CS2040S DATA STRUCTURES & ALGORITHMS

DG1 – WEEK 3

WEINENG@U.NUS.EDU

CS2040S DATA STRUCTURES & ALGORITHMS

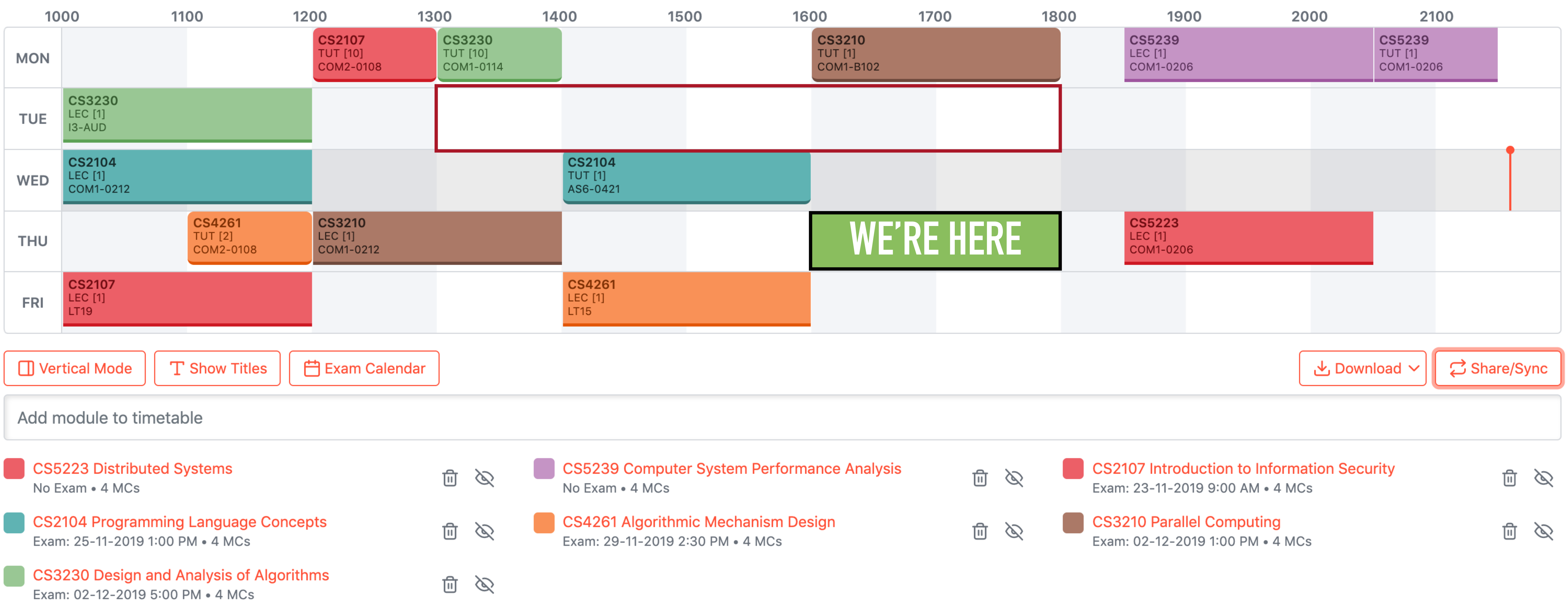
ADMINISTRATIVE MATTERS

WEINENG@U.NUS.EDU

ABOUT ME!

- Ang Wei Neng
- Year 3 Computer Science
- weineng@u.nus.edu

NEED HELP?


1000
1100
1200
1300
1400
1500
1600
1700
1800
1900
2000
2100
MON
CS2107 TUT [10] COM2-0108
CS3230 TUT [10] COM1-0114
CS3210 TUT [1] COM1-B102
CS5239 LEC [1] COM1-0206
CS5239 TUT [1] COM1-0206
TUE
CS3230 LEC [1] I3-AUD
WED
CS2104 LEC [1] COM1-0212
CS2104 TUT [1] AS6-0421
THU
CS4261 TUT [2] COM2-0108
CS3210 LEC [1] COM1-0212
WE'RE HERE
CS5223 LEC [1] COM1-0206
FRI
CS2107 LEC [1] LT19
CS4261 LEC [1] LT15
Vertical Mode
Show Titles
Exam Calendar
Download
Share/Sync
Add module to timetable
CS5223 Distributed Systems
No Exam • 4 MCs
CS5239 Computer System Performance Analysis
No Exam • 4 MCs
CS2107 Introduction to Information Security
Exam: 23-11-2019 9:00 AM • 4 MCs
CS2104 Programming Language Concepts
Exam: 25-11-2019 1:00 PM • 4 MCs
CS4261 Algorithmic Mechanism Design
Exam: 29-11-2019 2:30 PM • 4 MCs
CS3210 Parallel Computing
Exam: 02-12-2019 1:00 PM • 4 MCs
CS3230 Design and Analysis of Algorithms
Exam: 02-12-2019 5:00 PM • 4 MCs

SCHEDULE (DISCUSSION GROUPS)

Week	Topic	Week	Topic
1	-	8	Graphs
2	-	9	SSSP
3	Sort + Binary Search	10	Bridge Checker
4	Heaps	11	TSP
5	Red-black Trees	12	Kattis Problems
6	B-Trees	13	Revisions
7	Hashing		

STRUCTURE OF DISCUSSION GROUPS (MAY VARY)

- Summary of relevant content for the week
- Some fun additional topics that will not be tested
- Discussion sheets (if available) and
- Kattis practices
- Q & A regarding problem set

PROBLEM SETS

- Feel free to ask me questions relating to your problem set via **email**.
- Often more effective as I might not be able to “produce” the most optimal solution or spot your bug immediately.

DG THIS WEEK

- Recap of common sorting algorithms
- Sorting and binary search problems [DG 1 Sheet]

FUN VIDEO

15 SORTING ALGORITHMS IN 6 MINUTES

CS2040S DATA STRUCTURES & ALGORITHMS

QUICKSORT

WEINENG@U.NUS.EDU

QUICK SORT – OVERVIEW?

- **Can be In-place**: Only need a constant additional memory space for intermediate operations.
  - Operates on the input array only through repeated swaps of pairs of elements.
- Aesthetically pleasant?

QUICK SORT – SOLVING THE PROBLEM OF SORTING

- **Input:** An array of *n* numbers, in arbitrary order
- **Output:** An array of the same numbers, sorted from smallest to largest.

3	8	2	5	1	4	7	6

1	2	3	4	5	6	7	8

QUICK SORT – THE IDEA

- "partial sorting" around a "pivot element"

Step 1: Choose a pivot element

3	8	2	5	1	4	7	6

Pivot Element

Step 2: Rearrange the input array around the pivot

2	1	3	6	7	4	5	8

Less than pivot

Pivot Element

Greater than pivot

QUICKSORT – THE PARTITION ALGORITHM

- O(n) - Linear time! *[Just the Partitioning]*
- Can be implemented in-place
- **Significant progress towards sorting**:
  - Pivot element winds up in its rightful position (i.e. same position as in the sorted version)
  - Reduces the problem into two smaller sorting problems (less than pivot, more than pivot) – recursively sort!

QUICKSORT – HIGH LEVEL IDEA

- Input: array A of *n* distinct integers.
- Post-condition: elements of A are sorted from smallest to largest.

if *n* ≤ 1, **then** return // base case - already sorted
choose a pivot element *p* // to be implemented
partition *A* around *p* // to be implemented
recursively sort first part of *A* (less than *p*)
recursively sort second part of *A* (more than *p*)

2	1	3	6	7	4	5	8

< p *p* > p

PARTITIONING AROUND A PIVOT ELEMENT – THE BAD EXAMPLE

- The easy way out, but takes **O(n)** space!
- Do a single scan over the input array, and copy over its non-pivot elements one by one into a new array of same length, populating from the front (< p) and back (> p). The pivot element can be copied into B after everything is in.

BAD!

PARTITIONING AROUND A PIVOT ELEMENT – THE RIGHT EXAMPLE

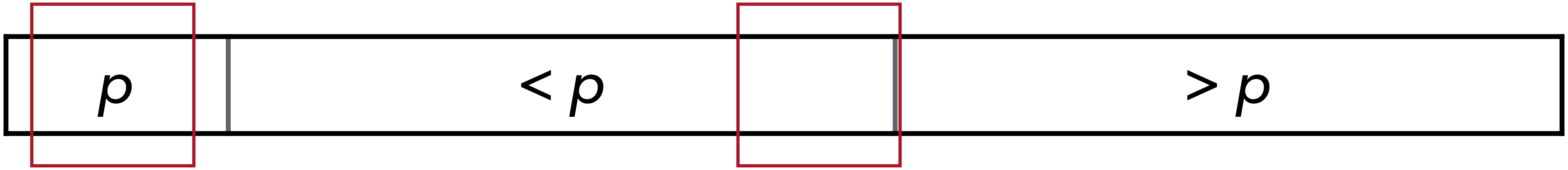
- Doing it in-place, and takes **O(1)** space!
- Do a single scan through the array, swapping pairs of elements as needed so that the array is properly partitioned by the end of the pass.
- Take the first element to always be the pivot (O(1)).
- As we scan and transform the input array, we will take care to ensure it always has the form:

p	$< p$	$> p$	?

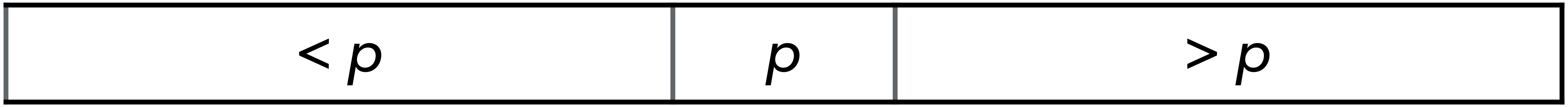
Already partitioned **Unpartitioned**

PARTITIONING AROUND A PIVOT ELEMENT – THE RIGHT EXAMPLE

- And once we're done, we'll get:



- To complete the partitioning, we can swap the pivot element (*p*) with the last element less than it:



PARTITIONING AROUND A PIVOT ELEMENT - EXAMPLE

- All elements between the pivot and *i* are less than the pivot, and all elements between *i* and *j* are greater than the pivot.

Pivot **Unpartitioned**

3	8	2	5	1	4	7	6

i j

Pivot **Partitioned** **Unpartitioned**

3	8	2	5	1	4	7	6

i j

PARTITIONING AROUND A PIVOT ELEMENT – EXAMPLE

- All elements between the pivot and *i* are less than the pivot, and all elements between *i* and *j* are greater than the pivot.

Pivot **Partitioned** **Unpartitioned**

3	8	**2**	5	1	4	7	6

i j

1. Swap 8 and 2
2. Increment *i*

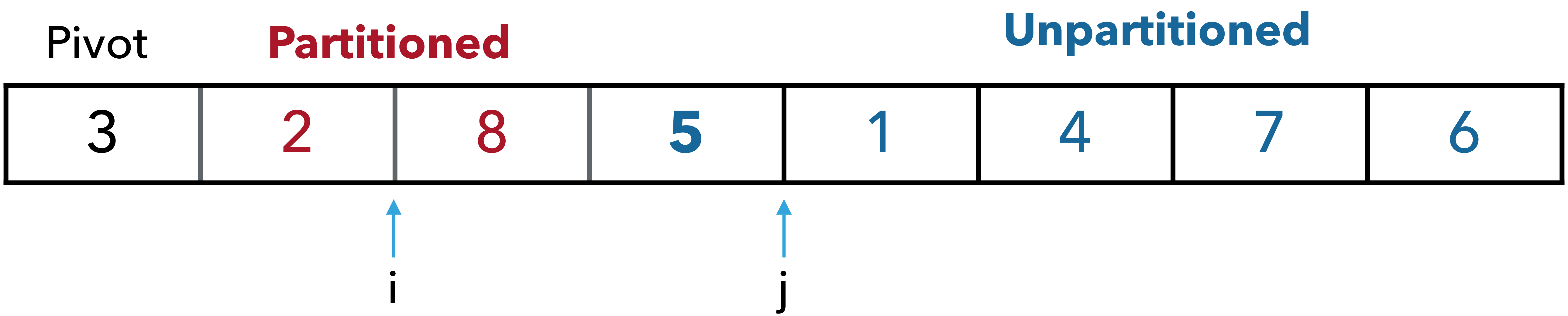
Pivot **Partitioned** **Unpartitioned**

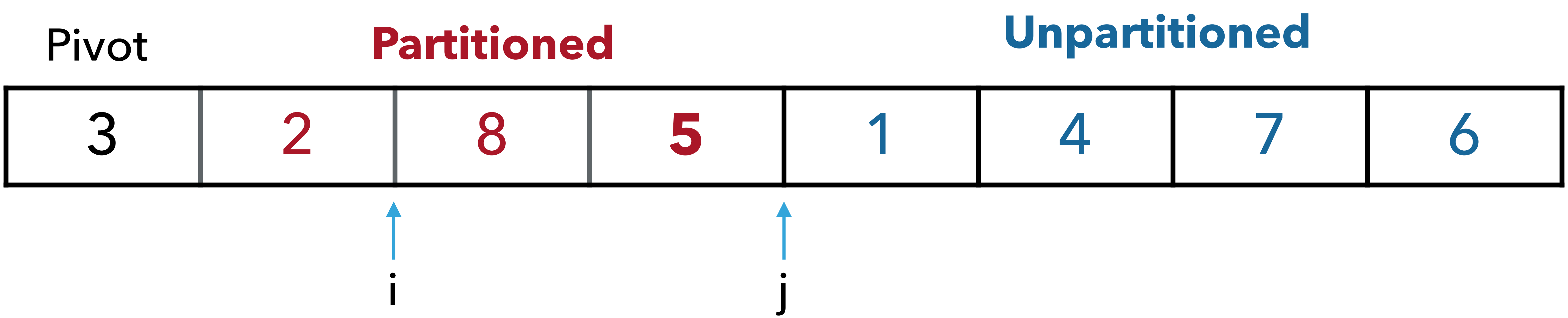
3	**2**	8	5	1	4	7	6

i j

PARTITIONING AROUND A PIVOT ELEMENT – EXAMPLE

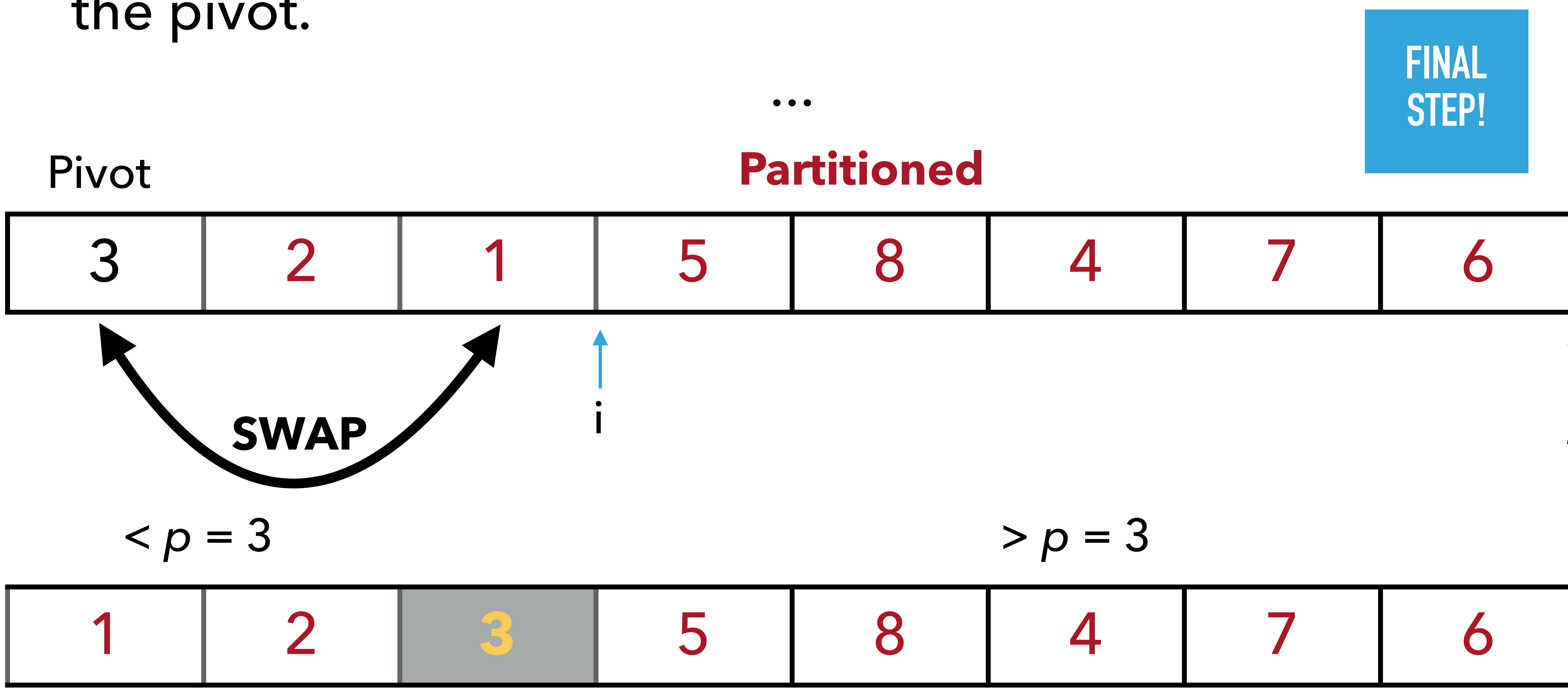
- All elements between the pivot and *i* are less than the pivot, and all elements between *i* and *j* are greater than the pivot.





PARTITIONING AROUND A PIVOT ELEMENT - EXAMPLE

- All elements between the pivot and *i* are less than the pivot, and all elements between *i* and *j* are greater than the pivot.



THE PARTITION ALGORITHM – JAVA CODE

```
public static int partition (int a[], int s, int e) {
    int p = a[s]; // p is the pivot, the sth item
    int i = s;    // Initially S1 and S2 are empty
    for (int j = s+1; j <= e; j++) {
        if (a[j] < p) { // case 2: put a[k] to S1
            swap (a,j,i);
            i++;
        } else { // case 1: put a[k] to S2! Do nothing!
        }
    }
    swap (a,s,i);  // put the pivot at the right place
    return i;      // i is the pivot final position
}
```

CS2040S DATA STRUCTURES & ALGORITHMS

SORTING

WEINENG@U.NUS.EDU

BUBBLE SORT – IMPLEMENTATION

```
public static void bubbleSort (int[] a) {

    for (int i = 1; i < a.length; i++) {
        for (int j = 0; j < a.length-i; j++) {
            if (a[j] > a[j+1]) {
                int temp = a[j];   The larger item bubbles down (swap)
                a[j] = a[j+1];
                a[j+1] = temp;
            }
        }
    }
}
```

BUBBLE SORT – ANALYSIS

- Guaranteed inefficiency: does not make an effort to check whether the input is sorted or not.
- Worse case: **$O(n^2)$**
- Best case: **$O(n^2)$**

Use the IMPROVED BUBBLE SORT!

- Have a flag `is_sorted`.
- Improved Best case: **$O(n)$**

IMPROVED BUBBLE SORT – IMPLEMENTATION

```
public static void bubbleSortImproved(int[] a) {
    for (int i = 1; i < a.length; i++) {
        boolean is_sorted = true;

        for (int j = 0; j < a.length-i; j++) {
            if (a[j] > a[j+1]) {
                int temp = a[j];
                a[j] = a[j+1];
                a[j+1] = temp;
                is_sorted = false;

            }
        }

        if (is_sorted) return;
    }

}
```

is_sorted = true if a[] is sorted

The larger item bubbles up, and is_sorted is set to false (i.e. the data was not sorted)

INSERTION SORT – IMPLEMENTATION

```
public static void insertionSort(int[] a) {
    for (int i = 1; i < n; i++) {

        int next = a[i];

        int j;

        for (j=i-1; j>=0 && a[j]>next; j--)
            a[j+1] = a[j];
        a[j+1] = next;
        }
    }
}
```

a[i] is the next data to insert

Scan backwards to find a place: why?

Now insert the value next after index j at the end of the loop

INSERTION SORT - ANALYSIS

- Worse case: **$O(n^2)$**
- Best case: **$O(n)$**
- Works best when array is almost sorted.

SELECTION SORT – IMPLEMENTATION

```
public static void selectionSort (int[] a) {

    for (int i = a.length - 1; i >= 1; i--) {
        int index = i;

        for (int j = 0; j < i; j++) {
            if (a[j] > a[index])
                index = j;
        }
        int temp = a[index];
        a[index] = a[i];
        a[i] = temp;
    }

}
```

i is the last item position, index is the largest element position

Loop to get the largest element

j is the current largest element

Swap the largest item a[index] with the last item a[i]

SELECTION SORT - ANALYSIS

- Worse case: **$O(n^2)$**
- Best case: **$O(n^2)$**

IN-PLACE SORT

- In-place sorting algorithm sorts the list **only by modifying** the order of the elements **within the list**.
  - Selection sort? **Yes**
  - Insertion sort? **Yes**
  - Bubble sort? **Yes**
  - Merge sort? **No**
  - Quick sort? **Yes**

STABLE SORT

- A sorting algorithm is stable if **two objects with equal keys appear in the same order in sorted output** as they appear in the input array to be sorted.
  - Selection sort? **No**
  - Insertion sort? **Yes**
  - Bubble sort? **Yes**
  - Merge sort? **Yes**
  - Quick sort? **No**

SUMMARY OF SORTING ALGORITHMS

Type	Worst	Best	In-Place?	Stable?
Selection	$O(n^2)$	$O(n^2)$	Yes	No
Insertion	$O(n^2)$	$O(n)$	Yes	Yes
Bubble	$O(n^2)$	$O(n^2)$	Yes	Yes
Bubble*	$O(n^2)$	$O(n)$	Yes	Yes
Merge	$O(nlogn)$	$O(nlogn)$	No	Yes
Quick	$O(n^2)$	$O(nlogn)$	Yes	No

PROBLEMS?!