Topic 12.1: Analysis of Algorithms

Learning Goals:

- Describe why Computer Scientists care about the major “steps” in an algorithm over raw measurements like CPU time.
- Express the complexity of a basic algorithm using big-O notation.
- Compare and contrast the running times of linear versus binary search using big-O notation.
- Compare and contrast the running times of insertion, selection, merge, and quick sort using big-O notation

Merge Sort: A Quick Reminder

- The basic algorithm we saw previously:
  - Split the array into two small arrays (half each)
  - Sort the two halves (using to merge sort calls)
  - (recurse over all of this)

- Merge the two sorted halves into a sorted array after the two splits have returned

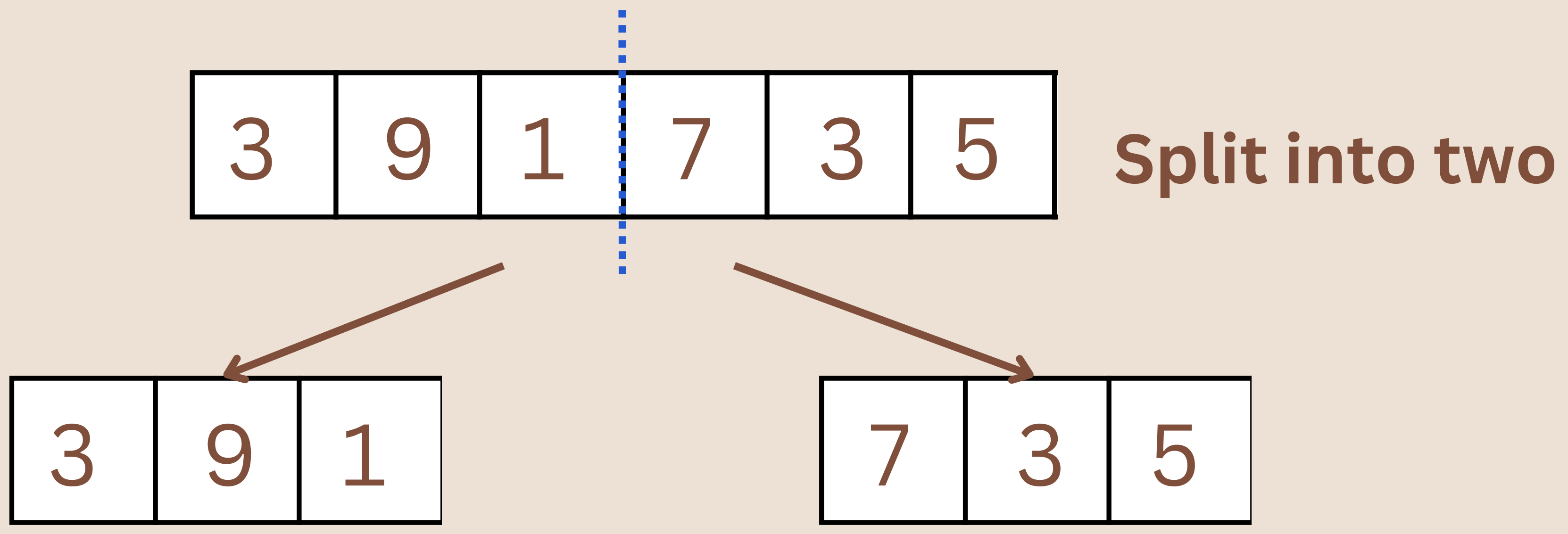
Merge Sort: An Example

- A review of what we previously saw:

3	9	1	7	3	5

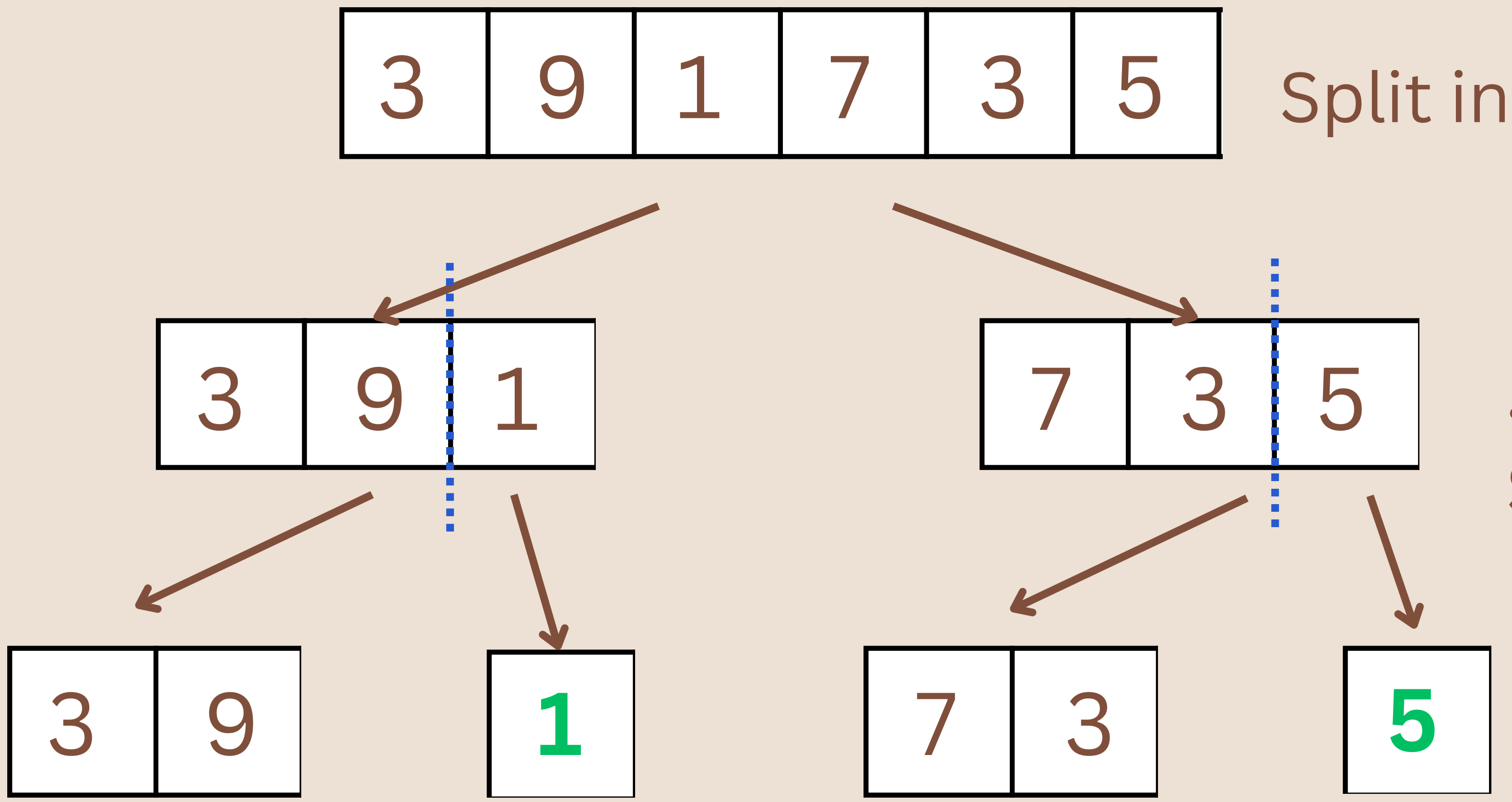
Merge Sort: An Example

- A review of what we previously saw:



Merge Sort: An Example

- A review of what we previously saw:

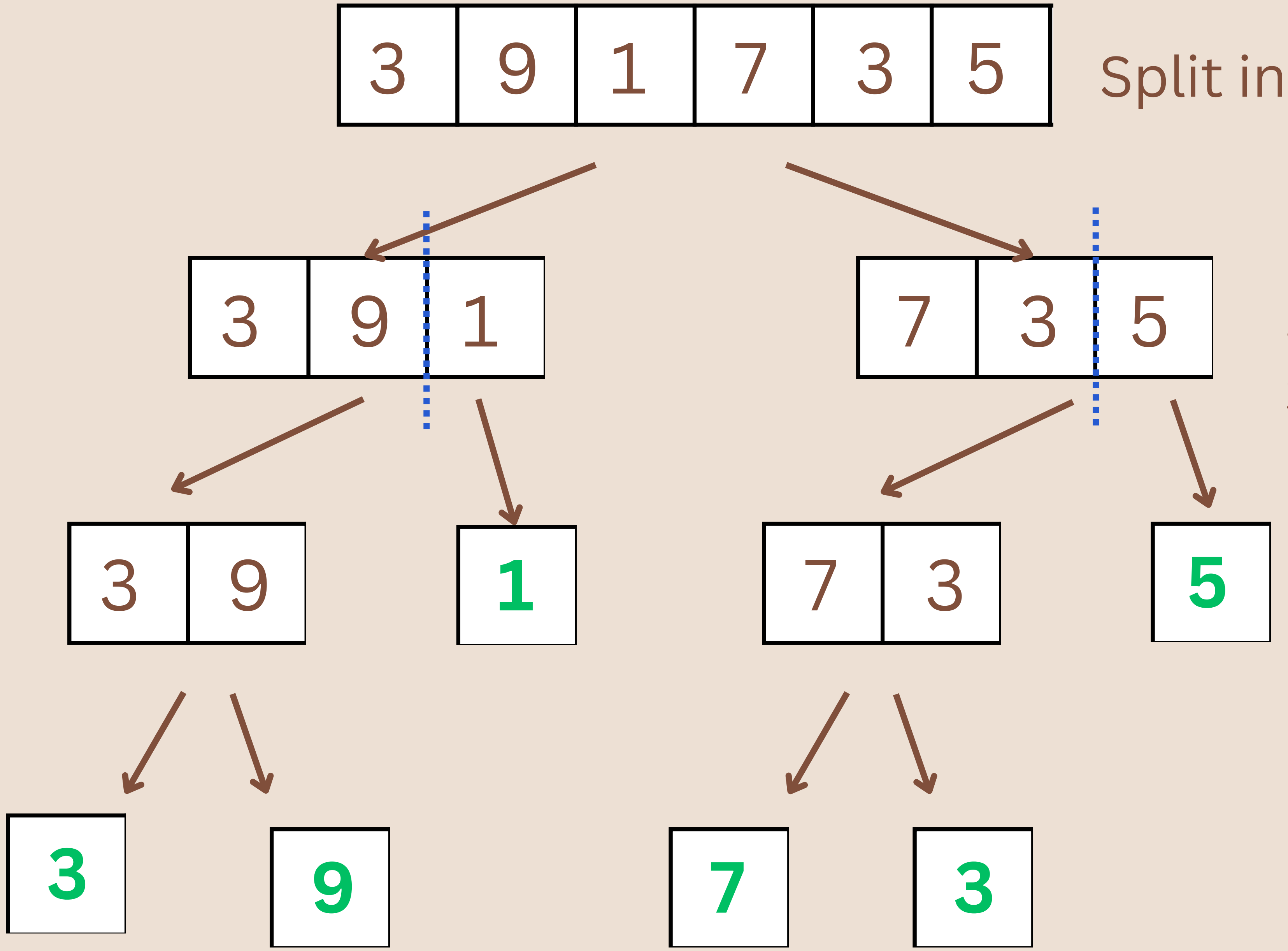


Split into two

...and call mergeSort on each half
Split into two

Merge Sort: An Example

- A review of what we previously saw:



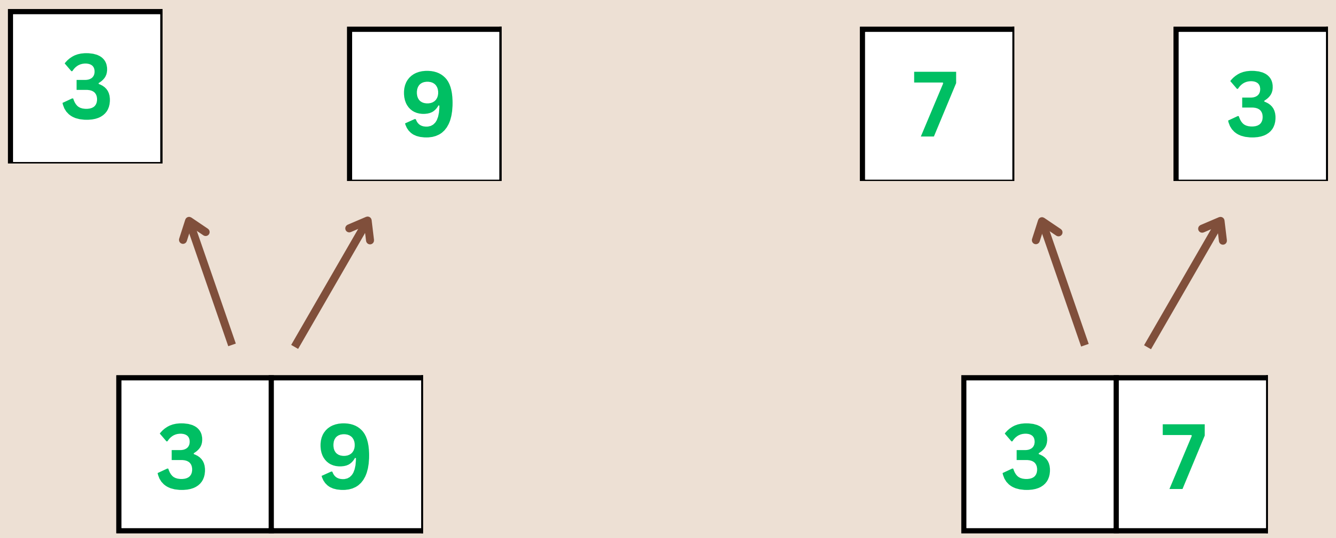
Split into two

...and call mergeSort on each half
Split into two

...and call mergeSort on each half
No further splitting on base case

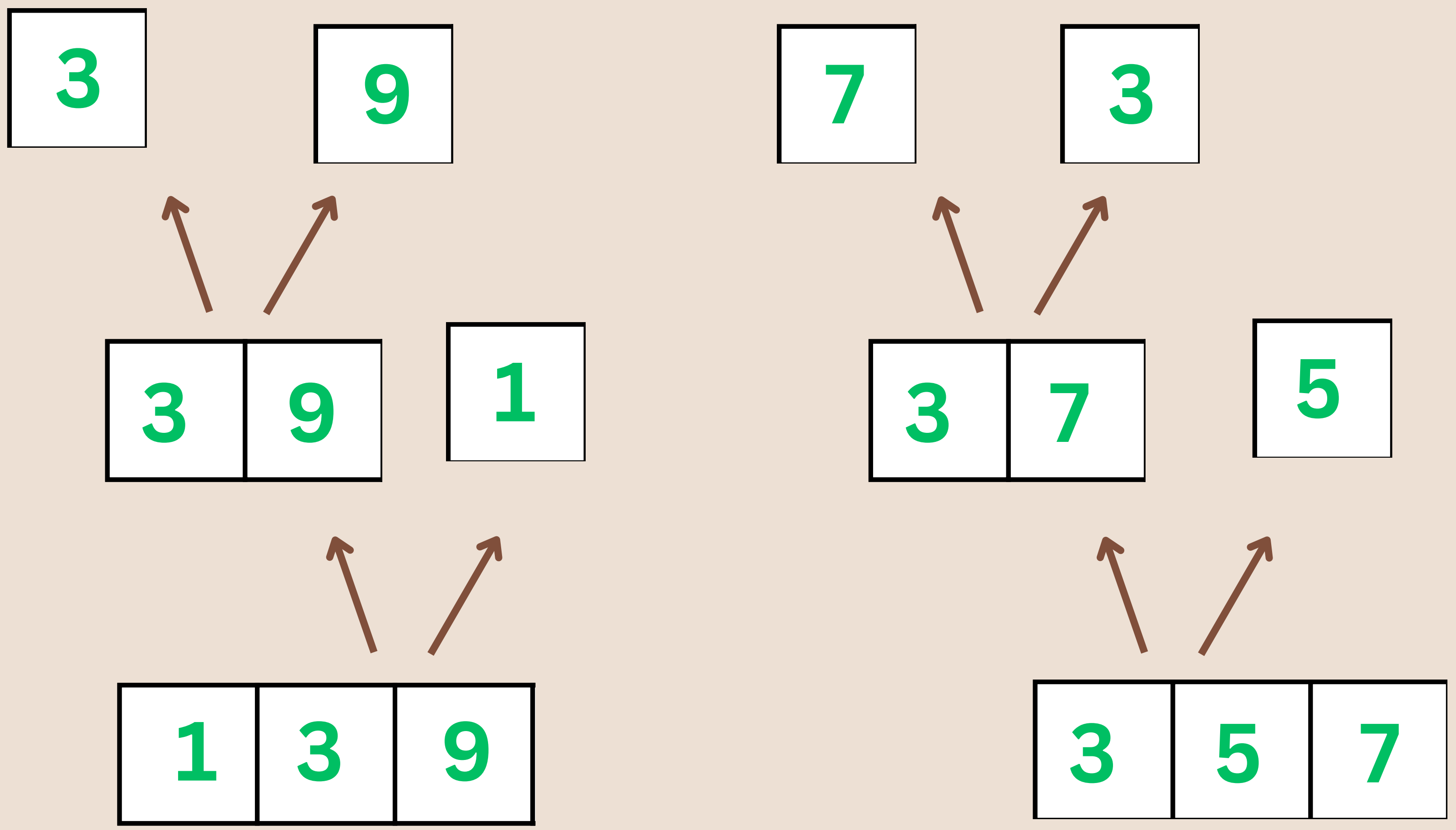
Merge Sort: An Example

- Time to merge back up



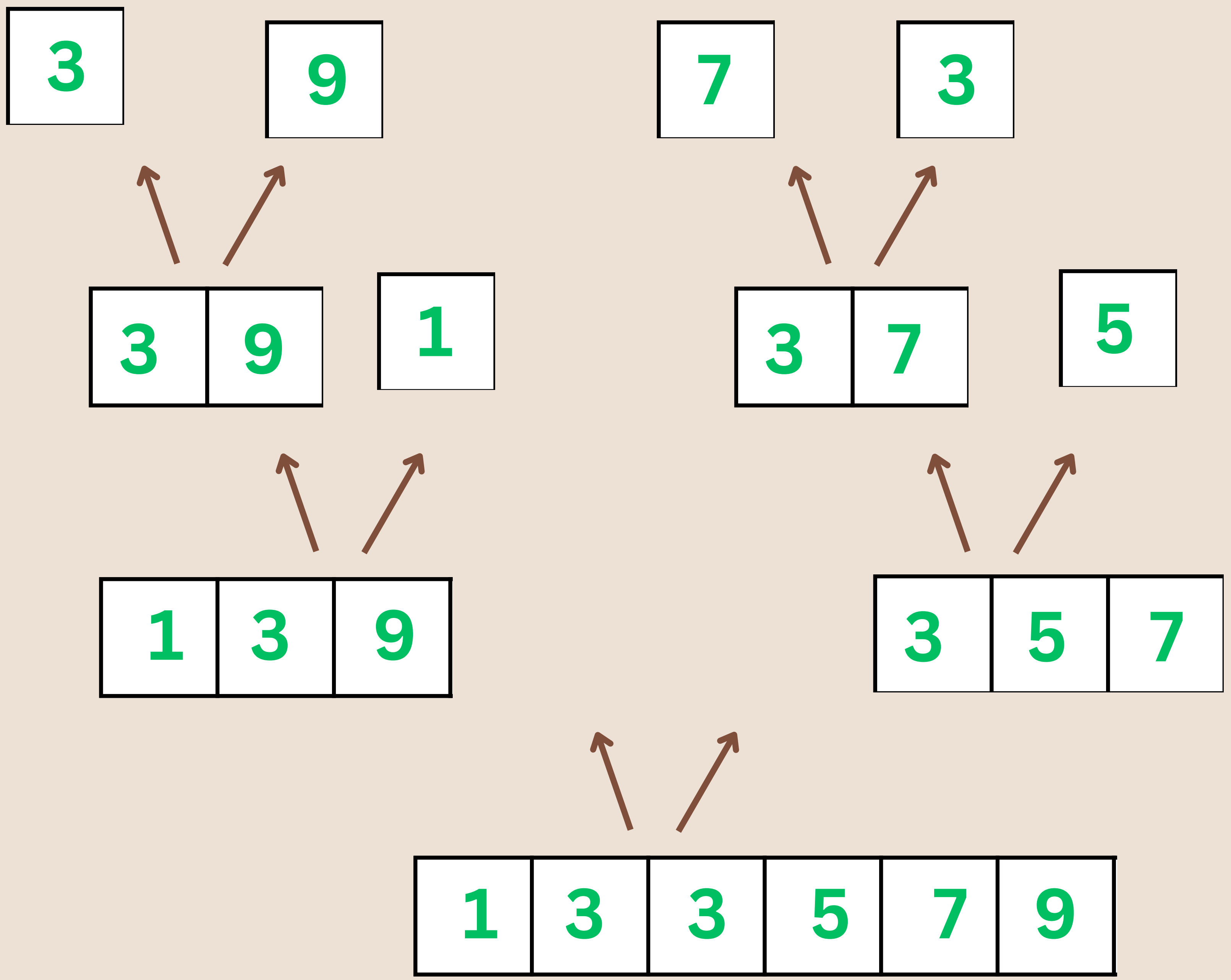
Merge Sort: An Example

- Time to merge back up



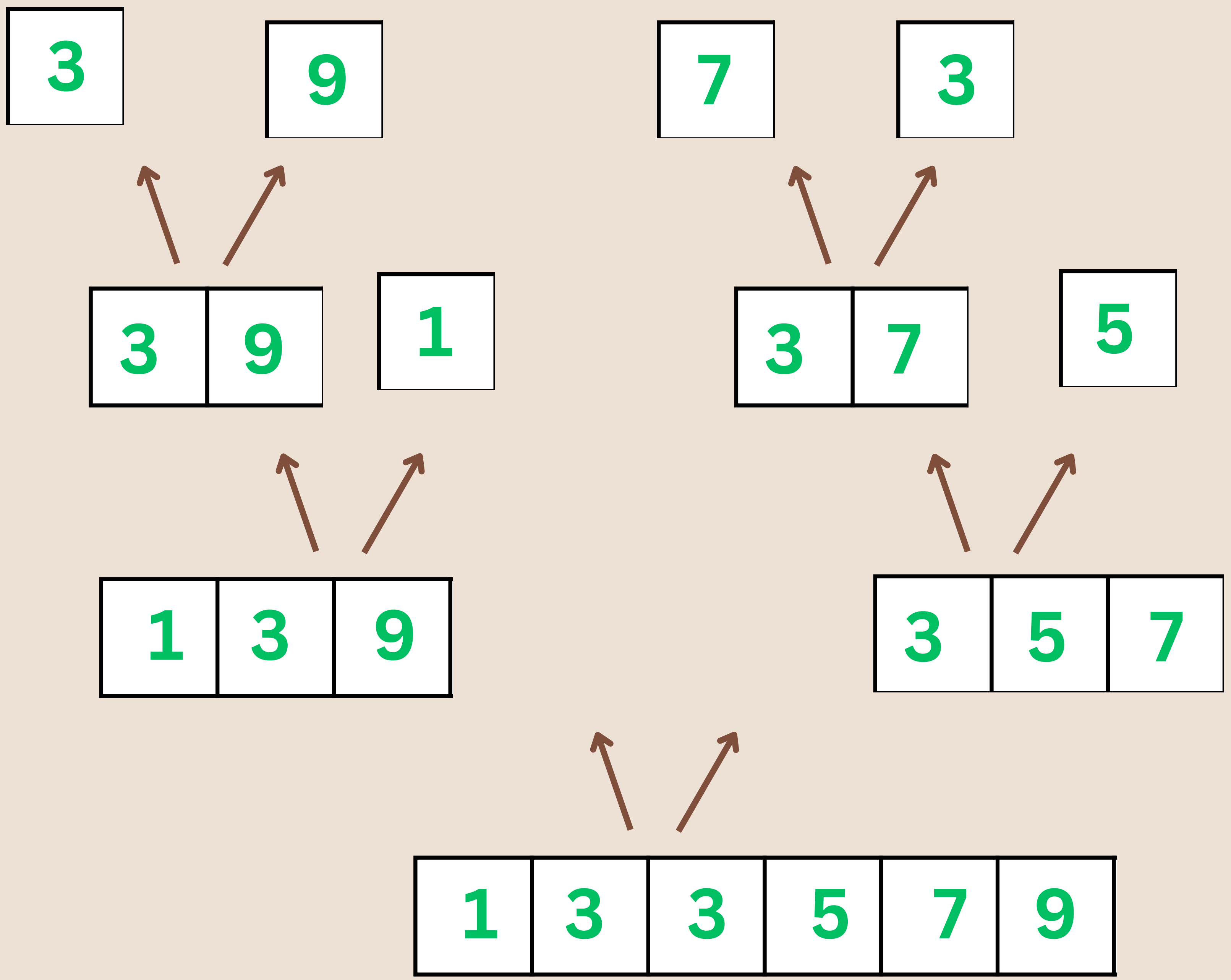
Merge Sort: An Example

- Time to merge back up



Merge Sort: An Example

- Time to merge back up



- But of course this doesn't really happen "all at once" or "in - parallel"

- Let's take a look at the runtime stack

Merge Sort: An Example

sorted = mergeSort(3,9,1,7,3,5)

main()

Merge Sort: An Example

sorted = mergeSort(3,9,1)

sorted = mergeSort(3,9,1,7,3,5)

main()

Merge Sort: An Example

sorted = mergeSort(3,9)

sorted = mergeSort(3,9,1)

sorted = mergeSort(3,9,1,7,3,5)

main()

Merge Sort: An Example

sorted = mergeSort(3)

sorted = mergeSort(3,9)

sorted = mergeSort(3,9,1)

sorted = mergeSort(3,9,1,7,3,5)

main()

Merge Sort: An Example

sorted = mergeSort(9)

~~sorted = mergeSort(3)~~ //base case: left = [3]

sorted = mergeSort(3,9)

sorted = mergeSort(3,9,1)

sorted = mergeSort(3,9,1,7,3,5)

main()

Merge Sort: An Example

~~sorted = mergeSort(9)~~ //base case: right = [9]

~~sorted = mergeSort(3)~~ //base case: left = [3]

sorted = mergeSort(3,9)

sorted = mergeSort(3,9,1)

sorted = mergeSort(3,9,1,7,3,5)

main()

Merge Sort: An Example

sorted = merge([3],[9]) //sorted [3,9]

sorted = mergeSort(3,9)

sorted = mergeSort(3,9,1)

sorted = mergeSort(3,9,1,7,3,5)

main()

Merge Sort: An Example

~~sorted = merge([3],[9])~~ //sorted [3,9]

~~sorted = mergeSort(3,9)~~

sorted = mergeSort(3,9,1)

sorted = mergeSort(3,9,1,7,3,5)

main()

Merge Sort: An Example

sorted = mergeSort(1)

sorted = mergeSort(3,9,1)

//left = [3,9]

sorted = mergeSort(3,9,1,7,3,5)

main()

Merge Sort: An Example

sorted = mergeSort(1)

//sorted = [1]

sorted = mergeSort(3,9,1)

//left = [3,9]

sorted = mergeSort(3,9,1,7,3,5)

main()

Merge Sort: An Example

~~sorted = mergeSort(1)~~

sorted = mergeSort(3,9,1)   //left = [3,9] and right = [1]

sorted = mergeSort(3,9,1,7,3,5)

main()

Merge Sort: An Example

sorted = merge([3,9], [1]) //sorted [1,3,9]

sorted = mergeSort(3,9,1)

sorted = mergeSort(3,9,1,7,3,5)

main()

Merge Sort: An Example

sorted = mergeSort(3,9,1) //sorted = [1,3,9]

sorted = mergeSort(3,9,1,7,3,5)

main()

Merge Sort: An Example

~~sorted = mergeSort([illegible])~~

sorted = mergeSort(3,9,1,7,3,5) //left = [1,3,9]

main()

Merge Sort: An Example

sorted = mergeSort(7,3,5)

sorted = mergeSort(3,9,1,7,3,5)

main()

//same process down the right

//left = [1,3,9]

Merge Sort: An Example

~~sorted = mergeSort([illegible])~~

sorted = mergeSort(3,9,1,7,3,5) //left = [1,3,9] and right = [3,5,7]

main()

Merge Sort: An Example

sorted = merge(left, right) //left = [1,3,9] and right = [3,5,7]

sorted = mergeSort(3,9,1,7,3,5) //left = [1,3,9] and right = [3,5,7]

main()

Merge Sort: An Example

~~sorted = mergeSort(left, right)~~ ~~//~~left = [1,3,9] and right = [3,5,7]

sorted = mergeSort(3,9,1,7,3,5) // sorted = [1,3,3,5,7,9]

main()

Merge Sort: An Example

main() // sorted = [1,3,3,5,7,9]

Merge Sort: An Example

~~main()~~ // Empty stack: Program Exit

Merge Sort: Analysis

- This sort is actually pretty fast
- Did I hear you ask how fast?

Merge Sort: Analysis

- This sort is actually pretty fast
- Did I hear you ask how fast?

- The arrays are divided in half a total of logn times **(we have seen this before a binary search!)**
- On each "level" (you can see the levels in the previous example; there are **logn levels**), we have to **merge O(n) elements**

- Result: merge sort is in **O(nlogn)**

Extra Content: Another Sort

- Computer Scientists love sorting algorithms (Wikipedia lists 43 different algorithms and I am sure that is not exhaustive....wikipedia being wikipedia)
- Let's talk about another way to sort data:
  - choose a "pivot" value
  - split our array into bigger than or smaller than the pivot (left/right side of the array)
  - place the pivot in the middle of the two
  - do it again on both the left and right sides

- Similar to merge sort with divide/conquer and partitioning in half
- Different because we are choosing a "pivot" value
- That pivot value choice matters A LOT (default: just pick the first one)
- This is called Quick Sort
- [Here is a visualizer that chooses the middle element as the pivot]()

Extra Content: Quick Sort

- This is also a fast sort in practice
- **How fast did you say!? Well, this is why Quick Sort is neat....**

- ***If*** we are lucky with the choices of the pivots, the arrays are divided in half roughly **logn** times

- On each “level” (you can see the levels in the previous example), we have to partition **O(n)** elements
- Result: quick sort is in **O(nlogn)** on **average**

Extra Content: Quick Sort

- **BUT**
- If we make bad choices for the pivot, the arrays are not divided in half and we lose the efficiency of quick sort
- Worst-case scenario: choose the first value in the array for the pivot, but the array is already sorted…
- Result: we remove only 1 cell at a time, and we have to call quick sort O(n) times (instead of logn)

- Quick Sort is in **O(n2)** in the **worst case**

- There are lots of ways designed to choose a better pivot and you'll see more Quick Sort in your second year

Main Takeaways

- **You should know:**
  - how to roughly analyze the speed of algorithm (**not just memorize the analyses you have seen but how to implement this process of analyzing them**)
  - The names of all the algorithms we have seen and what they are
  - The general idea for each of them
  - Reminder: You should be able to implement simple searches/sorts and describe the general idea of merge/quick sort.

Pause and Practice

- Think about the code you've written throughout this course.
  - What is the running time of your search and finds? Could you make them better?
  - What is the running time of inserting to the front of a Singly Linked List? Array? ArrayList?
    - Is there a situation where it would be worth using an Array but also requiring front insertion?
- Consider different scenarios you encounter in your life and think about how efficient the algorithm must be that does those things:
  - Virtual Taylor Swift Queue?
  - Priority Registration for Classes?
  - The best way to spend your weekly grocery budget?
  - Getting through the corn maze at A Maze N Corn in Fall?

YOU DID IT! WE ARE DONE! CONGRATS!