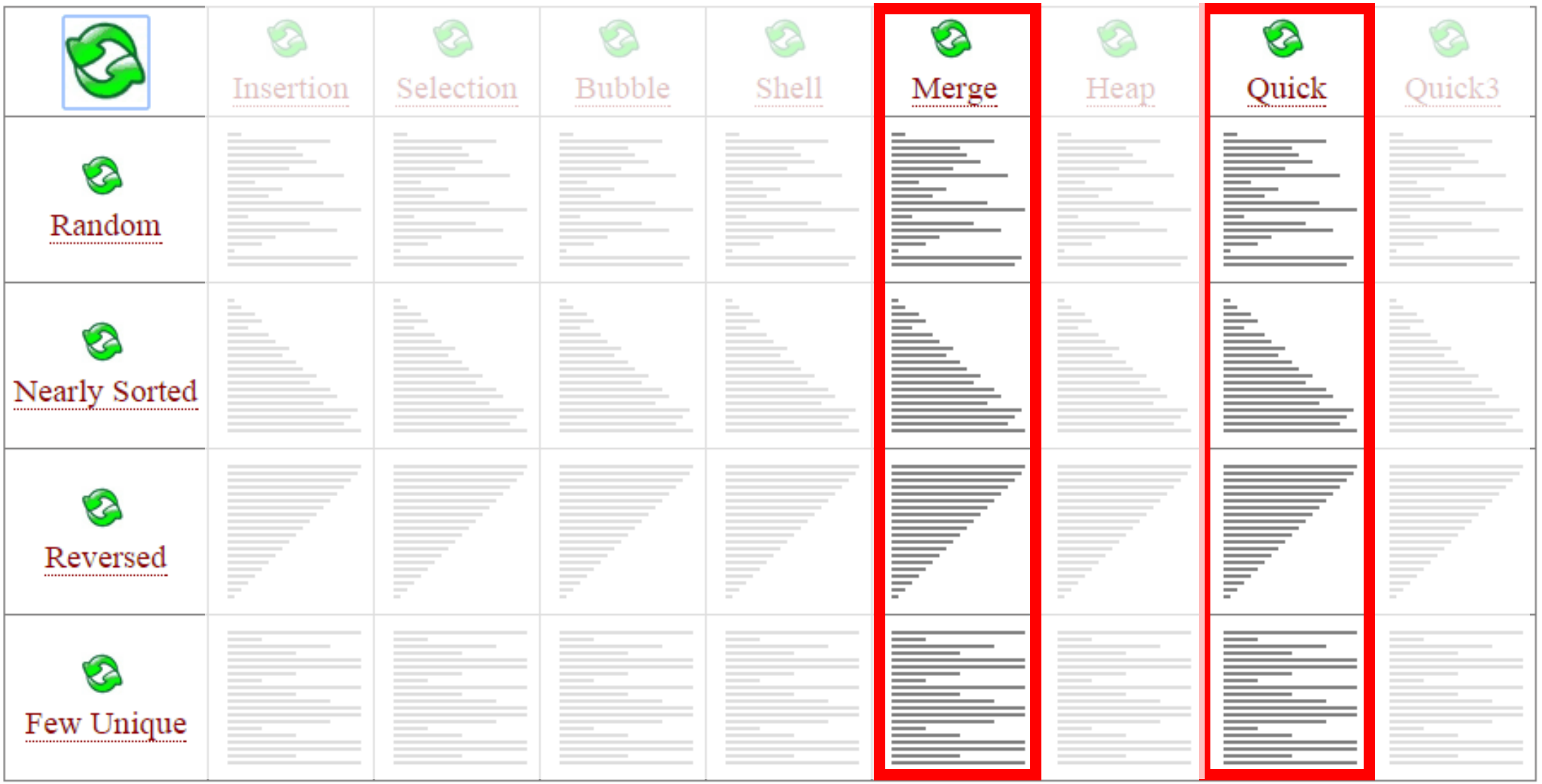
Lecture 7: Recursive Sorting Algorithms

CS32: Object Oriented Design and Implementation

Nabeel Nasir, PhD
UC Santa Barbara

October 20th 2025

Content based on slides from Ziad Matni



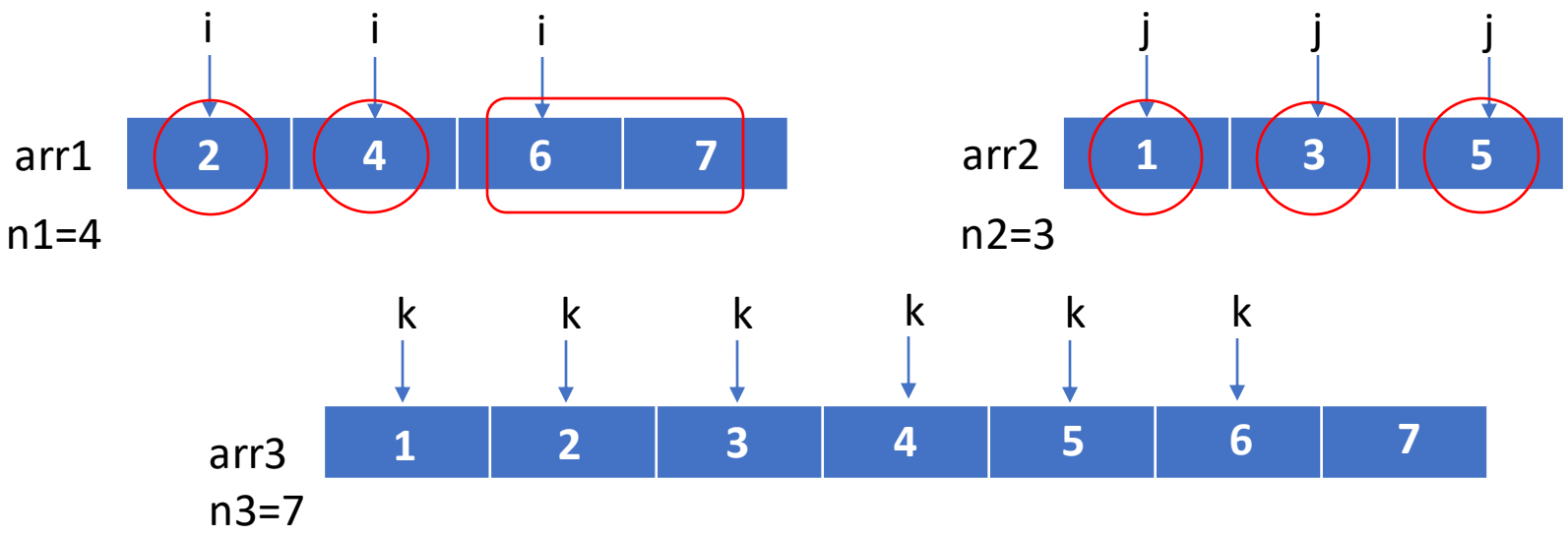
https://www.toptal.com/developers/sorting-algorithms

“Divide and Conquer” Approach

- The idea is to subdivide a **larger** problem into **smaller parts**
- Solve each smaller part by itself
- Combine solutions of smaller sub-problems back into the larger problem
- These are often **recursive solutions**

Merge Sort

Idea #1: Merging two sorted arrays to make a combined sorted array



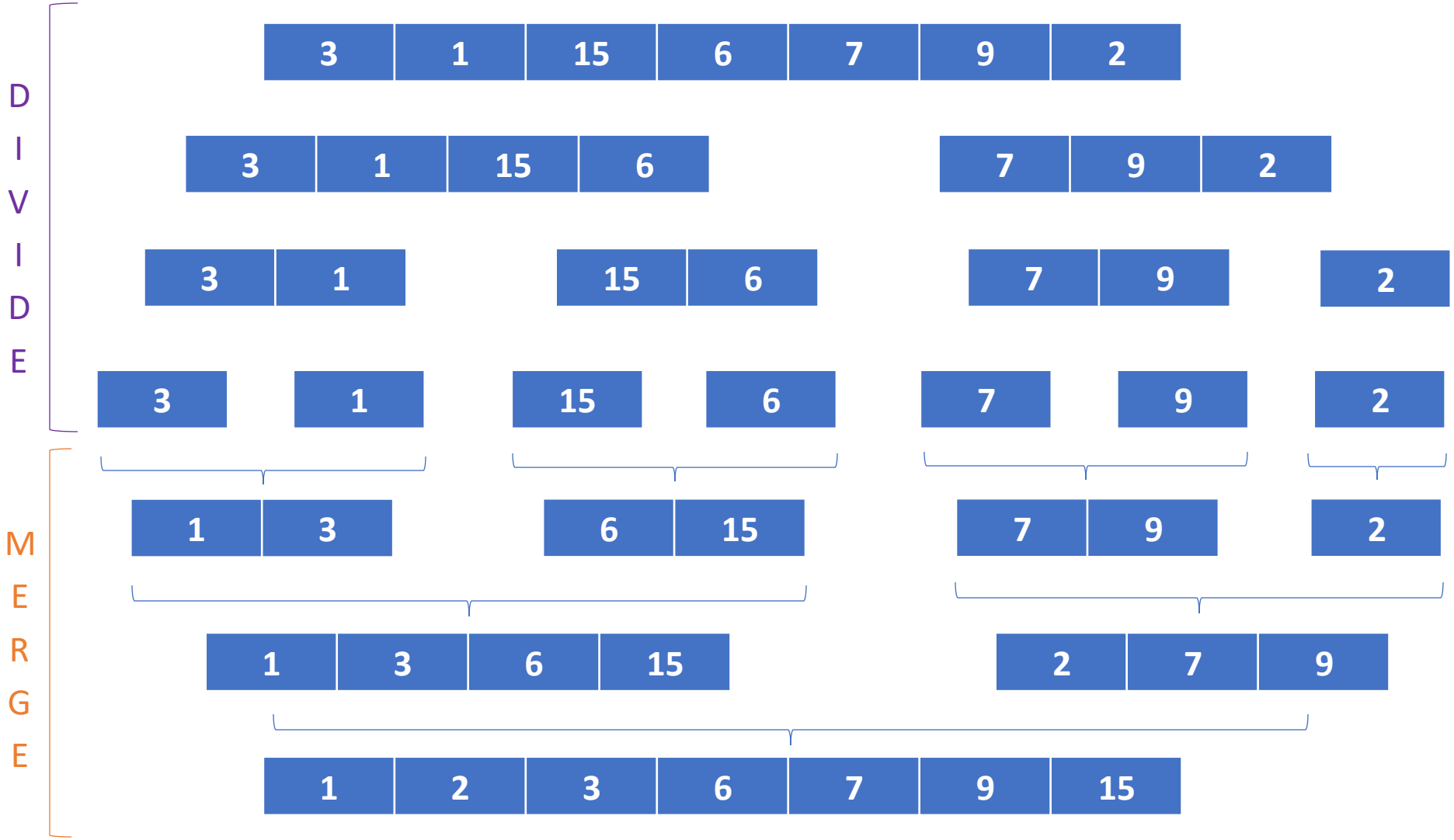
while (`i < n1`) and (`j < n2`):
- Compare and copy
- Increment i or j

if j had finished first:
while (`i < n1`)
- Do rest of the copy without comparison

if i had finished first:
while (`j < n2`)
- Do rest of the copy without comparison

How can this merge idea be used to sort a single unordered array?

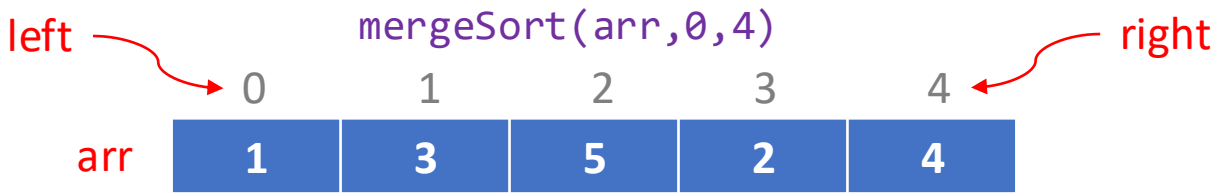
Idea #2: Keep dividing single unordered array into halves, merge them back sorted!

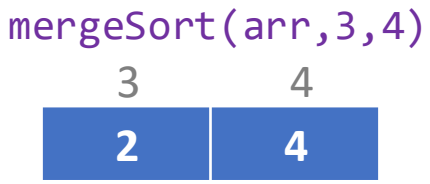


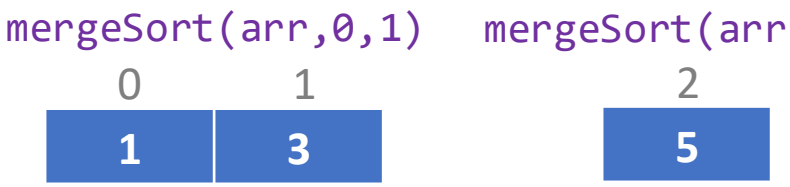
- Our merge idea only works when we have two sorted arrays…! What do we do?
- Keep dividing into halves until you have one element pieces
- A one element array is sorted (duh!)
- Now we can apply our merge skills!
  - Combine adjacent one element pieces into a sorted two element piece
  - Repeatedly merge to sort full array!

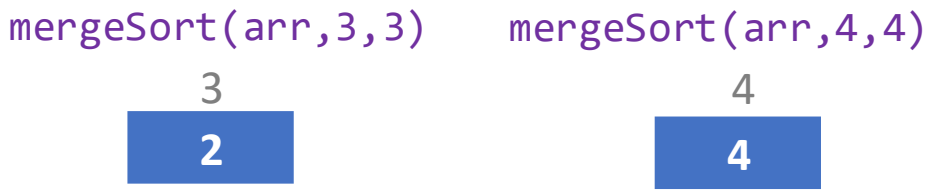
The Divide step is achieved with recursive calls

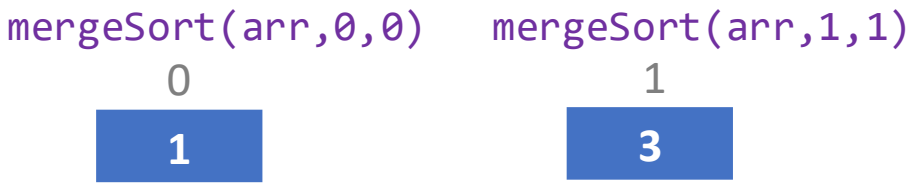
```
void mergeSort(int[] arr, int left, int right)
```











Activity1: Fix mergeSort() function to recursively divide the array

- Setup (3mins)
  - Download zip file from Canvas
  - Open `ms.cpp`
  - Pair up with your neighbor!

Activity (5mins):

- Goal: Fix implementation for recursive function `mergeSort()` in `ms.cpp`
- Do steps 1, 2, and 3 listed in `ms.cpp`

If Canvas is down:
nabeel-nasir.github.io/cs32/activities/lec07-files.zip

Expected output:

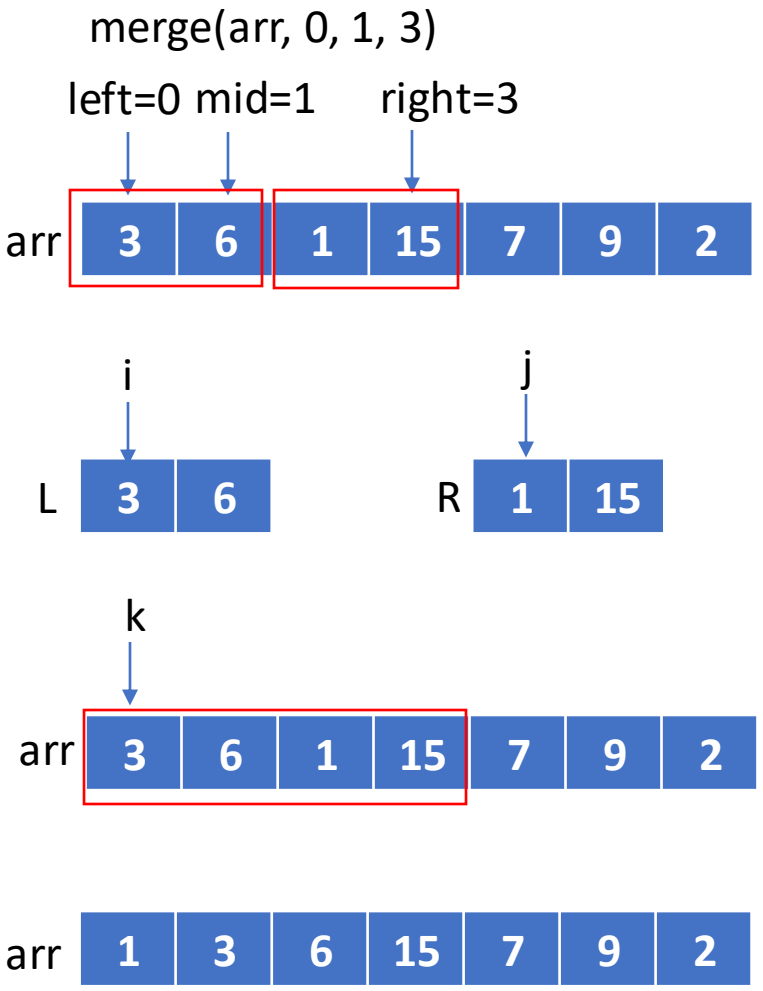
- `make dividePrint` should print the following:

```
arr={1, 3, 5, 2, 4}, left=0, right=4
mergeSort(arr, 0, 4)
mergeSort(arr, 0, 2)
mergeSort(arr, 0, 1)
mergeSort(arr, 0, 0)
mergeSort(arr, 1, 1)
mergeSort(arr, 2, 2)
mergeSort(arr, 3, 4)
mergeSort(arr, 3, 3)
mergeSort(arr, 4, 4)
```

merge step involves comparing and copying

```
void merge(int arr[], int left, int mid, int right)
```

- Steps for `merge(arr, left, mid, right)`
  - Consider elements in `arr` from `left` to `right`
  - Assume two halves
    - 1st half: `left` to `mid` (included)
    - 2nd half: `mid` (excluded) to `right`
  - Assume each half is sorted
  - Copy these two halves to temporary arrays
  - Merge these halves back into `arr` in sorted order



merge() only modifies the range [left, right]

Activity2: Fix Merge Sort merge() function

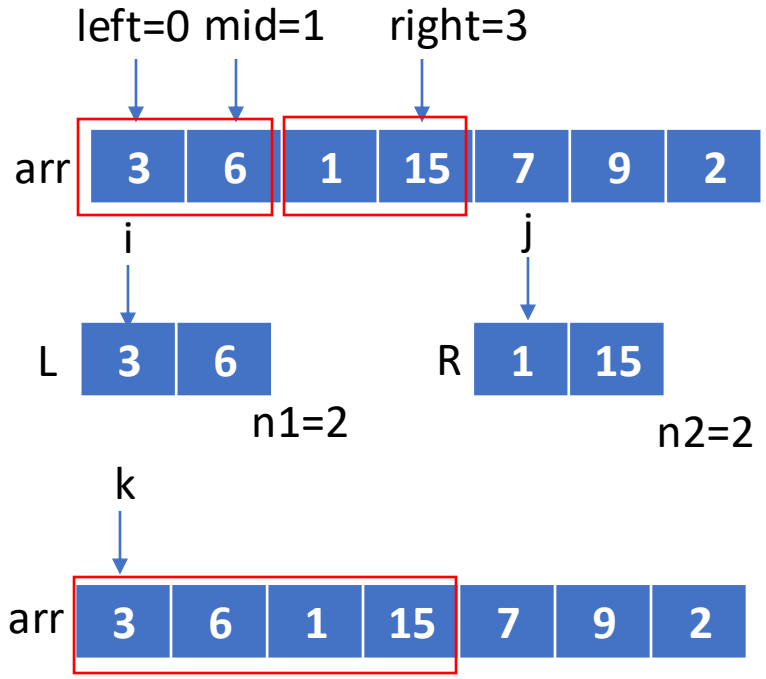
- Setup (1min)
  - Download zip file from Canvas
  - Open `ms.cpp`
  - Pair up with your neighbor!

Activity (7mins):

- Goal: Fix the implementation for `merge()` in `ms.cpp`
- Do steps 4, 5, 6, 7, and 8 listed in `ms.cpp`

- To test: `make mergeTest`
  - Can you try to pass all the tests?

If Canvas is down:
nabeel-nasir.github.io/cs32/activities/lec07-files.zip

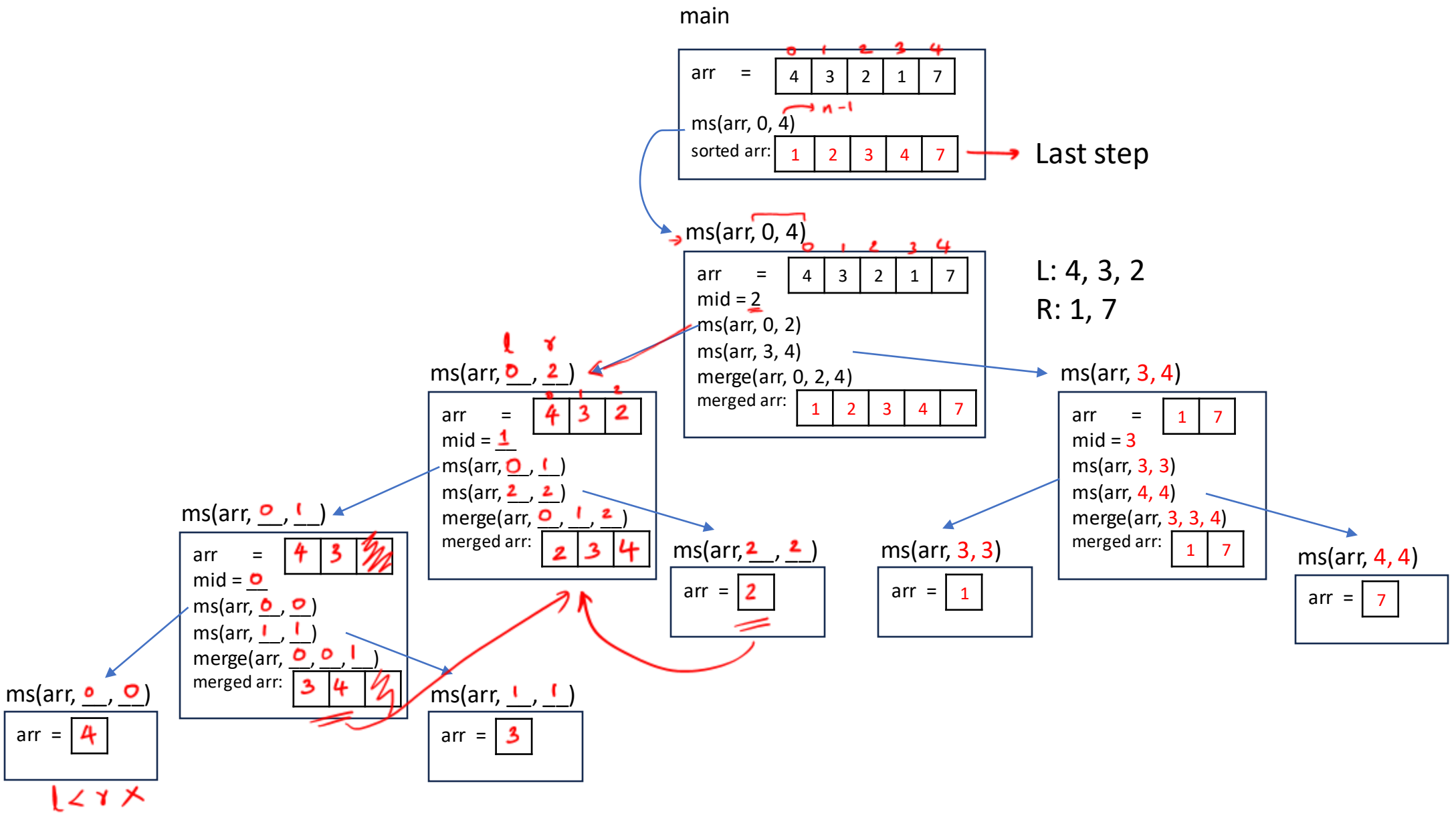


Hint to do the merge:

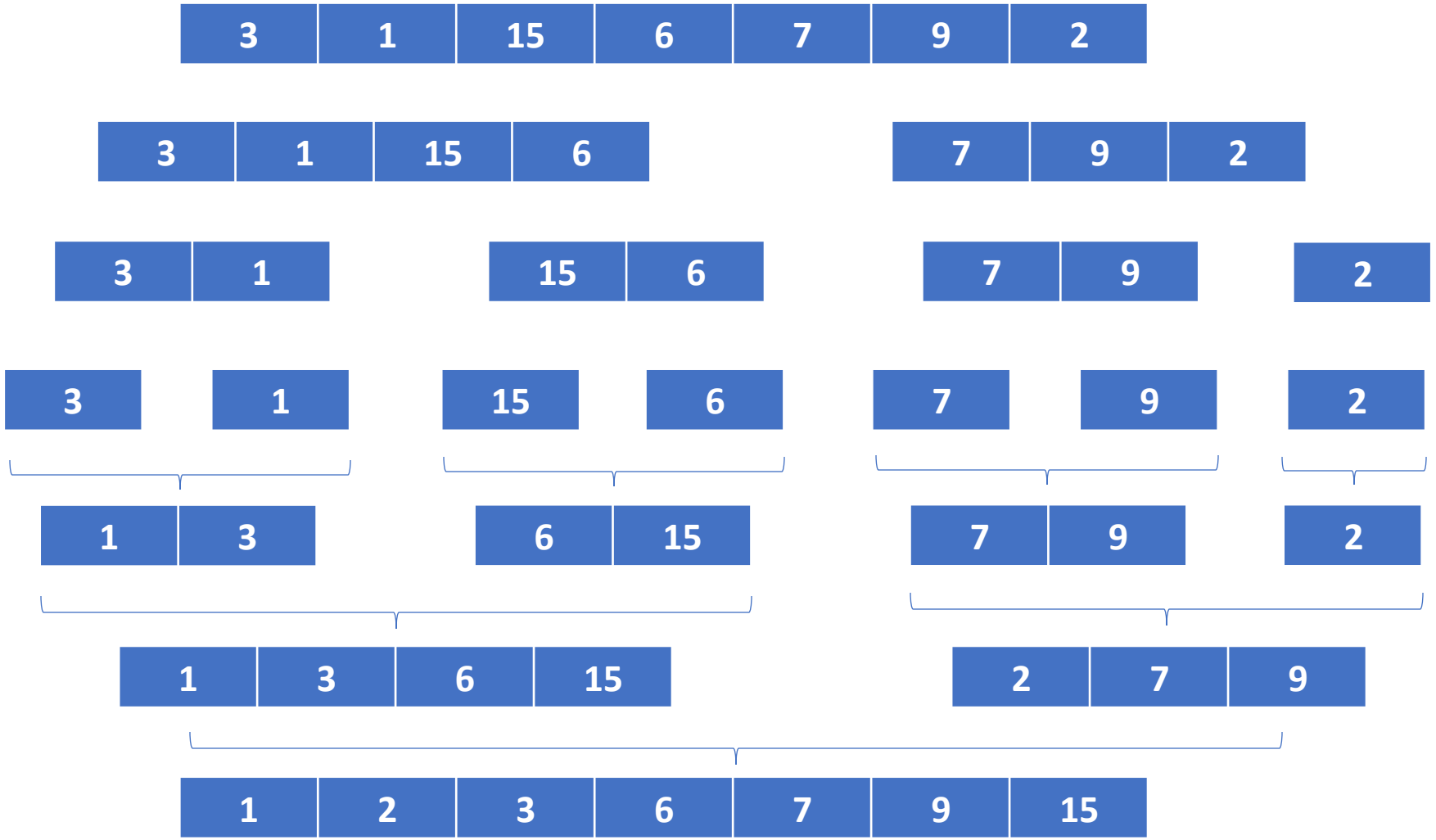
while (`i < n1`) and (`j < n2`):
- Compare and copy to `arr[k]`

if j had finished first:
while (`i < n1`)
- Do rest of the copy without comparison

if i had finished first:
while (`j < n2`)
- Do rest of the copy without comparison


main
arr = 4 3 2 1 7
ms(arr, 0, 4)
sorted arr: 1 2 3 4 7
Last step
ms(arr, 0, 4)
arr = 4 3 2 1 7
mid = 2
ms(arr, 0, 2)
ms(arr, 3, 4)
merge(arr, 0, 2, 4)
merged arr: 1 2 3 4 7
L: 4, 3, 2
R: 1, 7
ms(arr, 0, 2)
arr = 4 3 2
mid = 1
ms(arr, 0, 1)
ms(arr, 2, 2)
merge(arr, 0, 1, 2)
merged arr: 2 3 4
ms(arr, 3, 4)
arr = 1 7
mid = 3
ms(arr, 3, 3)
ms(arr, 4, 4)
merge(arr, 3, 3, 4)
merged arr: 1 7
ms(arr, 0, 1)
arr = 4 3
mid = 0
ms(arr, 0, 0)
ms(arr, 1, 1)
merge(arr, 0, 0, 1)
merged arr: 3 4
ms(arr, 2, 2)
arr = 2
ms(arr, 3, 3)
arr = 1
ms(arr, 4, 4)
arr = 7
ms(arr, 0, 0)
arr = 4
ms(arr, 1, 1)
arr = 3

Merge sort Analysis



- **Divide**: What's the depth of the tree you create for dividing the full array to single element arrays?
  - $O(\log_2 n)$
- **Merge**: When you're merging two arrays of size n1 and n2
  - Each element from both arrays looked at once and added to output exactly once
  - O(n1 + n2)
  - For merge sort n1 ≈ n2
- At each level of the tree, you do O(n) comparisons
- So complexity turns out to be O(n logn)

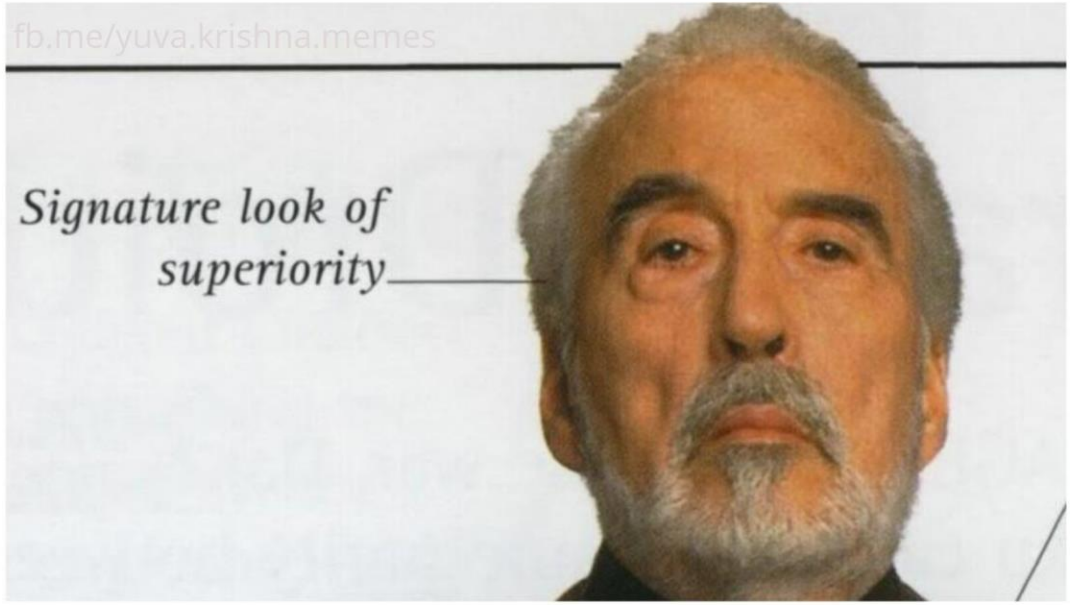
Merge sort Analysis

- Best-case: ***O*(n log n)**
- Average-case: ***O*(n log n)**
- Worst-case: ***O*(n log n)**

- It does require additional space (the **`temp`** array), so ***O*(n)** additional space, to merge the little arrays into a sorted one

Break!

When your array has only 6 elements, but sorted them with quick sort ..



Questions?

Join at

slido.com

#2265 987

After the break:

- Quick Sort

Quick Sort

Idea: Partition items based on a "pivot" item

4 10 8 7 6 5 3 12 14 2

- Partitioning means:
  - Pick a pivot
  - Move items ≤ pivot to its left
  - Move items > pivot to its right
- After each partition:
  - Pivot fixed in its "correct" position, as it would appear in the final sorted array
  - Items to left of pivot and right of pivot may still be **unordered**
- To fully sort array:
  - Keep partitioning until you have all pivots fixed!

final sorted array 2 3 4 5 6 7 8 10 12 14

Partitioning can be done in many ways

- Can be variations in partitioning algorithms
  - Pivot can be any item in the array: first, middle, median, last item,…
  - Partitioning itself can be achieved in different ways
    - How do you iterate through the array?
    - How do you make space for the pivot to be eventually placed?
- We will focus on a specific type of partitioning called Lomuto Partition
  - The **last item in the array** is always chosen the **Pivot**

How does Lomuto Partition partition an array?

7 1 6 2 3

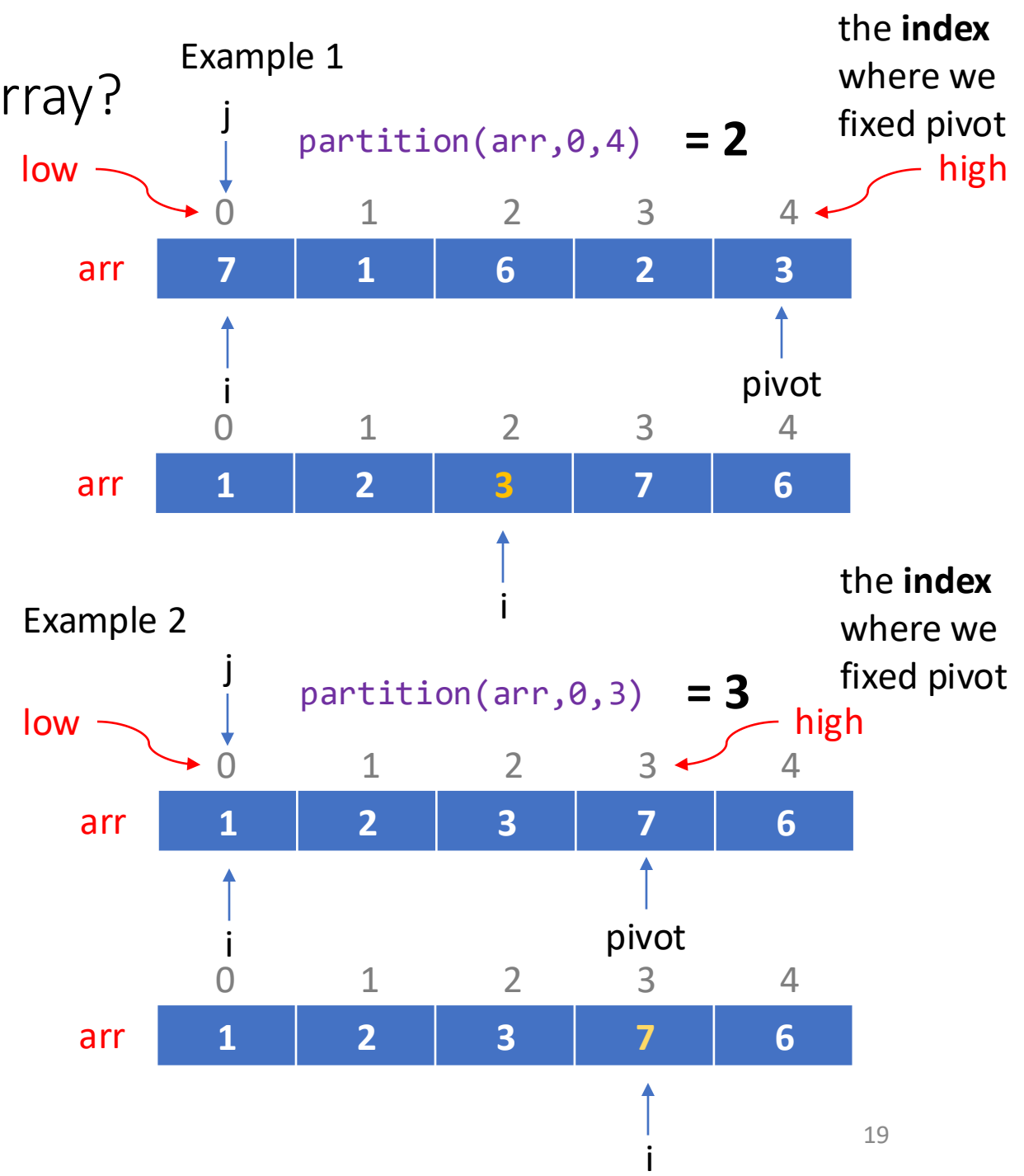
- Pivot = last element
- i: tracks boundary between elements smaller than pivot and rest
  - Or “what’s the ideal position to finally place the pivot?”
- j: “explorer”
  - Scans from first element to last element (everything except pivot)
  - Compares each element with pivot
  - If `a[j] <= pivot`, swap `a[j]` and `a[i]`
  - Move j forward regardless of swap
- At the end of partition, swap `a[i]` and `pivot`

How does Lomuto Partition partition an array?

```
int partition(int arr[], int low, int high)
```

Steps:

- `pivot` = `a[high]`
- Move `j` from `low` to `high`
- Compare `a[j]` with `pivot`
  - If `a[j] <= pivot`
    - Swap `a[j]` and `a[i]`
    - Increment `i`
  - Increment `j`
- At the end of partition, swap `a[i]` and `pivot`



Activity: Fix Quick Sort `partition()` function

- Setup (1min)
  - Download zip file for today's lecture from Canvas.
    - `qs.cpp`: has the incorrect implementation for `partition()`
  - Pair up with your neighbor!

Activity (7mins):

- Goal: Fix the implementation for `partition()` in `qs.cpp`
- Do steps 1, 2, 3, and 4 in `partition()`
- To test: `make partitionTest`
  - Can you pass all the tests?

Activity: Fix quickSort() function

- Setup (2mins)
  - Download zip file for today's lecture from Canvas.
    - `qs.cpp`: has the incorrect implementation for `quickSort()`
  - Pair up with your neighbor!

Activity (5mins):

- Goal: Fix the recursive function `quickSort()` in `qs.cpp`
- Do steps 5, and 6 in `quickSort()`

  - What does `void quickSort(int[] arr, int low, int high)` do?
    - Partition array and get pivot's index
    - Do quick sort for left part of the partition
    - Do quick sort for right part of the partition
    - Stop when you have just 1 element

- To test: `make qsTest`

Quicksort Analysis

- In the best case and average case:
  - Partitioning the array into two roughly equal halves
  - Partitioning does O(n) comparisons
  - Depth of the tree created will be of O(logn) size
  - Total comparisons needed is O(logn) x O(n) = O(nlogn)
    - Once you partition into two parts, elements in left part never have to be compared to the right part, or to the rest of the array

Quicksort Analysis

- In the worst case:
  - Happens when array already sorted
  - The partition is uneven (not an equal split)
  - At each step, pivot needs to be compared with all other elements to the left
  - Comparisons: n-1 + n-2 + … + 1. Time complexity = $O(n^2)$
  - Space complexity = O(n)
    - Skewed tree!

Quicksort Analysis

- Best-case: **O(n log n)**
- Average-case: **O(n log n)**
- Worst-case: **O(n^2)**

- Worst case: already sorted array
  - At each partition, you will have to do O(n) comparisons => O(n^2)

- Unlike (our version of) Mergesort, **Quicksort** does *not* require additional buffer space and can sort the array in-place.
  - But you need to think about the stack frames in the recursive calls
  - Space complexity = O(logn)

Thank you!