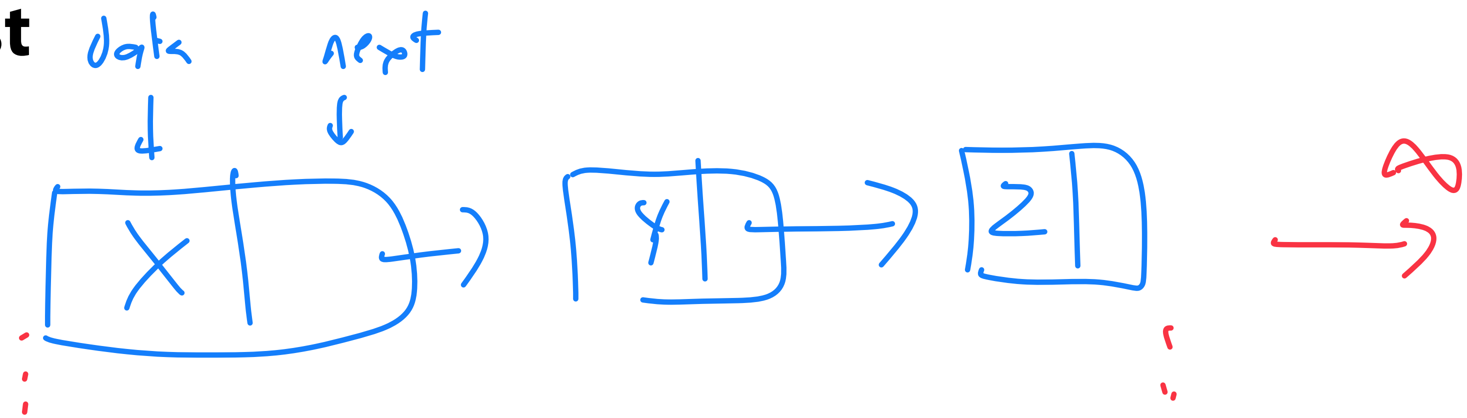


equal size

Set of data in an ordered context

Memory is continuous

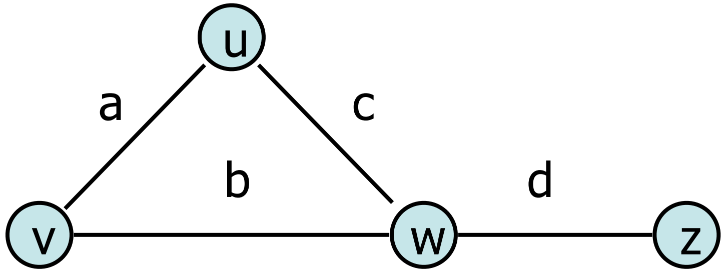
Linked List



∞ growth allowed

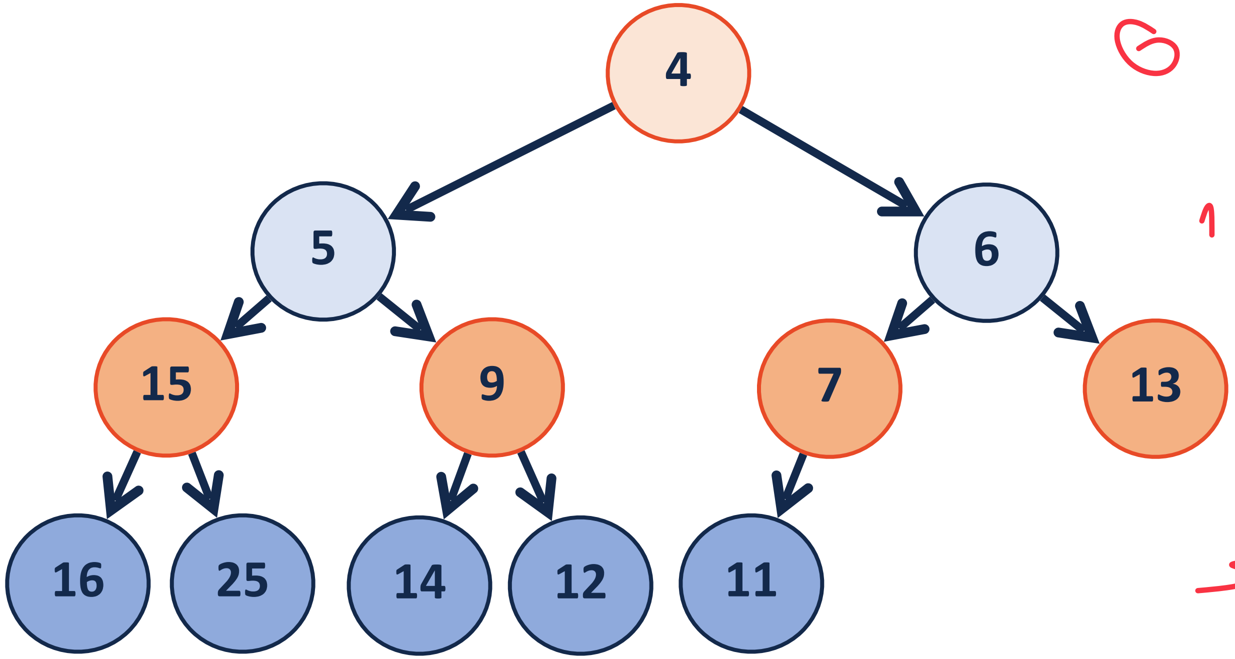
Individual objects

Lists are a great way to store **data structures**



u
v
w
z

u	v	a
v	w	b
u	w	c
w	z	d



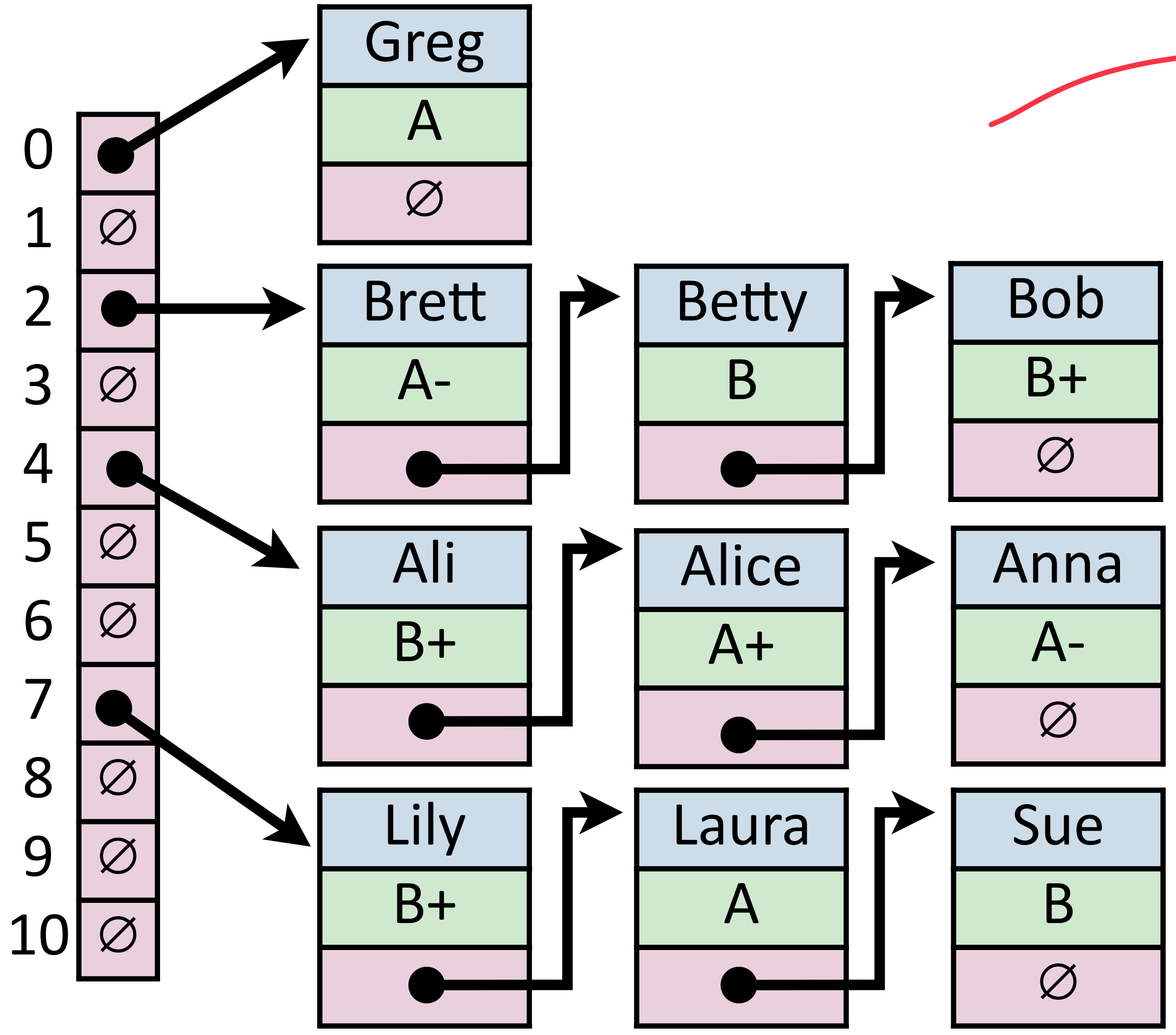
0
1
2
3

Depth

0 1 2 3

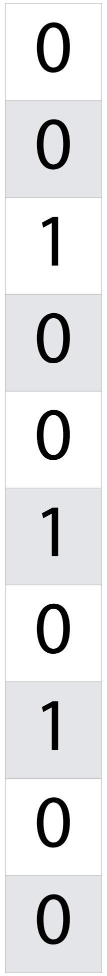
	4	5	6	15	9	7	20	16	25	14	12	11			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Lists are a great way to store **data structures**

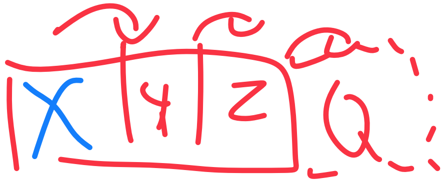


$$H = \{h_1, h_2, \ldots, h_k\}$$

Bloom filter



Array Implementation



	Array	Singly Linked List
Look up given an input **index**	$O(1)$	$O(n)$
Search given an input **value**	$O(n)$	$O(n)$
Insert/Remove at **front**	$O(n)$ Expected $O(1)^*$	$O(1)$
Insert/Remove at **arbitrary** location	$O(n)$ Assume array doesnt need to be resized	$O(n)$

The Sorting Problem

Given a collection of objects, C, with comparable values, order the objects such that $\forall x \in C, x_i \leq x_{i+1}$

Input:	8	4	3	1	2	5	6	9	0	7
Output:	0	1	2	3	4	5	6	7	8	9

SelectionSort

4	3	6	7	1
1	3	6	7	4
1	3	6	7	4
1	3	4	7	6
1	3	4	6	7

for list size n

$O(n)$

1. Find the i-th smallest value

2. Place it at position i via swap $O(1)$

3. Repeat for $0 \leq i \leq n-1$ (n) times

Big O: $n \times [O(n) + O(1)] = O(n^2)$

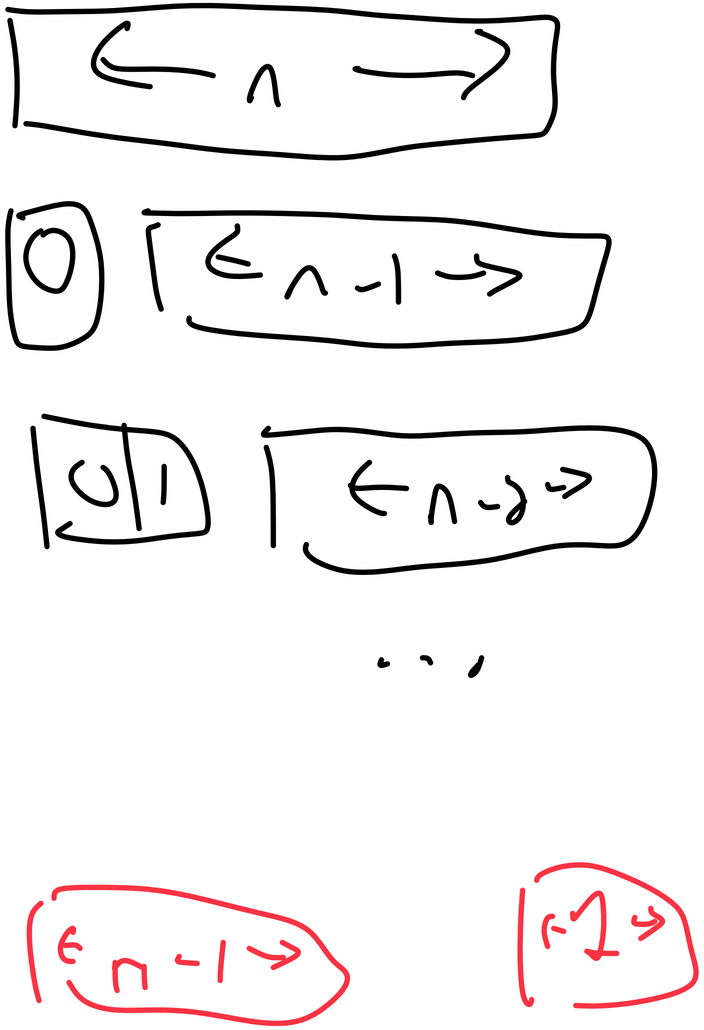
Expand swap:

tmp = L[i] $O(1)$

L[i] = L[0] $O(1)$

L[0] = tmp $O(1)$

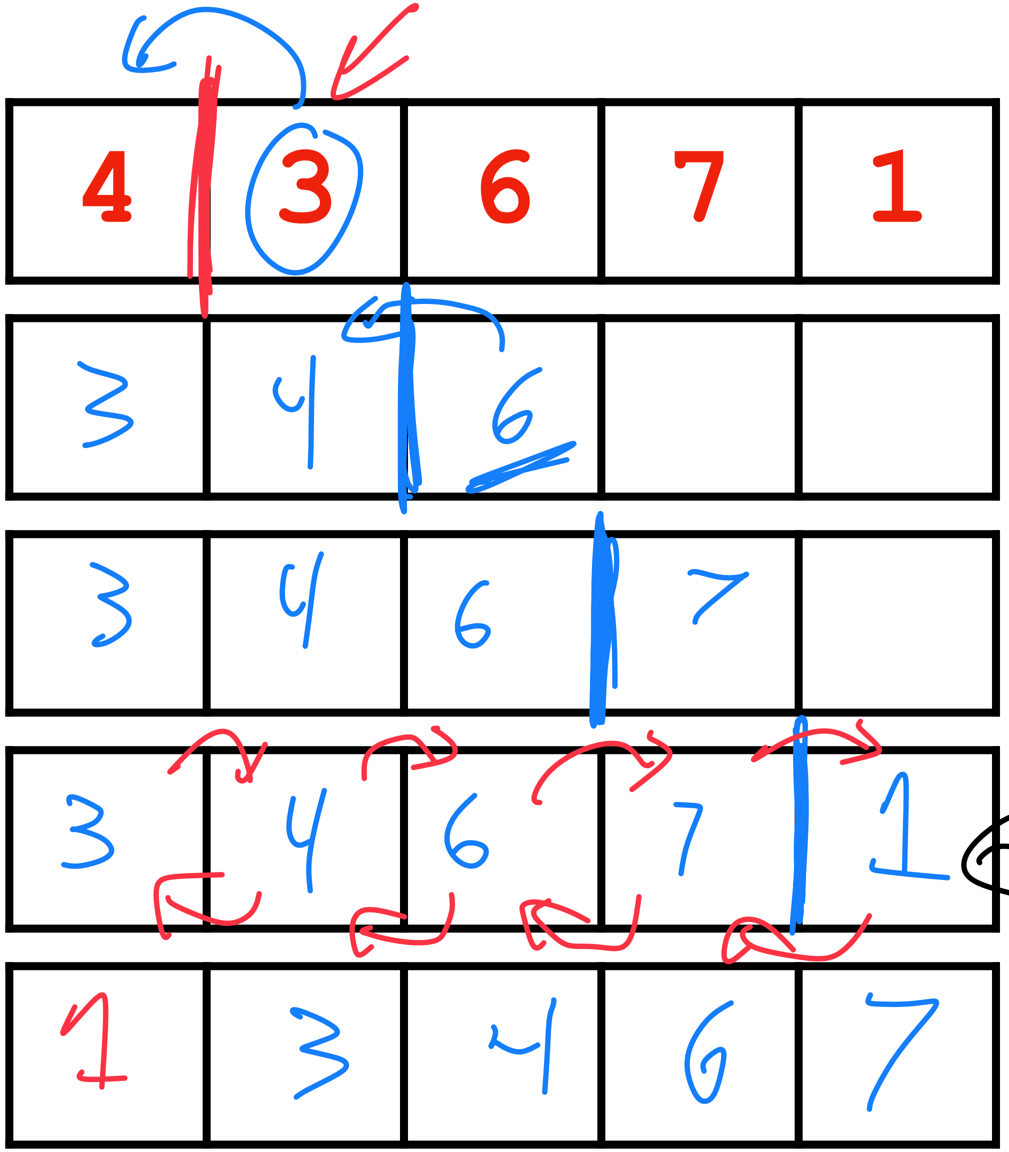
SelectionSort Efficiency **(large n)**



$$n + (n-1) + (n-2) + \dots + 1$$

$$= \frac{n(n-1)}{2} \approx O(n^2)$$

InsertionSort



$O(1)$

1. Assume first value is 'sorted'

$(n-1) \times \sim n \times$

2. Loop through remaining values:

3. Insert value into the 'sorted' array

Key trick: Insert by swapping!

worst case performance

$O(n \times (n+1)) = O(n^2)$

$O(n)$

$O(1) \times n$ swaps

InsertionSort Efficiency

(large n)

Worst case swap #
1
2
3

$$1 + 2 + 3 + \dots + (n-1)$$

Best and Worst Case insertionSort

$O(n^2)$

Given the numbers 0 — 6, what is the **best** possible insertionSort input?

0	1	2	3	4	5	6

0 swaps, 0, 0, 0, 0, 0

can run in $O(n)$ time in the best case

Given the numbers 0 — 6, what is the **worst** possible insertionSort input?

6	5	4	3	2	1	0

Recursive Array Sorting

0	3	7	5	8	9	2	1	4	6

Base Case: Array w/ one item is sorted

Recursive Step: Split list in half, sort both halves

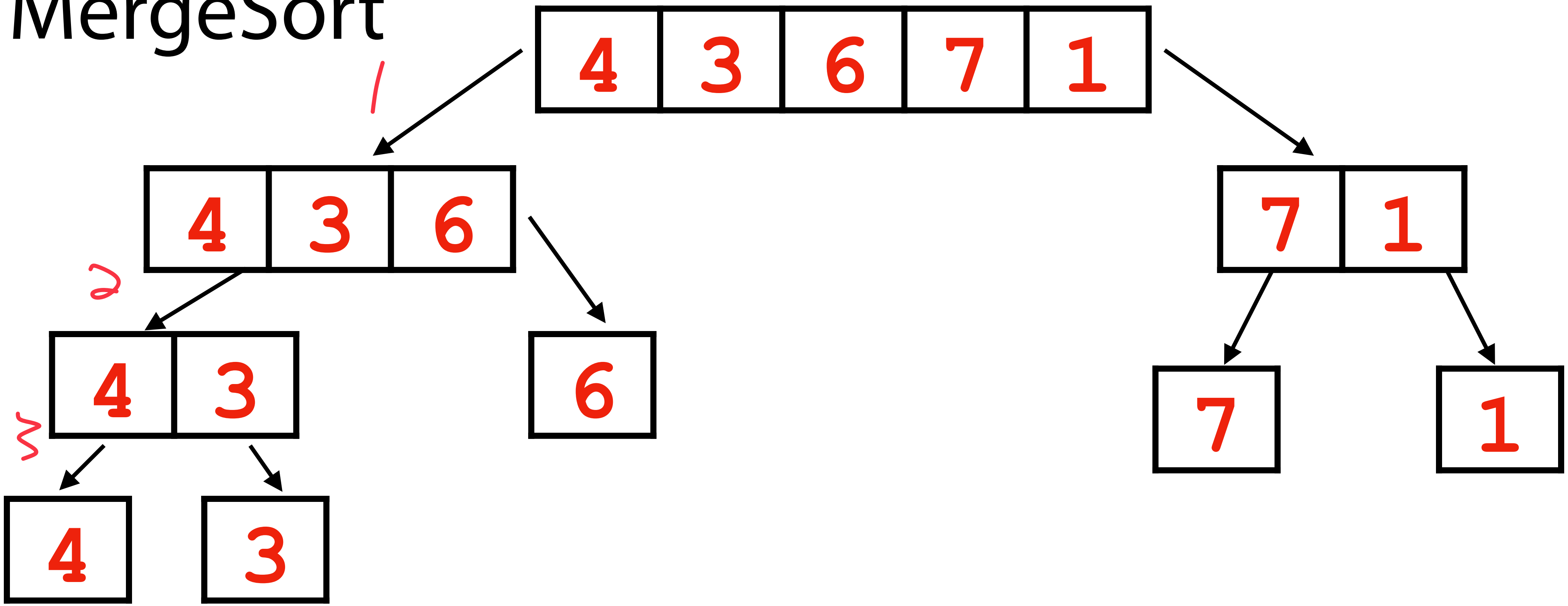
Combining: Merge sorted lists

MergeSort

4	3	6	7	1

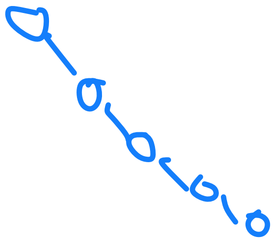
MergeSort

n = 5



height of R's tree?

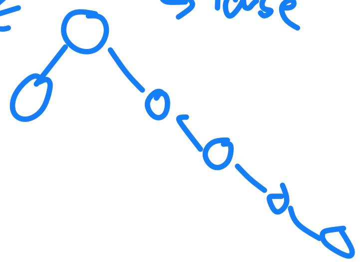
Binary tree

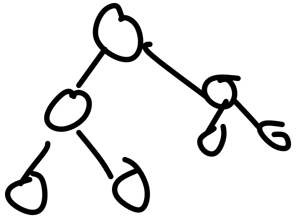


O(n)

Binary search tree

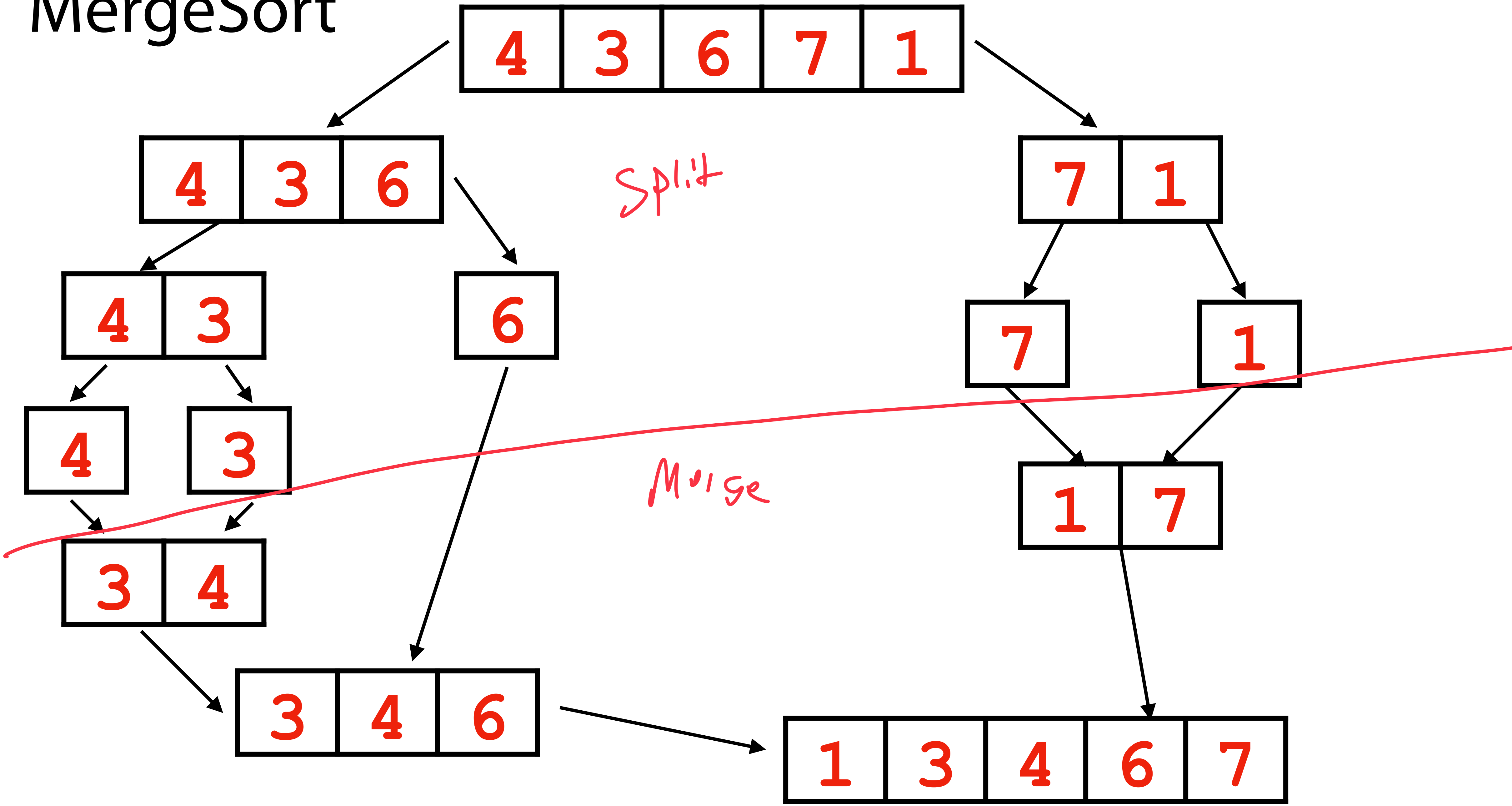
small ← → large





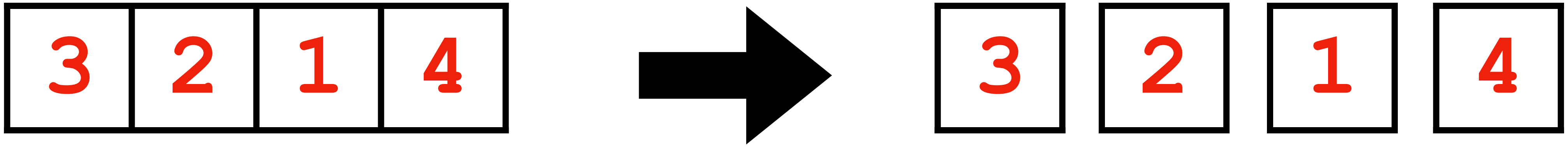
O(log n)

MergeSort

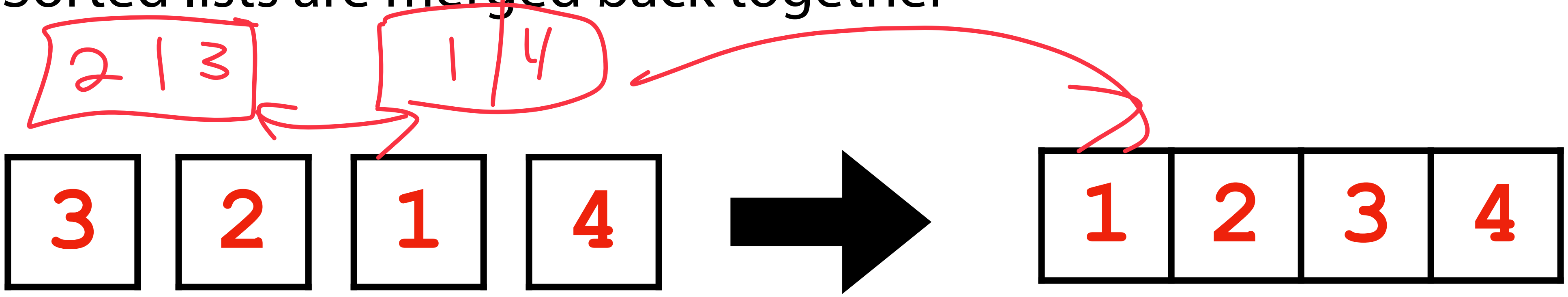


Recursive MergeSort

1) Input list recursively split to a collection of “sorted” base cases



2) Sorted lists are merged back together



Recursive MergeSort Efficiency (large n)

1) Input list split in half O(1)

3	2	1	4

→

3	2

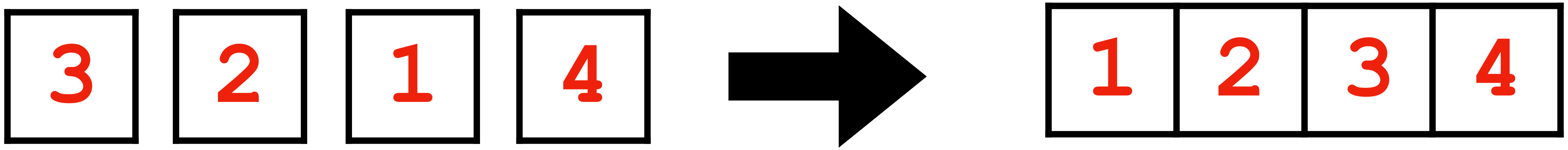
1	4

How many times do we have to split a list in half?

↳ Height of recursion tree is log(n)

Recursive MergeSort Efficiency (large n)

2) Sorted lists are merged back together



Recursive MergeSort Efficiency (large n)

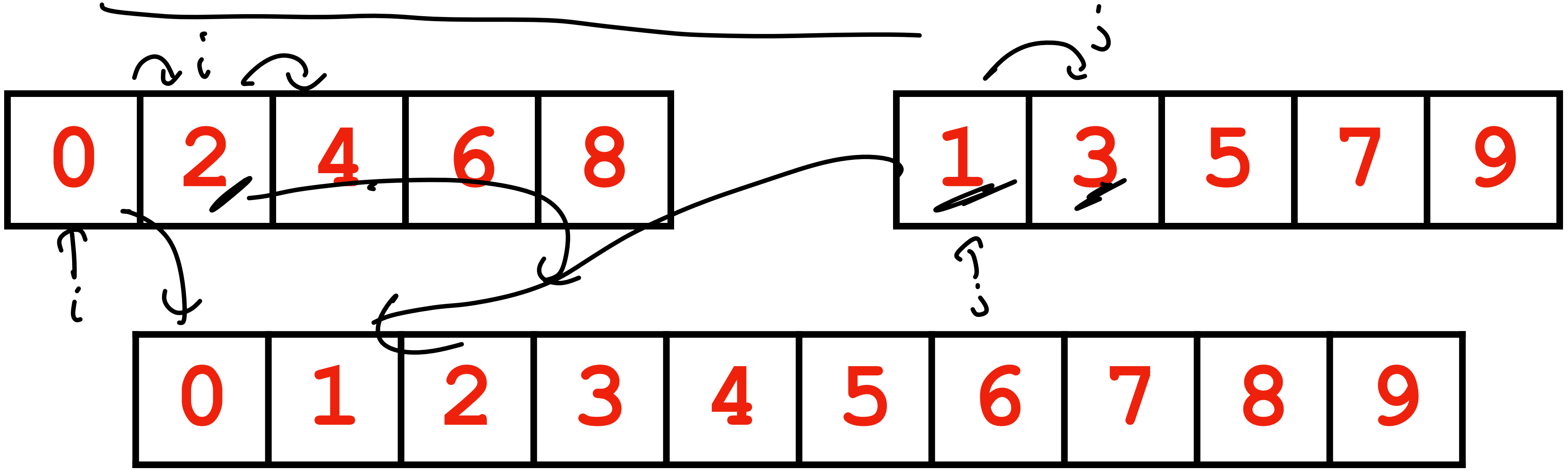
2) Sorted lists are merged back together

3	2	1	4

→

1	2	3	4

Claim: *Merging two sorted arrays can be done in* $O(n+m)$ *time*



Look at every item once

Recursive MergeSort Efficiency (large n)

$$T(n) = \underbrace{2\ T(n/2)}_{\text{Split list in half (recurse on halves)}} + \underbrace{C * n}_{\text{Merging}}$$

$(n/2 + n/2)$

$\rightarrow O(n \log n)$

Best and Worst Case mergeSort

Given the numbers 0 — 6, what is the **best** possible mergeSort input?

0	1	2	3	4	5	6

O(log n) chops

n merge

Given the numbers 0 — 6, what is the **worst** possible mergeSort input?

6	5	4	3	2	1	0

O(log n) chops

n merge

Optimal Sorting

Claim: Any deterministic comparison-based sorting algorithm must perform $O(n \log n)$ comparisons to sort n objects.

0	1	2	1	0	2	2	0	1
0	2	1	1	2	0	2	1	0

Sorting Algorithm Tradeoffs

	Best Case Time	Worst Case time	Best Case Space	Worst Case Space
SelectionSort	$O(n^2)$	$O(n^2)$	$O(1)$	$O(1)$
InsertionSort	$\boxed{O(n)}$	$O(n^2)$	$O(1)$	$O(1)$
MergeSort	$O(n \log n)$	$\boxed{O(n \log n)}$	$O(n)$	$O(n)$

fixed work

Sorting Algorithm Tradeoffs

	Best Case Time	Worst Case time	Best Case Space	Worst Case Space
SelectionSort	$O(n^2)$	$O(n^2)$	$O(1)$	$O(1)$
InsertionSort	$O(n)$	$O(n^2)$	$O(1)$	$O(1)$
MergeSort	$O(n \log n)$	$O(n \log n)$	$O(n)$	$O(n)$

Bonus Content: TimSort (Python's built-in sort)

An *adaptive* sort — adjusts behavior based on input data

Take advantage of *runs* of consecutive ordered elements

Start by using insertionSort to build sorted lists of <= 64 elements

$O(n^2)$ for $n=64$ isn't bad

Use MergeSort once all sub-arrays are ordered

Additional heuristics speed up merging in practice

QuickSort

6	1	0	3	7	9	2	4

1. Choose a *pivot* value

QuickSort

6	1	0	3	7	9	2	4

1. Choose a *pivot* value

2. Divide the array into two partitions (larger and smaller than pivot)

QuickSort

Sorted

1	0	3	2	4	9	6	7

1. Choose a *pivot* value

2. Divide the array into two partitions (larger and smaller than pivot)

3. Recursively QuickSort partitions

QuickSort

1	0	3	2	4	9	6	7

1. Choose a *pivot* value

2. Divide the array into two partitions (larger and smaller than pivot)

3. Recursively QuickSort partitions

QuickSort

1	0	2	3	4	6	7	9

1. Choose a *pivot* value

2. Divide the array into two partitions (larger and smaller than pivot)

3. Recursively QuickSort partitions

QuickSort

1	0	2	3	4	6	7	9

1. Choose a *pivot* value

2. Divide the array into two partitions (larger and smaller than pivot)

3. Recursively QuickSort partitions

QuickSort

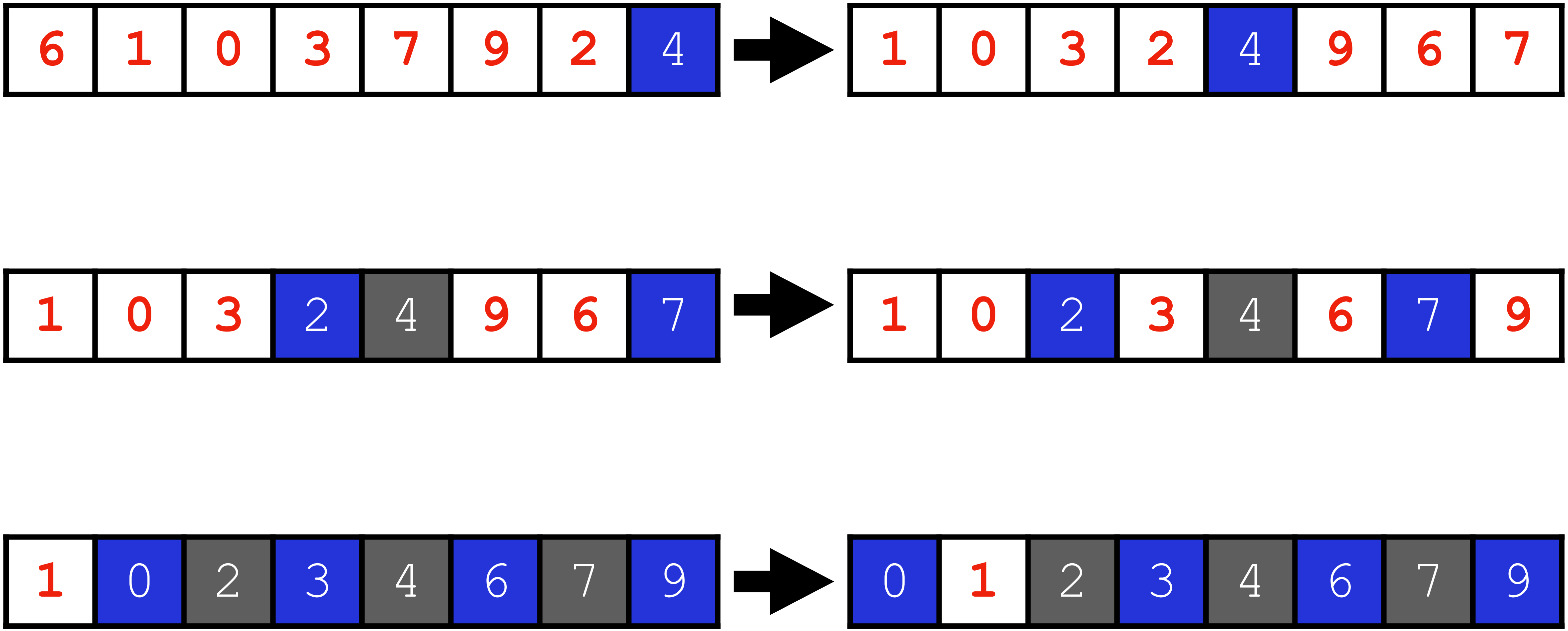
0	1	2	3	4	6	7	9

1. Choose a *pivot* value

2. Divide the array into two partitions (larger and smaller than pivot)

3. Recursively QuickSort partitions

QuickSort



Recursive Quicksort

0	3	7	5	8	9	2	1	4	6

Base Case: A list of size 1 is sorted

Recursive Step: Quicksort on both partitions around pivot

Combining: Nothing! (Put pivot in between partitions)