Algorithms and Data Structures for Data Science
Sorting

CS 277
Brad Solomon

March 6, 2023

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN
Department of Computer Science

Reminder: mp_sketching due 3/20

MP is shorter than the first — can reasonably completed this week.

Exam 1 Retake

Retake exam much better than first!

Average: 76% Median: 83%

Overall exam seemed more ‘fair’

If you are concerned about your grade, talk to me!

Exam 2 and Final Exam Registration Open

Exam 2 is **immediately after spring break** (sorry)

Must take exam between March 21-23

Content includes everything prior to spring break:

- Hashing (Hash tables, bloom filters, k-minima, minhash)
- Sorting (Selection, Insertion, Merge, and Quicksort)
- Search (Binary search and binary range search)

Informal Early Feedback

Form available here: Informal Early Feedback

Form is entirely anonymous — please give constructive criticism!

If majority of class fill it out, everyone gets bonus points!

Learning Objectives

Conceptualize mergeSort

Introduce and implement quickSort

Discuss sorting algorithm tradeoffs and real world sorting

Recursive Array Sorting

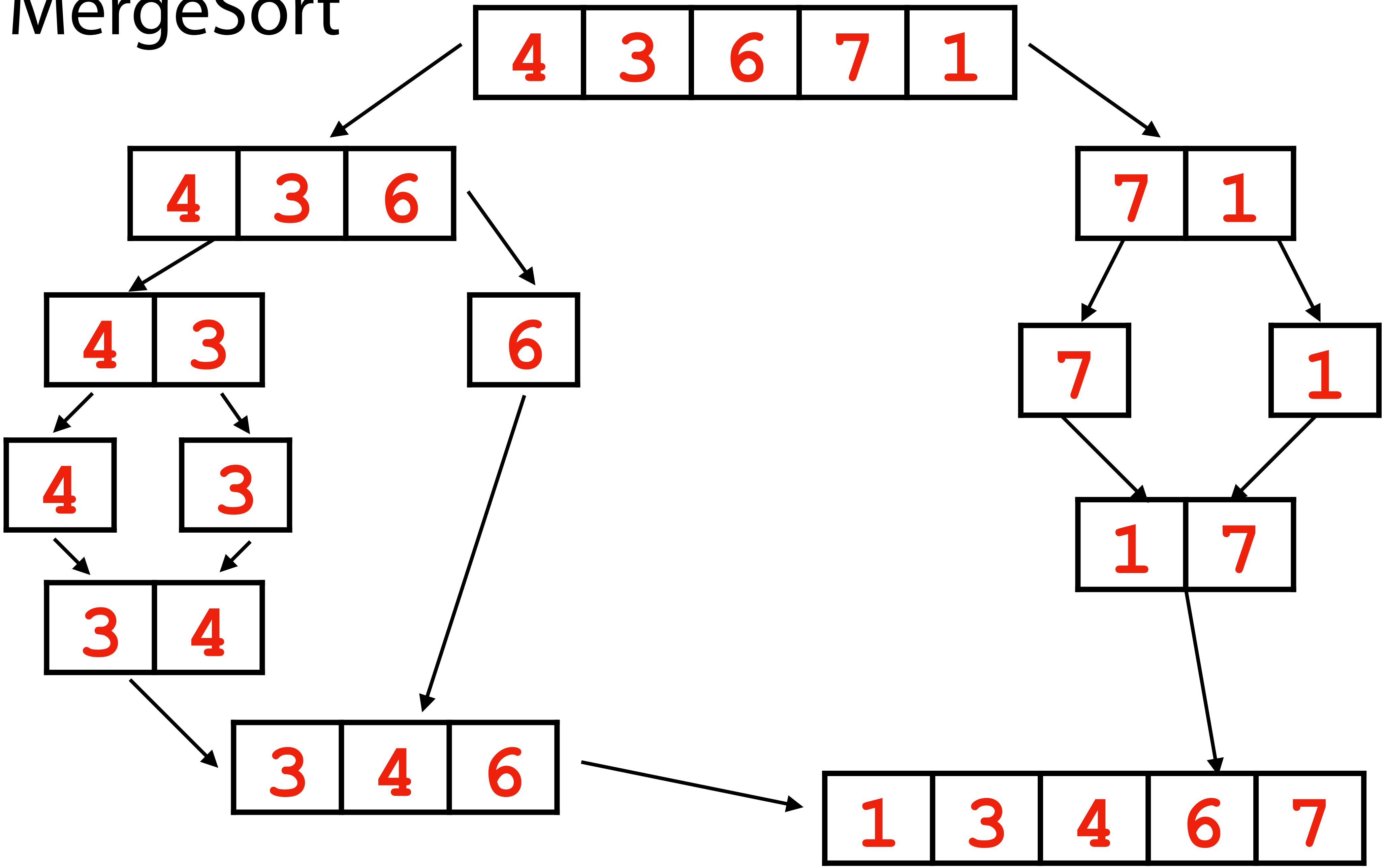
0	3	7	5	8	9	2	1	4	6

Base Case:

Recursive Step:

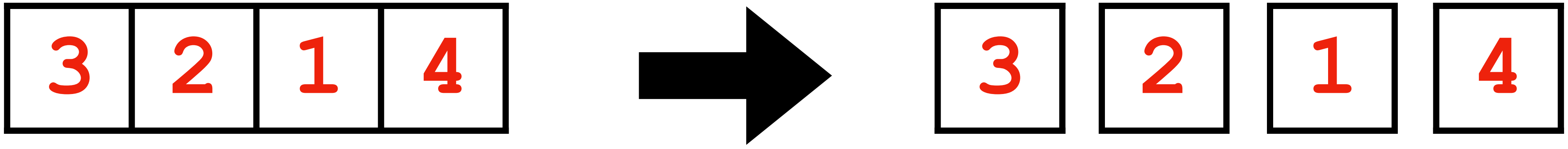
Combining:

MergeSort

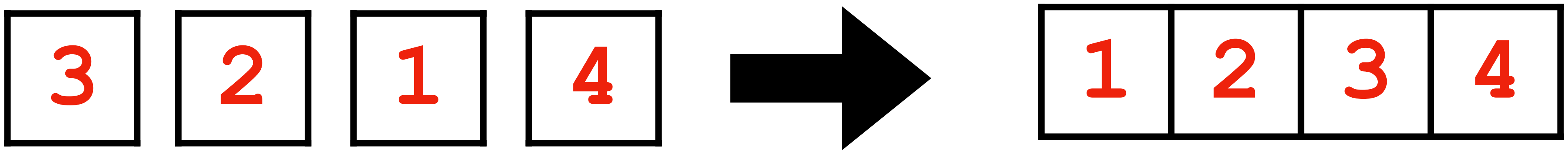


Recursive MergeSort

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

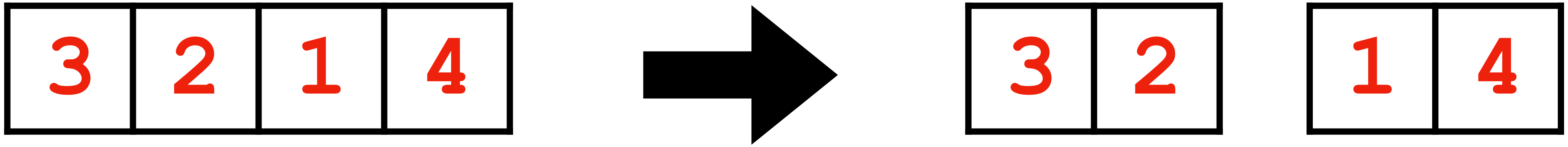


2) Sorted lists are merged back together



Recursive MergeSort

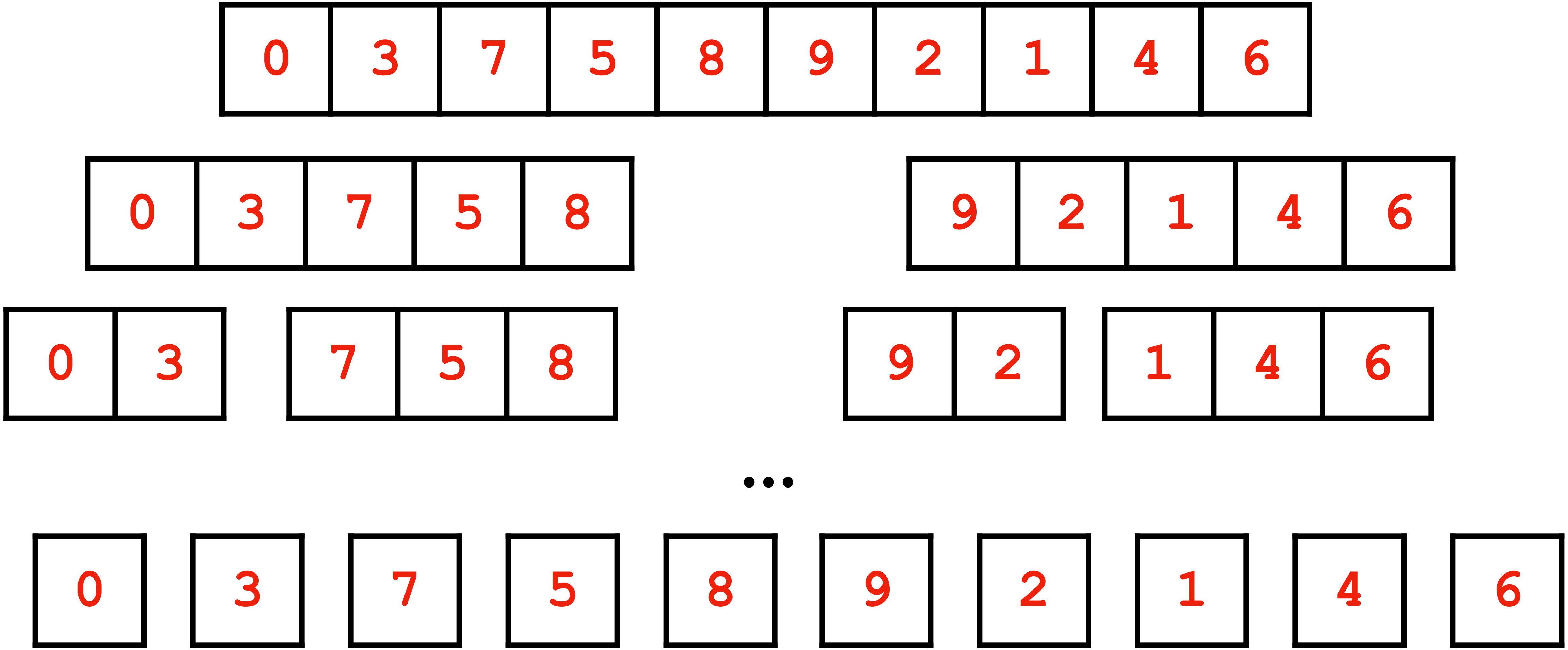
1) Input list recursively split to a collection of "sorted" base cases




```
def splitList(inList):

    if len(inList) > 1:
        mid = len(inList) // 2

        front = inList[:mid]
        back = inList[mid:]

        return front, back

    # Add cases for empty / single value

```

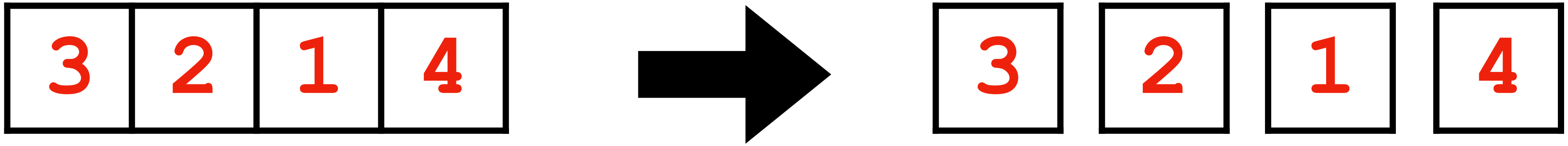
Recursive MergeSort

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



Recursive MergeSort

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

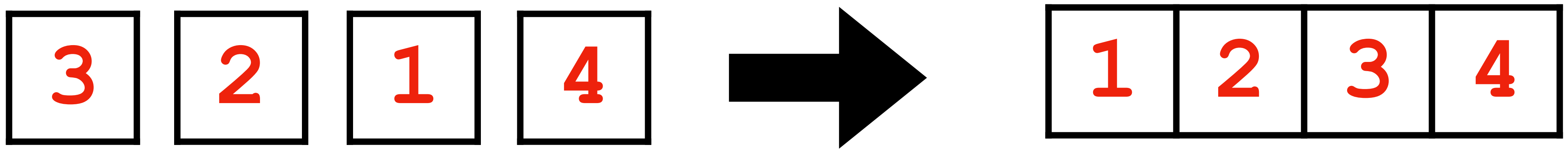


Time:

Space:

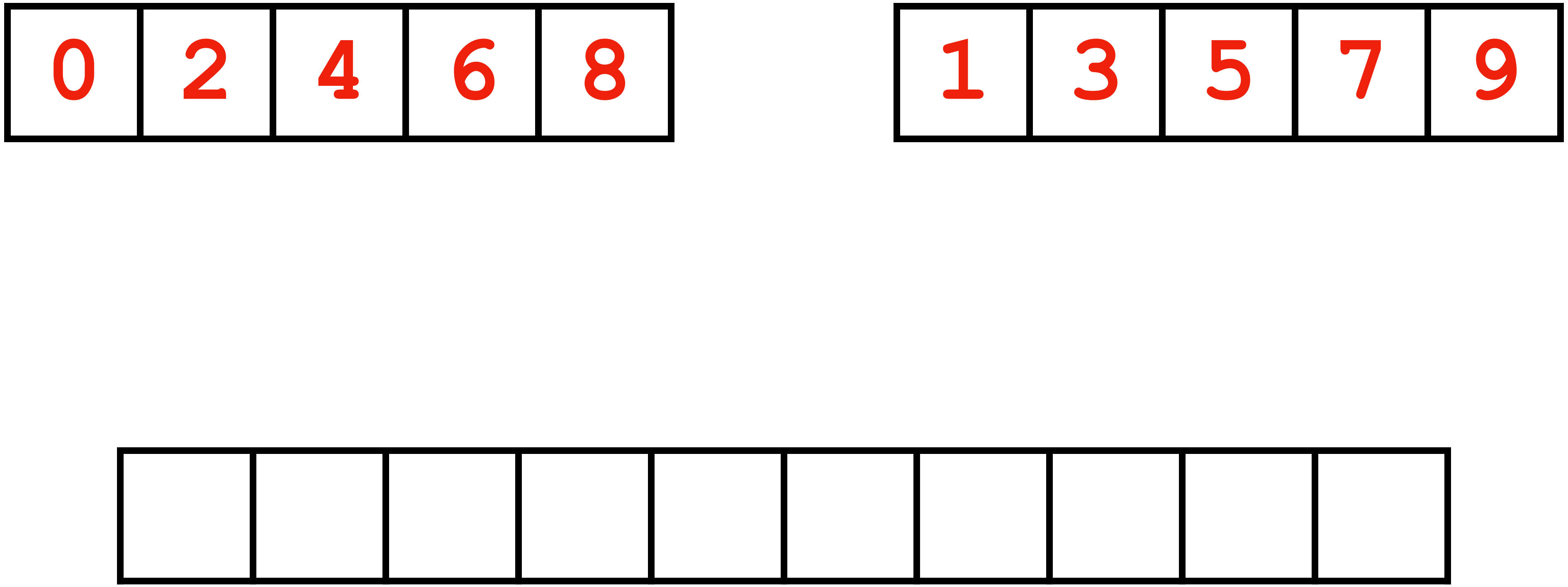
MergeSort Efficiency

2) Sorted lists are merged back together



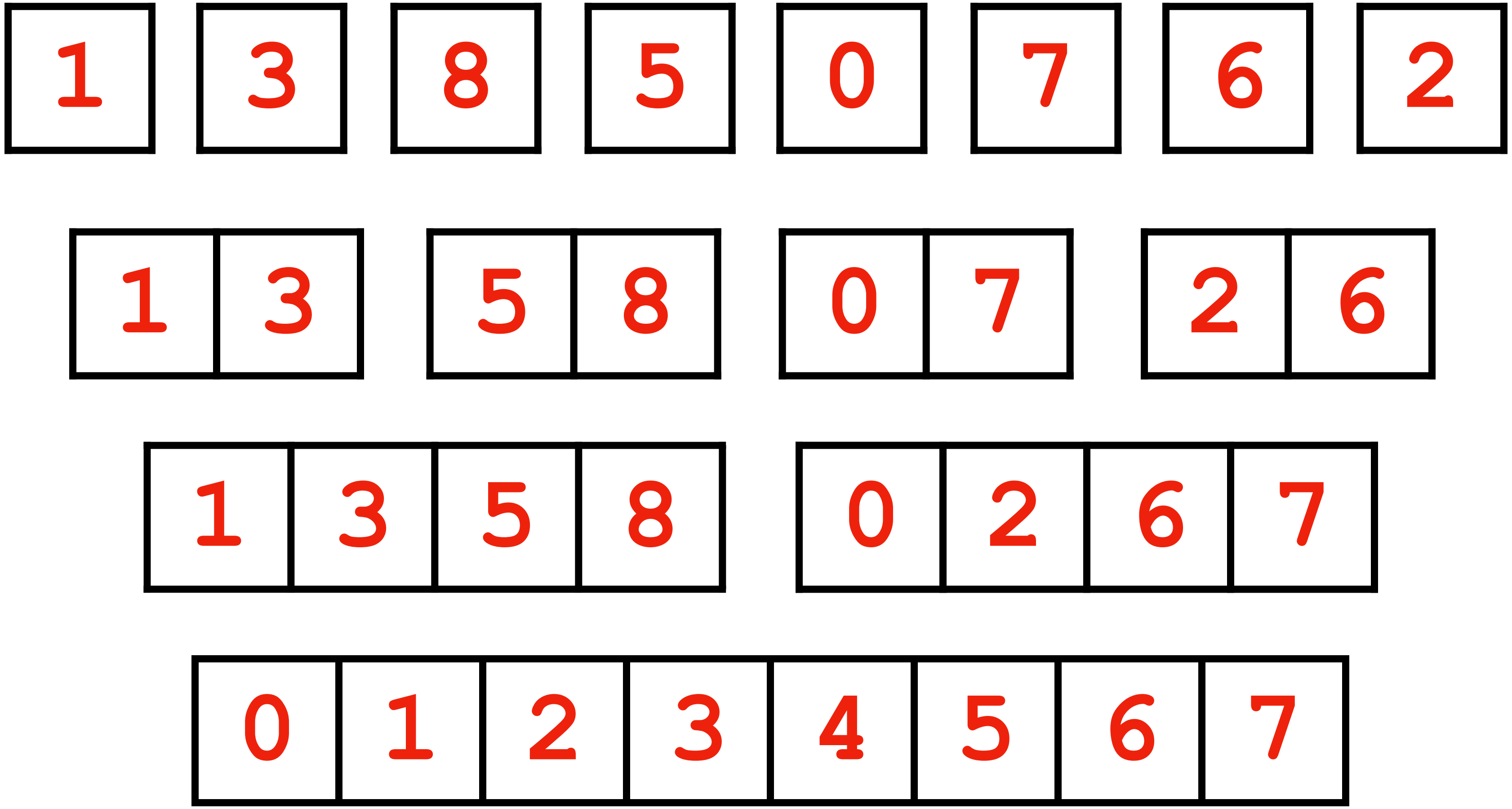
MergeSort Efficiency

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



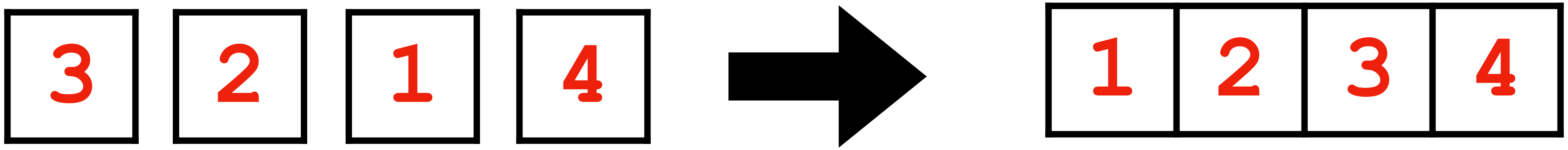
MergeSort Efficiency

2) Sorted lists are merged back together



MergeSort Efficiency

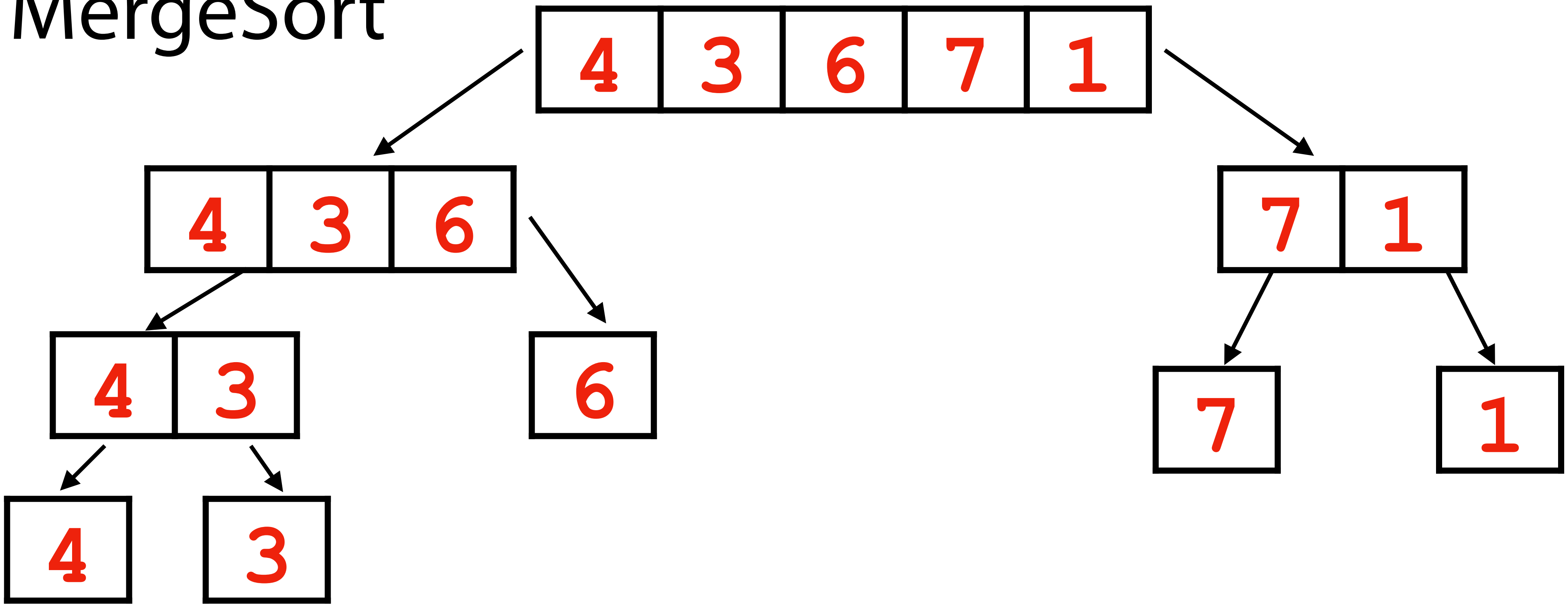
2) Sorted lists are merged back together



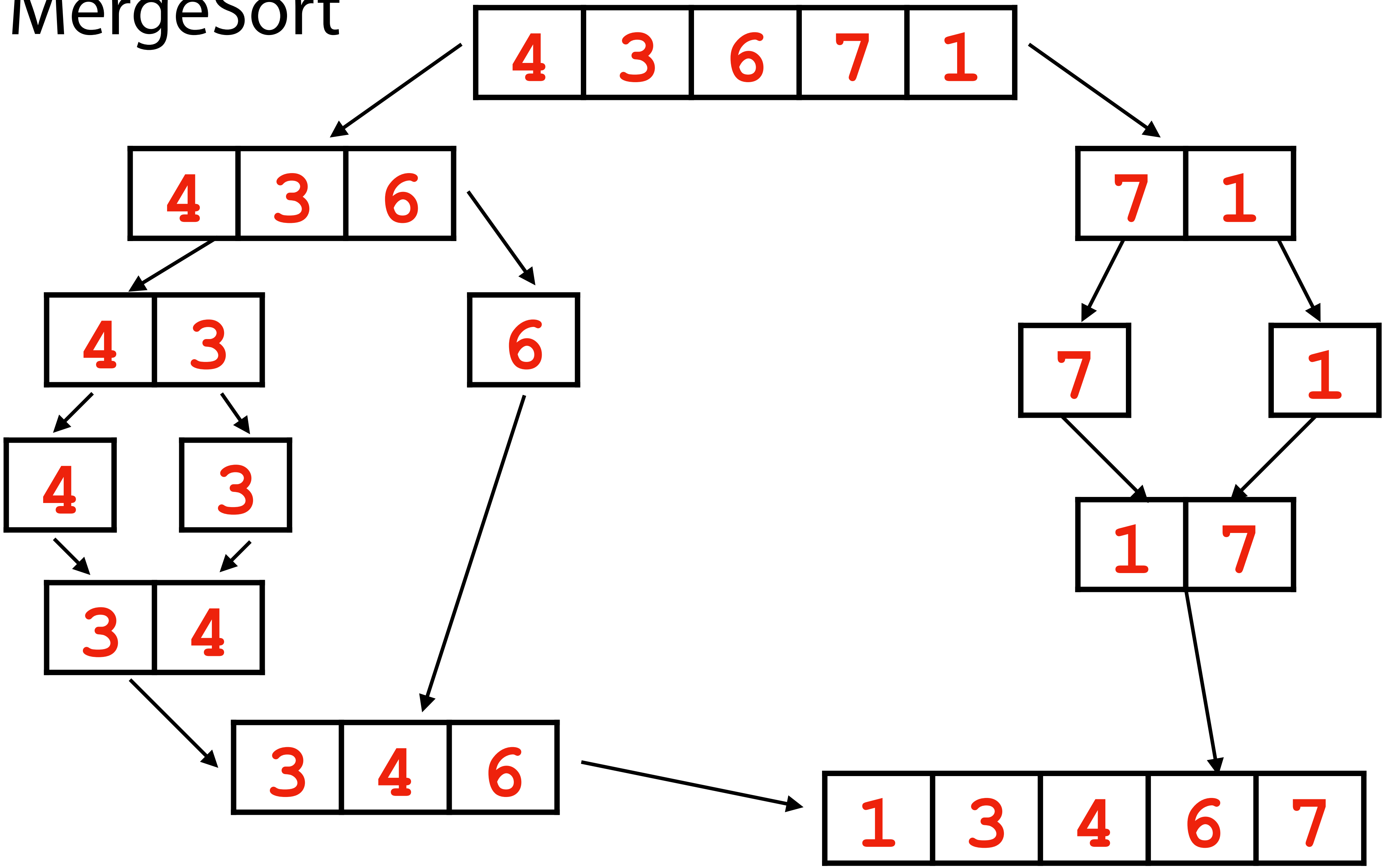
Time:

Space:

MergeSort



MergeSort



MergeSort by Index

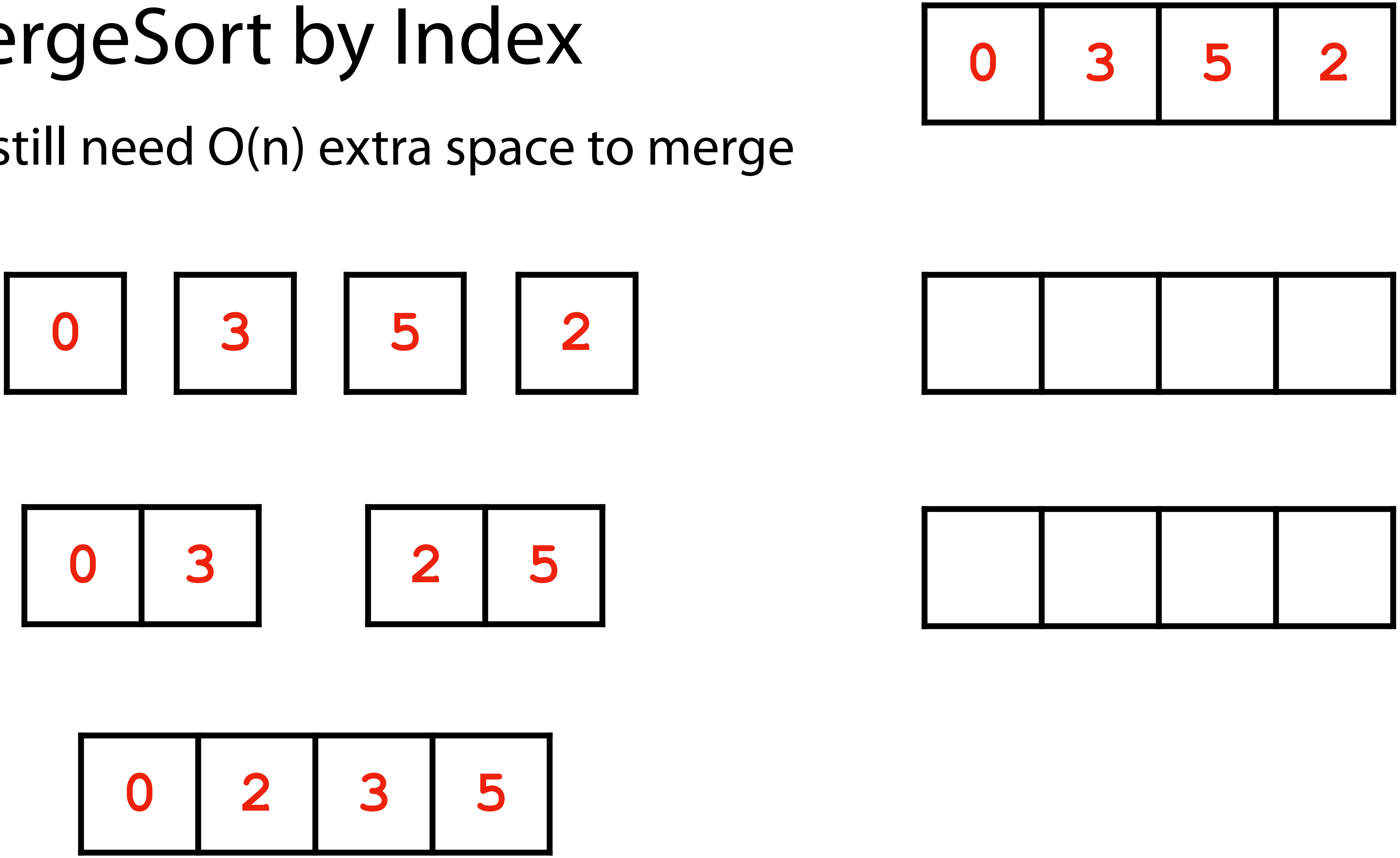
More efficient (in space) if we recurse without allocating new lists

```
def splitList_coords(inList):

    if len(inList) > 1:
        mid = len(inList) // 2

        frontCoords = (0, mid - 1)
        backCoords = (mid, len(inList) - 1)

        return frontCoords, backCoords

    # Add cases for empty / single value

```

0	3	2	5	4	1

MergeSort by Index

We still need O(n) extra space to merge



MergeSort

	Best Case	Worst Case
Time:		
Space:		

Sorting Algorithm Tradeoffs

	Best Case Time	Worst Case time	Best Case Space	Worst Case Space
SelectionSort				
InsertionSort				
MergeSort				

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	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

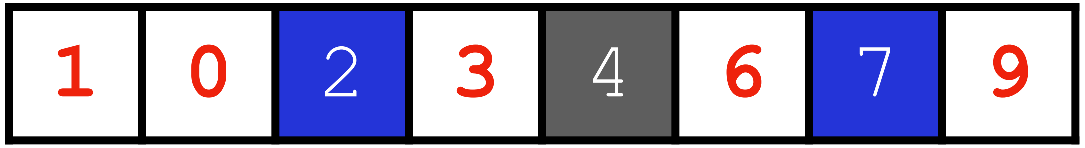
1	0	3	2	4	9	6	7

1. Choose a *pivot* value

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

3. Recursively QuickSort partitions

QuickSort

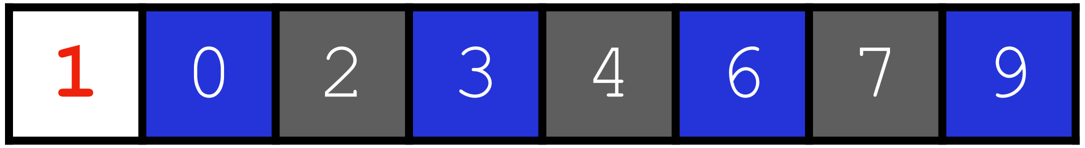


1. Choose a *pivot* value

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

3. Recursively QuickSort partitions

QuickSort

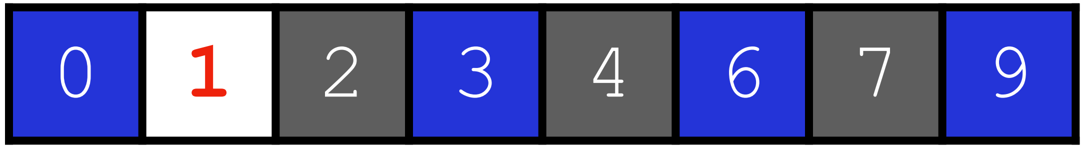


1. Choose a *pivot* value

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

3. Recursively QuickSort partitions

QuickSort

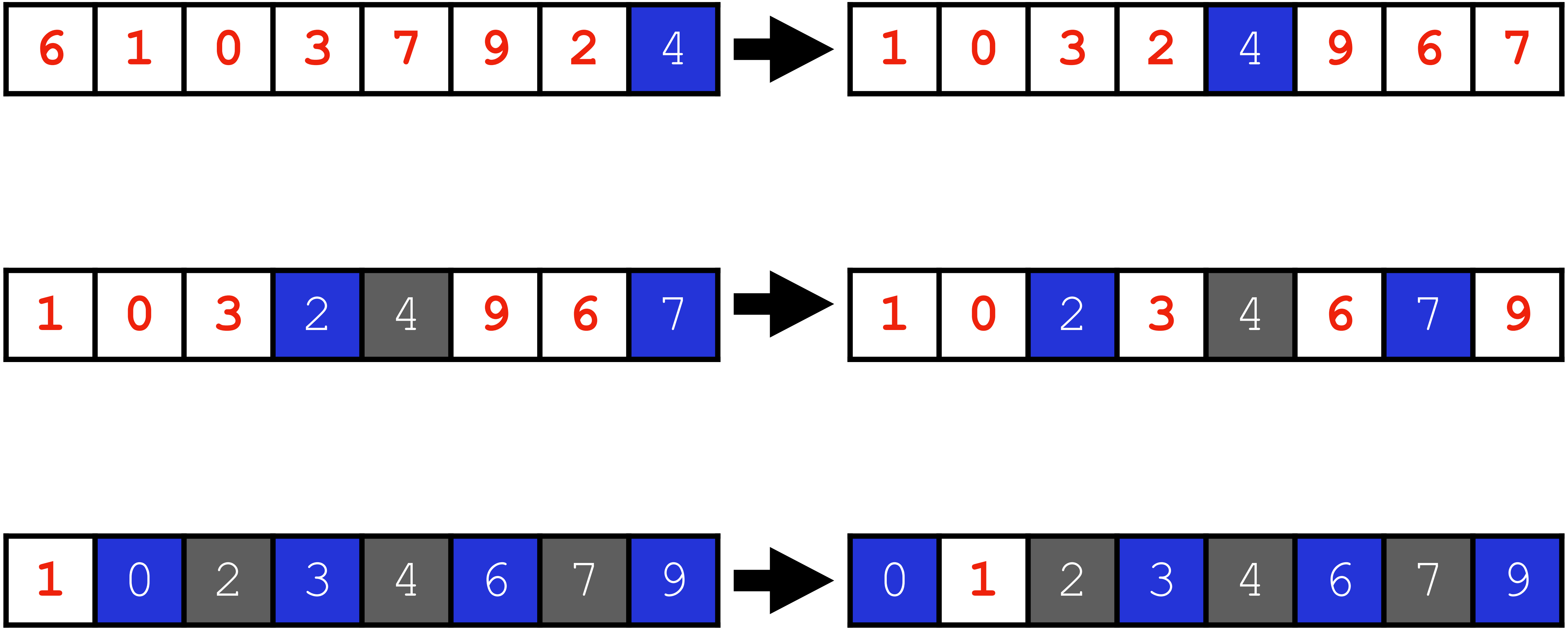


1. Choose a *pivot* value

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:

QuickSort is a ‘sort in place’ algorithm

1) quickSort recursion uses only a single array

```
def quickSort(inList, f, e):
    if f < e:
        p = partition(inList, f, e)
        quickSort(inList, f, p - 1)
        quickSort(inList, p + 1, e)
```

6	1	0	3	7	9	2	4

QuickSort is a ‘sort in place’ algorithm

1) quickSort recursion uses only a single array

```
def quickSort(inList, f, e):
    if f < e:
        p = partition(inList, f, e)


        quickSort(inList, f, p - 1)
        quickSort(inList, p + 1, e)
```

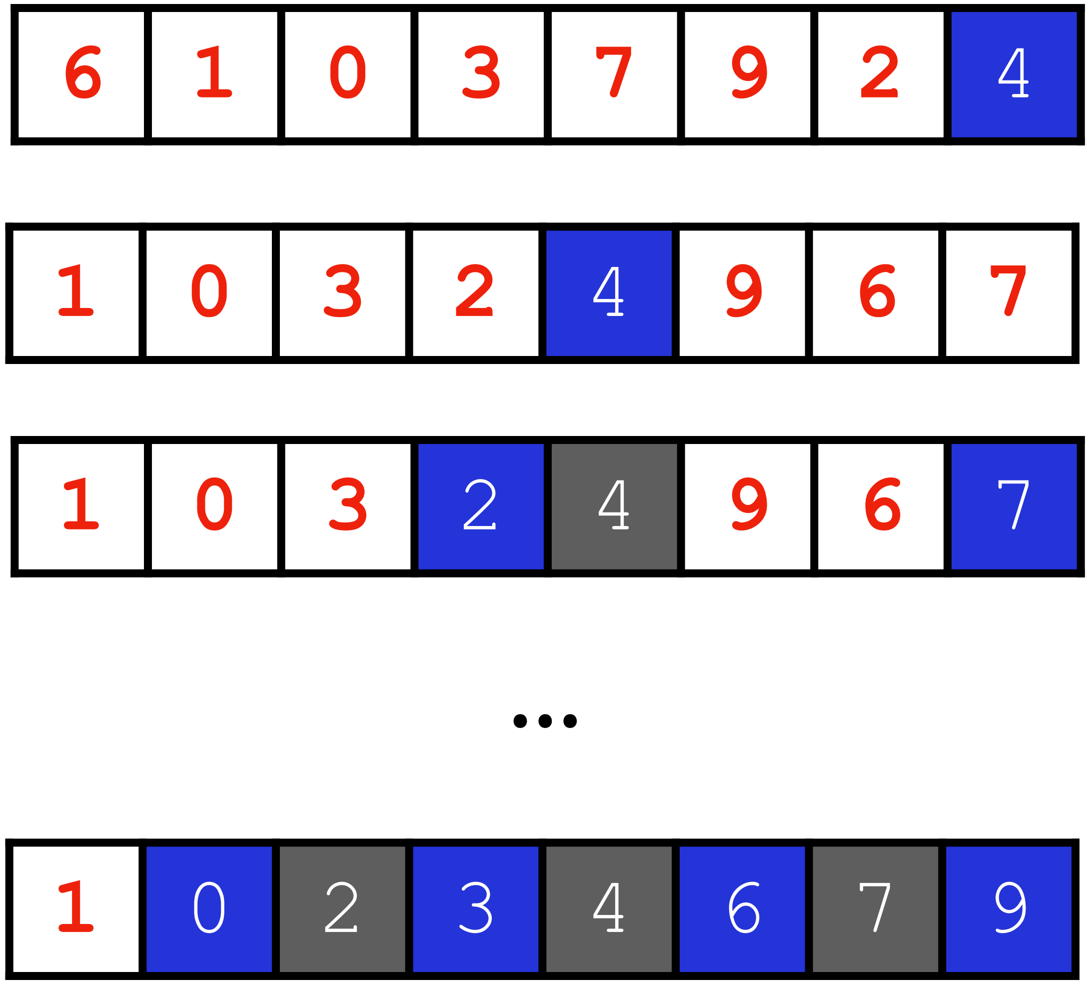
6	1	0	3	7	9	2	4

1	0	3	2	4	9	6	7

QuickSort is a 'sort in place' algorithm

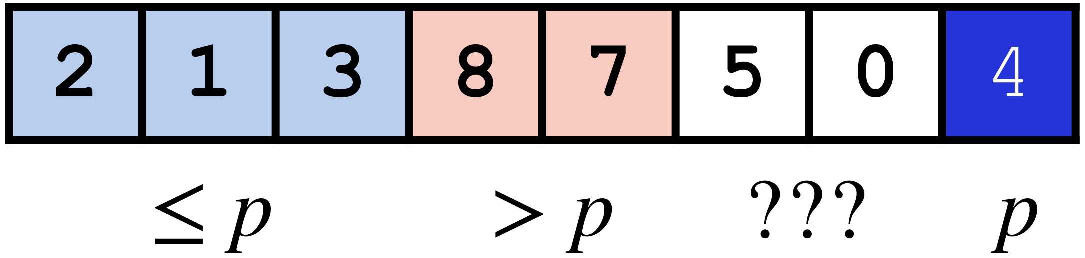
1) quickSort recursion uses only a single array

```
def quickSort(inList, f, e):

    if f < e:

        p = partition(inList, f, e)



        quickSort(inList, f, p - 1)

        quickSort(inList, p + 1, e)


```



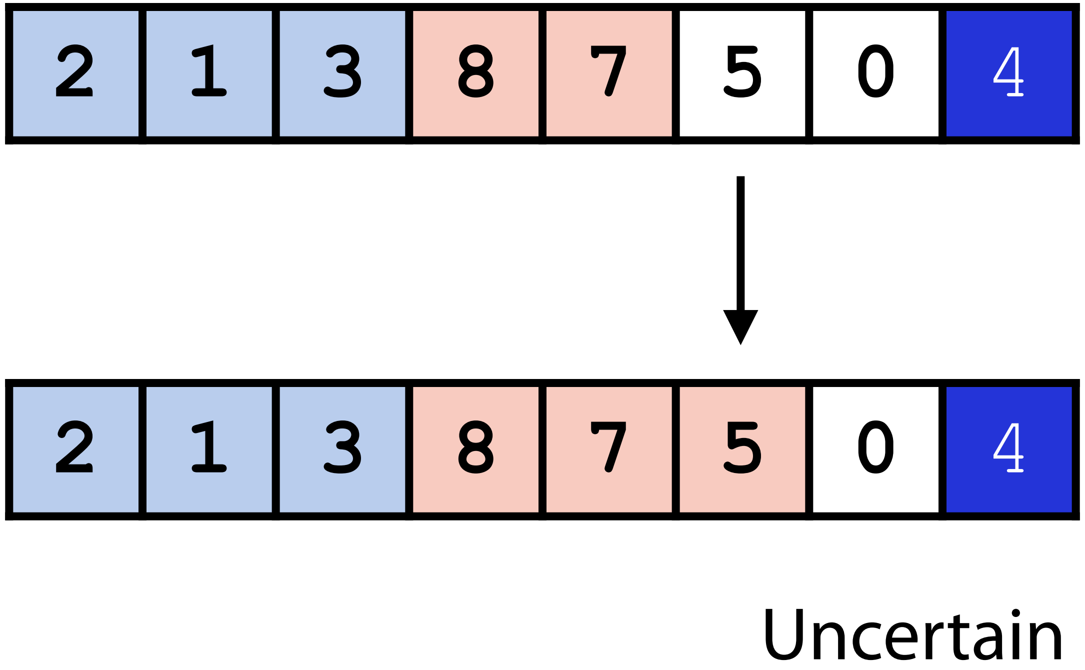
QuickSort is a 'sort in place' algorithm

2) quickSort partitions the array into four regions



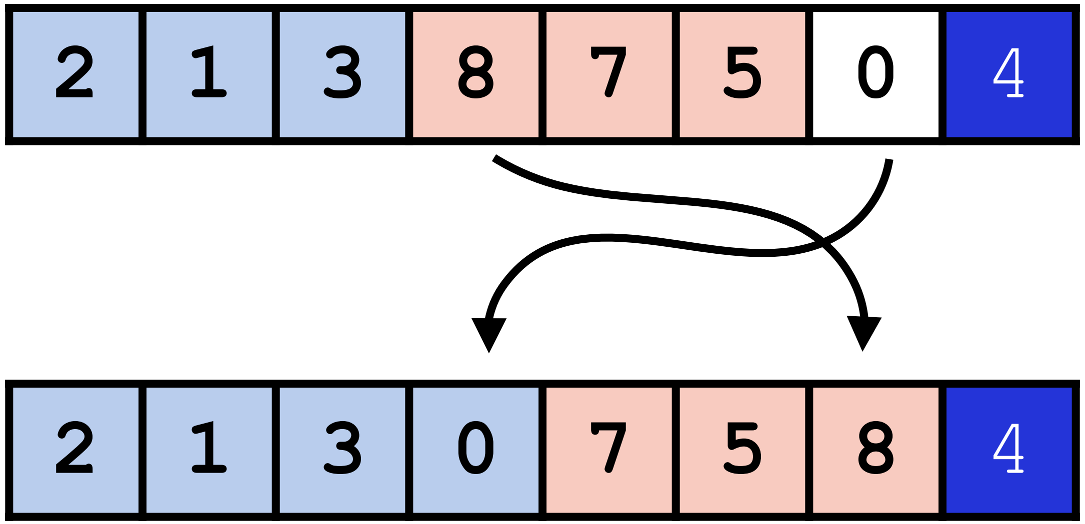
QuickSort is a 'sort in place' algorithm

2) quickSort partitions the array into four regions



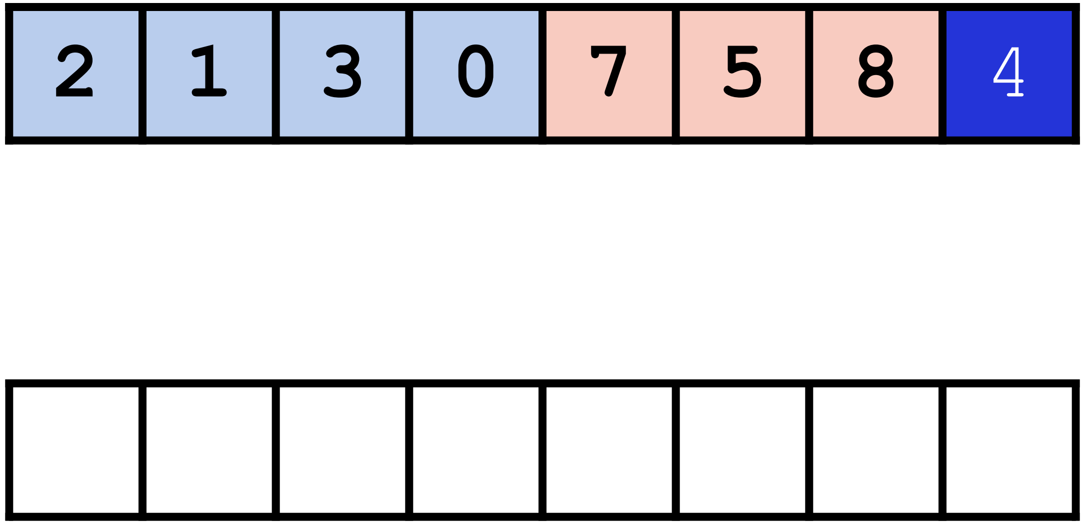
QuickSort is a 'sort in place' algorithm

2) quickSort partitions the array into four regions



QuickSort is a 'sort in place' algorithm

3) The last step in partition swaps the pivot



QuickSort

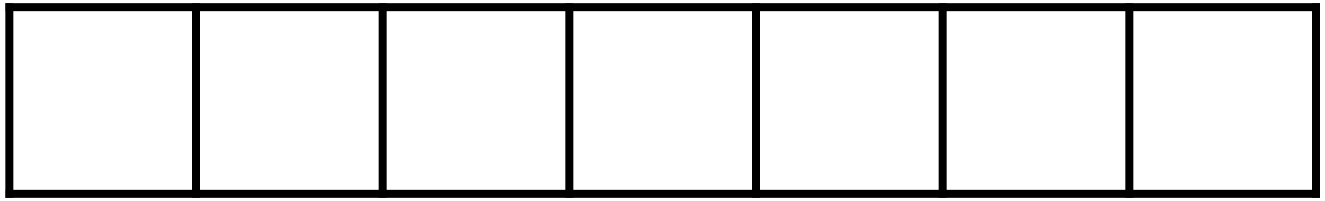
						3	7	9	2	4

```
def partition(inList, f, e):

    pivot = inList[e]

    i = f

    for j in range(f, e):

        if inList[j] <= pivot:

            inList[i], inList[j] = inList[j], inList[i]

            i+=1

    inList[i], inList[e] = inList[e], inList[i]

    return i
```

3	7	9	2	4

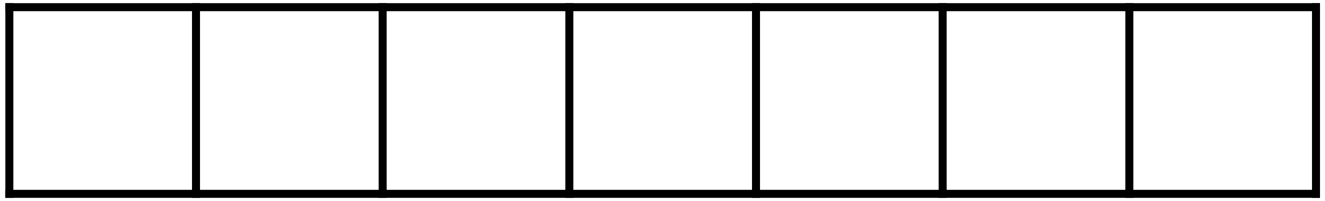
Best Case quickSort

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



Worst Case quickSort

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



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	$O(n \log n)$	$O(n \log n)$	$O(n)$	$O(n)$
QuickSort				

Selecting the pivot for quickSort

Can we do better than ‘pick the last element in the list’?

Next Class: Search

A sorted list is more efficient to search than a list — but how?

Consider: When do you want to search with a hash table vs a list?

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