Algorithms and Data Structures for Data Science
Sorting

CS 277
Brad Solomon

April 15, 2024

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN
Department of Computer Science

Exam Information

Exam 3 (4/23 — 4/25)

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

Final Exam (05/02 — 05/06)

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

50 minute makeup exams *during* final exam time!

Submit topics or concepts you want reviewed

Google form linked through Prairielearn

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

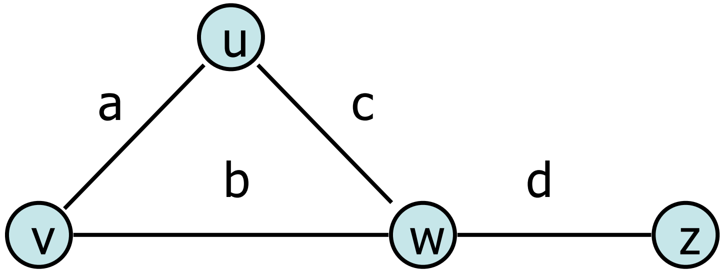
Practice analyzing the Big O of different methods

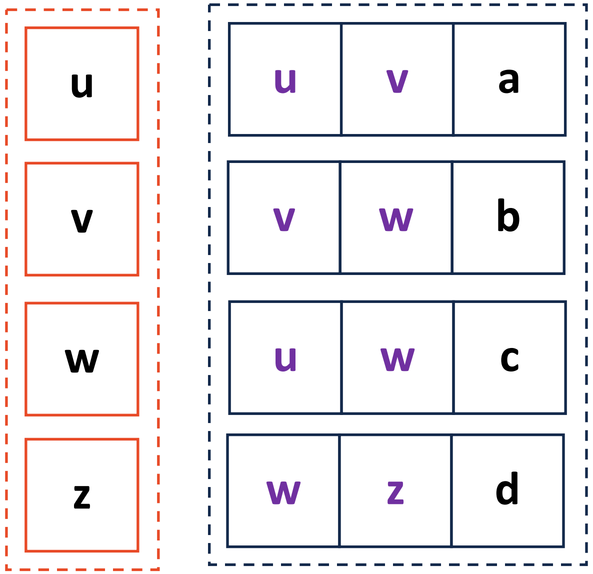
List Implementations

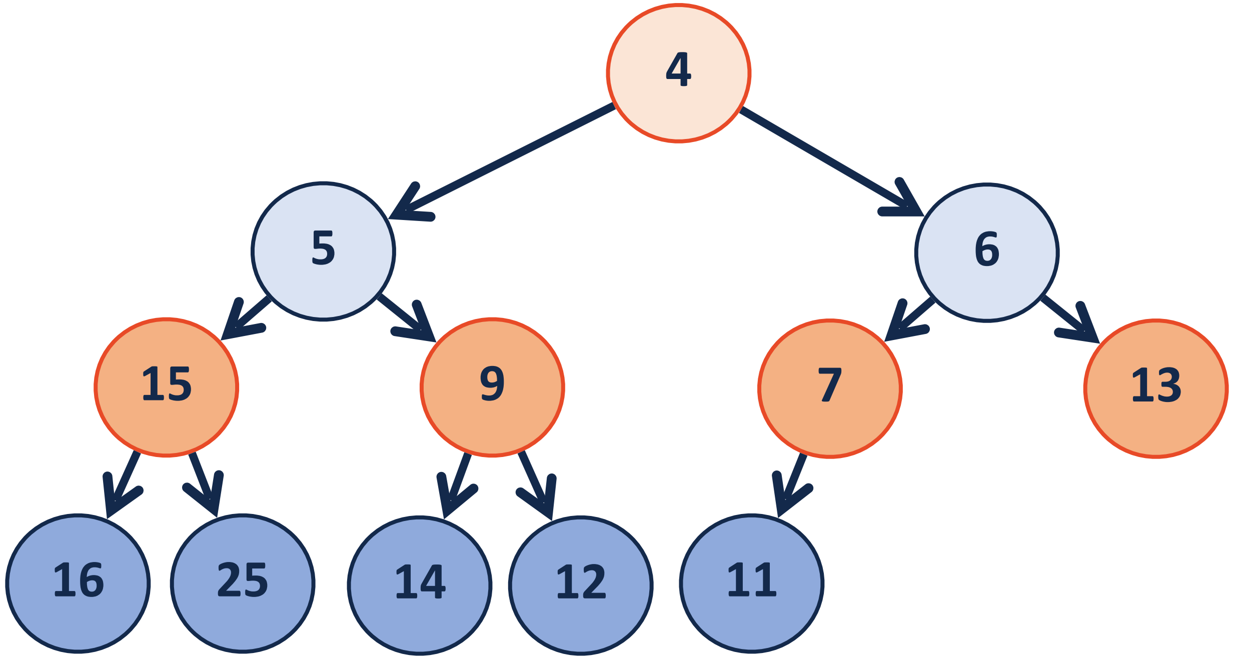
Array List

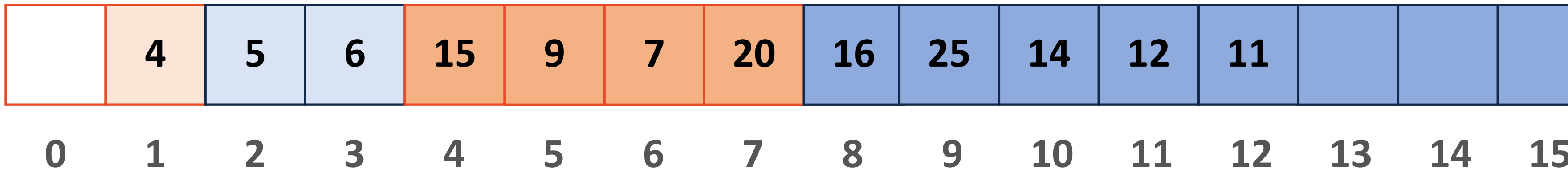
Linked List

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

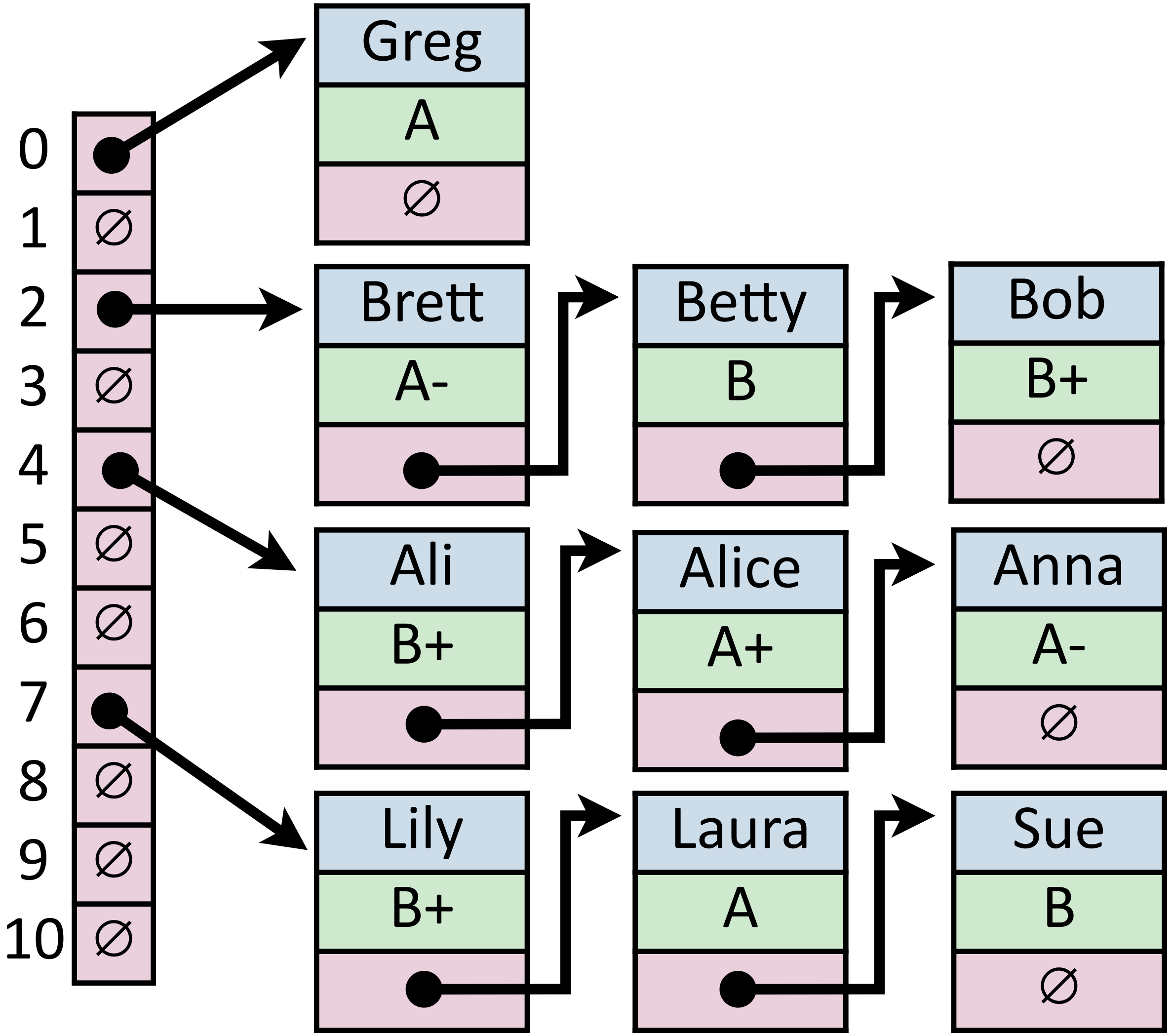








Lists are a great way to store **data structures**



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

0
0
1
0
0
1
0
1
0
0

Array Implementation

	Array	Singly Linked List
Look up given an input **position**		
Search given an input **value**		
Insert/Remove at **front**		
Insert/Remove at **arbitrary** location		

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. Find the i-th smallest value
2. Place it at position i via swap
3. Repeat for $0 \leq i \leq n - 1$

SelectionSort Efficiency `(large n)`

InsertionSort

4	3	6	7	1

1. Assume first value is 'sorted'

2. Loop through remaining values:

3.Insert value into the 'sorted' array

Key trick: Insert by swapping!

InsertionSort Efficiency (large n)

Best and Worst Case insertionSort

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

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

Recursive Array Sorting

0	3	7	5	8	9	2	1	4	6

Base Case:

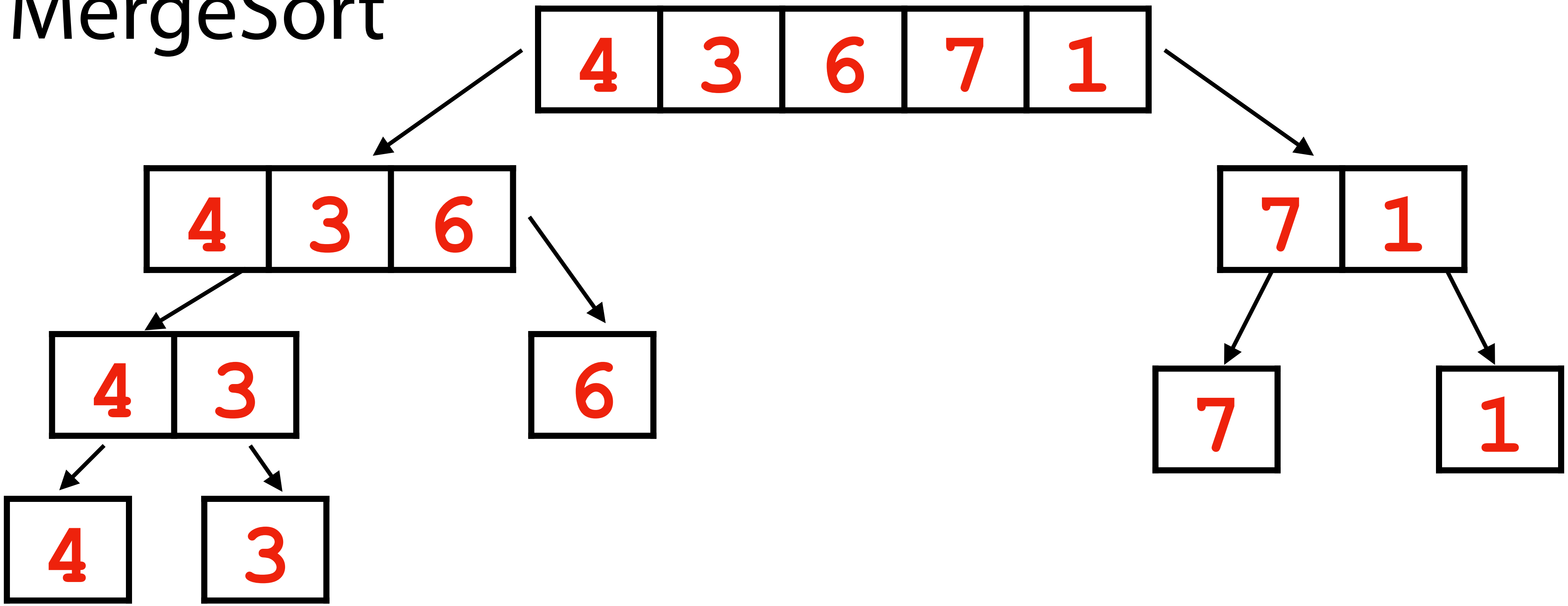
Recursive Step:

Combining:

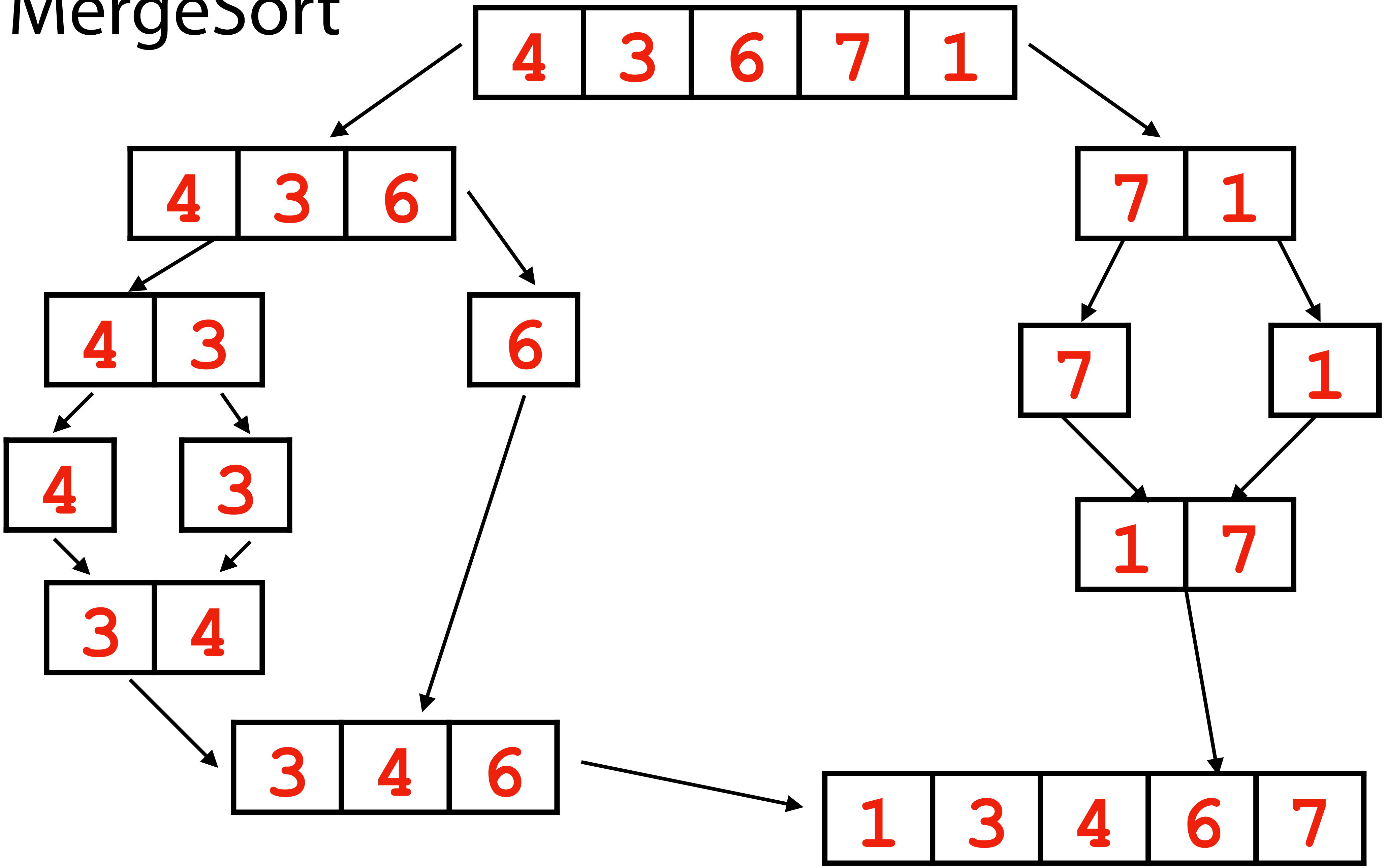
MergeSort

4	3	6	7	1

MergeSort

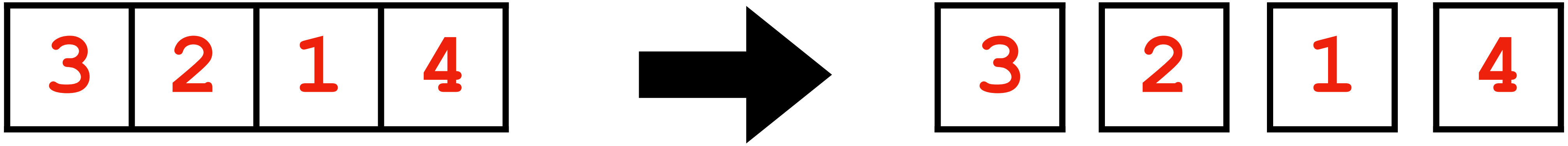


MergeSort

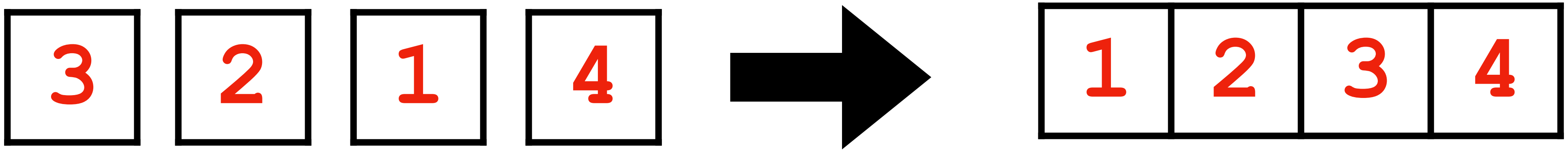


Recursive MergeSort

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

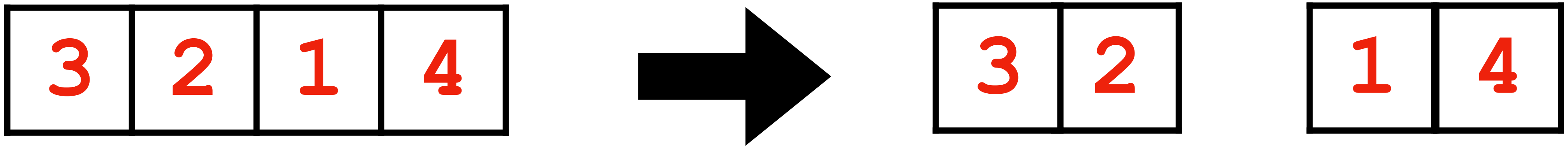


2) Sorted lists are merged back together



Recursive MergeSort Efficiency (large n)

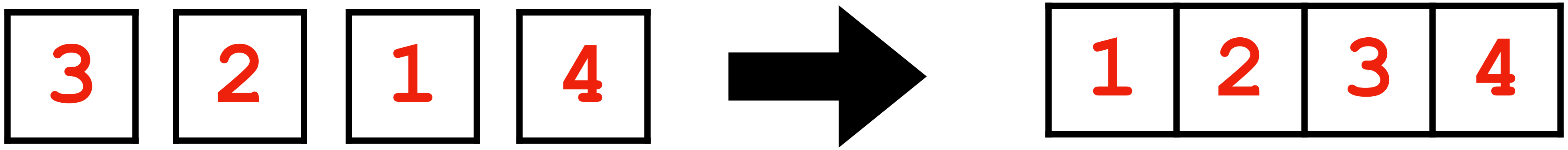
1) Input list split in half



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

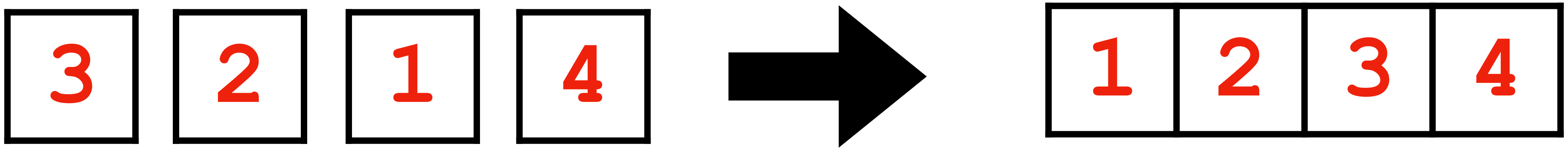
Recursive MergeSort Efficiency (large n)

2) Sorted lists are merged back together

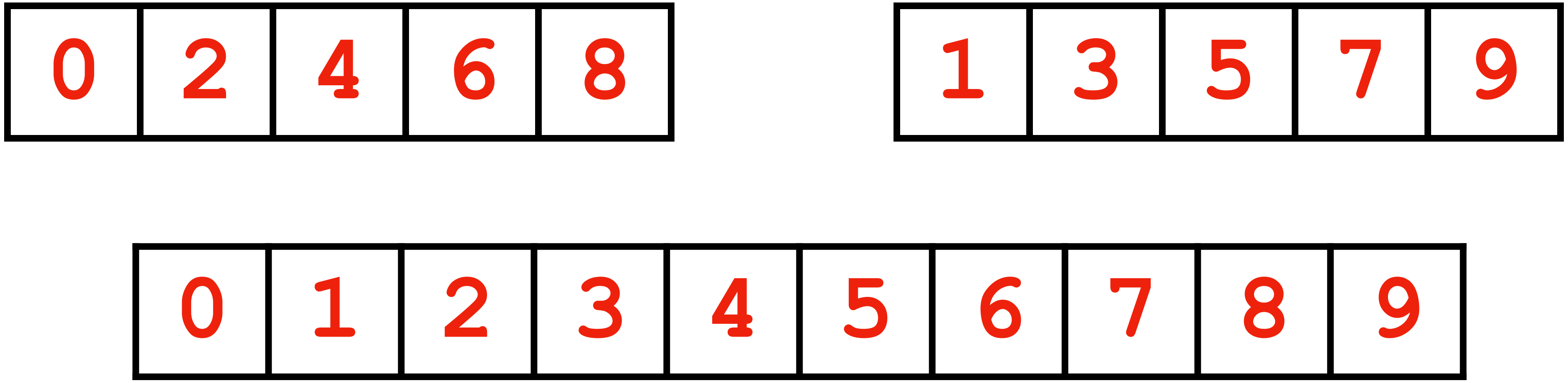


Recursive MergeSort Efficiency (large n)

2) Sorted lists are merged back together



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

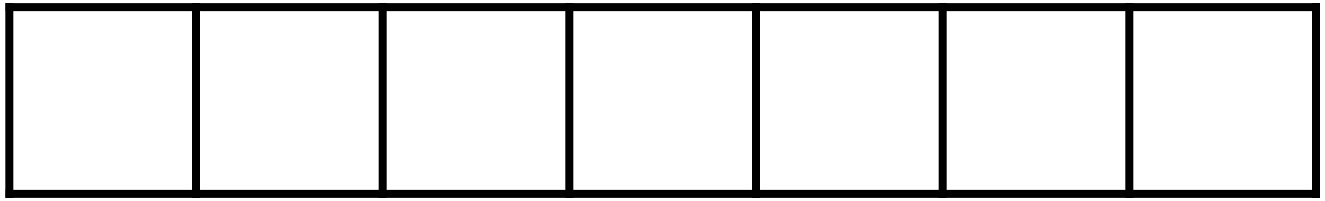


Recursive MergeSort Efficiency **(large n)**

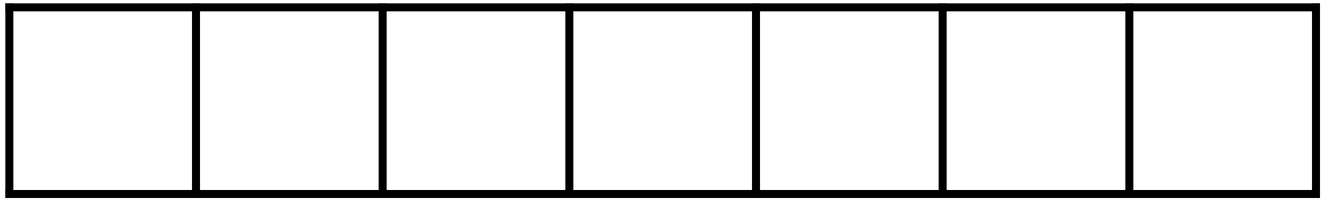
$$T(n) = 2\ T(n/2) + C * n$$

Best and Worst Case mergeSort

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



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



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				
InsertionSort				
MergeSort				

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

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

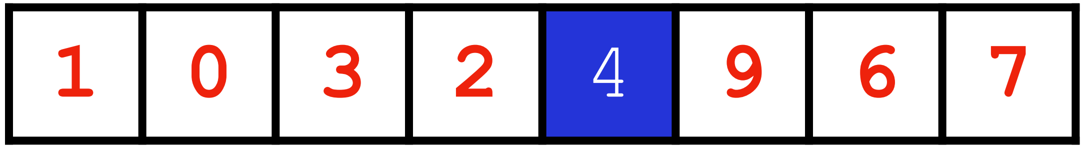
QuickSort

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



1. Choose a *pivot* value

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

3. Recursively QuickSort partitions

QuickSort

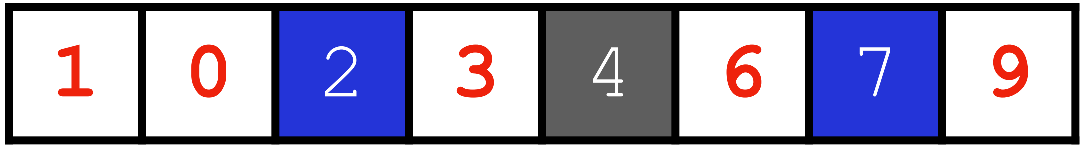
1	0	3	2	4	9	6	7

1. Choose a *pivot* value

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

3. Recursively QuickSort partitions

QuickSort

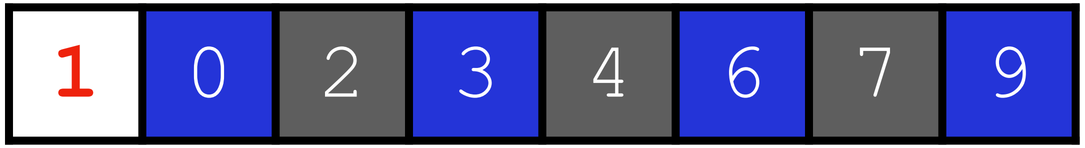


1. Choose a *pivot* value

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

3. Recursively QuickSort partitions

QuickSort

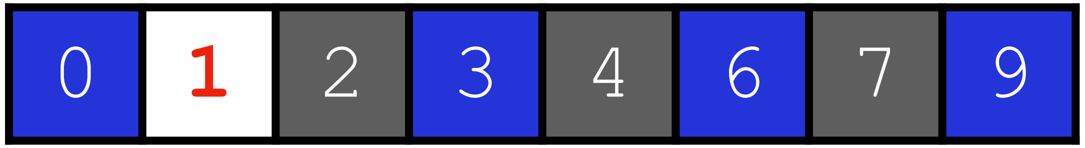


1. Choose a *pivot* value

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

3. Recursively QuickSort partitions

QuickSort

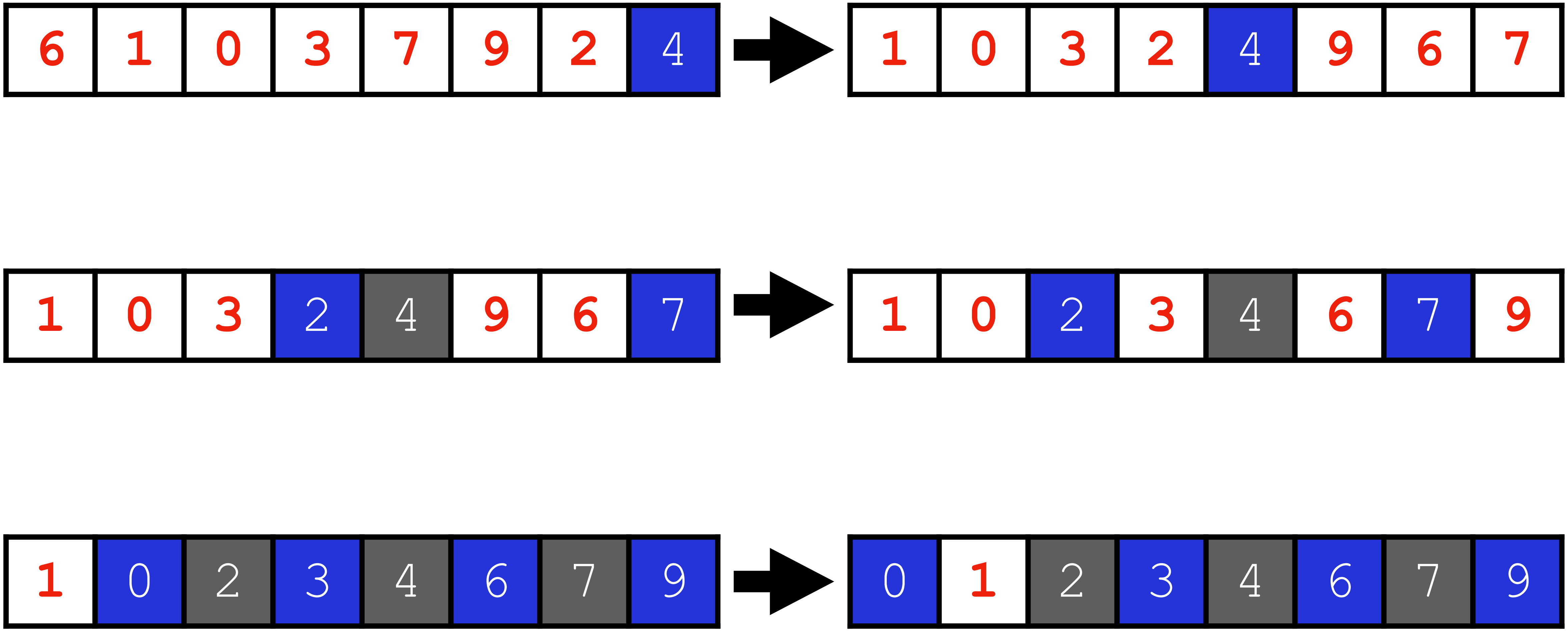


1. Choose a *pivot* value

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:

Recursive Step:

Combining: