# Section 4

CSCI E-22

Will Begin Shortly

## Quicksort

- Recursive, divide & conquer
- Achieves better average-case time complexity than O($n^2$)
- Elements are partitioned into two *subarrays*
  - Elements in left subarray are *less than or equal to* elements in right subarray
- To partition, pick a pivot value, then repeatedly swap elements between subarrays to satisfy above condition
- After partitioning is done, recursively quicksort the subarrays
- Say we want to call `partition()` on the array below. What would be the pivot value?

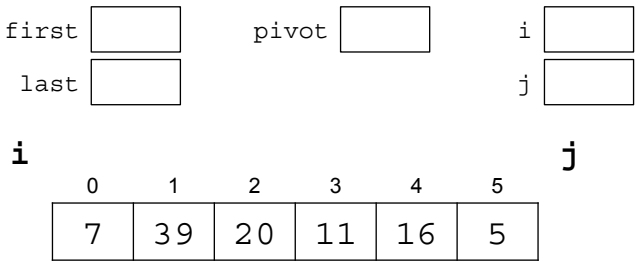
| 7 | 39 | 20 | 11 | 16 | 5 |
|---|---|---|---|---|---|

```java
public static int partition(
  int[] arr, int first, int last
) {
    int pivot = arr[(first + last)/2];
    int i = first - 1;
    int j = last + 1;

    while (true) {
        do {
            i++;
        } while (arr[i] < pivot);

        do {
            j--;
        } while (arr[j] > pivot);

        if (i < j) {
            swap(arr, i, j);
        } else {
            return j;
        }
    }
}
```




# Quicksort

- What is the time complexity of quicksort
  - in the best case?

# Quicksort

- What is the time complexity of quicksort
  - in the worst case?

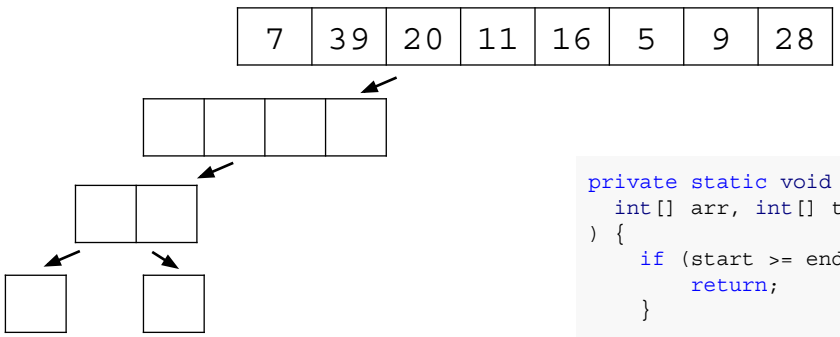
# Quicksort

- What is the time complexity of quicksort
  - in the average case?

- How would you characterize the performance of quicksort in the example we just stepped through? Was it an example of best case, worst case, or average case performance? Why?

# Merge sort

- Like quicksort, recursive divide & conquer
- Unlike quicksort, merge sort does not modify the array during the “division” phase of the algorithm
- Instead, sorting is done as subarrays are merged together into a fully sorted subarray
- `merge()` method takes two already sorted subarrays and merges them into a sorted whole

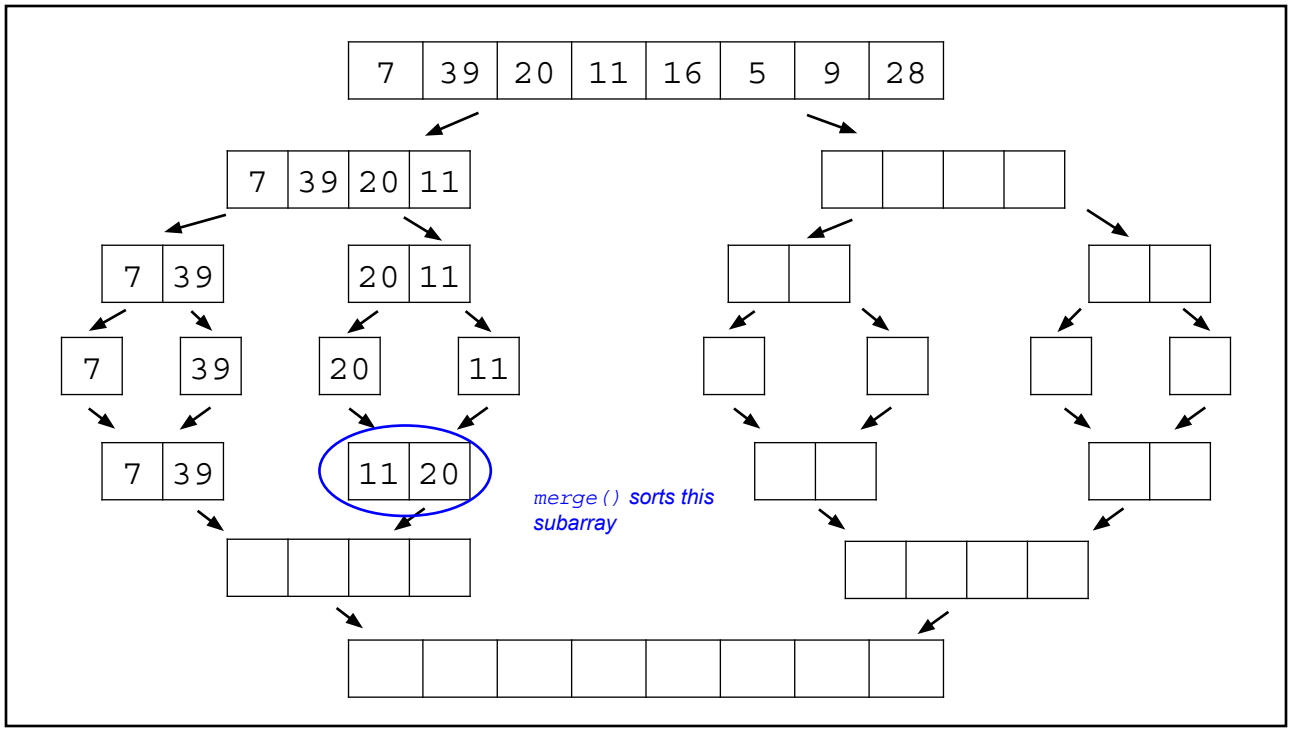


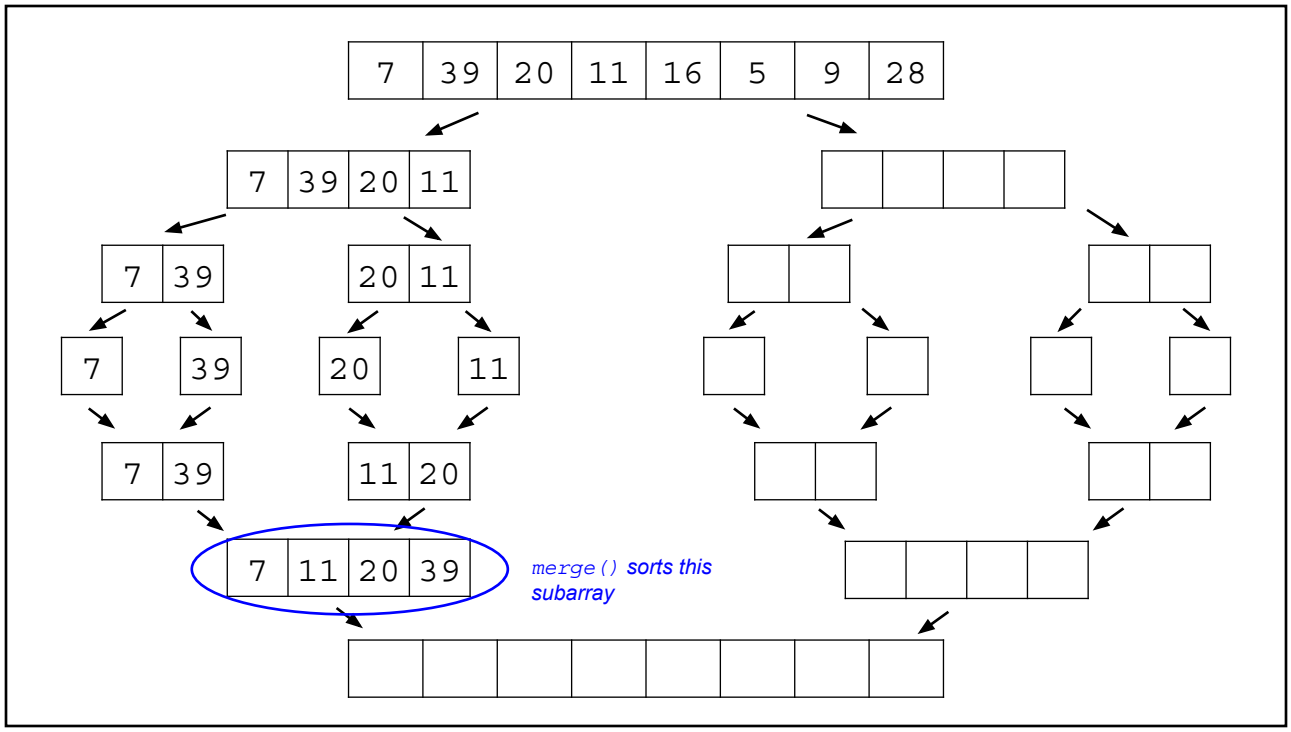

in the recursive case, `mSort()` finds the middle position of the array and makes two recursive calls (left, then right) before merging anything

```
mSort(0, 7)
```

```
private static void mSort(
  int[] arr, int[] tmp, int start, int end
) {
    if (start >= end) {
        return;
    }

    int middle = (start + end)/2;
    mSort(arr, tmp, start, middle);
    mSort(arr, tmp, middle + 1, end);
    merge(
      arr,
      tmp,
      start,
      middle,
      middle + 1,
      end
    );
}
```


7 39 20 11 16 5 9 28
7 39 20 11
7 39
20 11
7
39
20
11
7 39
11 20
merge() sorts this subarray


7 39 20 11 16 5 9 28
7 39 20 11
7 39
20 11
7
39
20
11
7 39
11 20
7 11 20 39
merge() sorts this subarray

## Merge sort

- What major **advantage** does merge sort have over quicksort with respect to **time** complexity?

- What major **disadvantage** does merge sort have compared to quicksort with respect to **space** complexity?

## Radix sort

- Stable, distributive sorting algorithm
- Can be used to sort integers, strings, complex data
- For integers, the algorithm processes individual digits of each element
  - For each element, place it into a “bucket” according to the value of its least significant digit, **maintaining order** in the bucket
  - When you reach the end of the array, repeat the process for the next most significant digit
  - Stop when all elements have been evaluated according to the most significant position of the largest element

| 41 | 326 | 18 | 1 | 117 | 56 | 86 | 7 | 14 | 221 | 19 | 30 |
|---|---|---|---|---|---|---|---|---|---|---|---|

first pass (buckets for the ones digit)

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | |


| 41 | 326 | 18 | 1 | 117 | 56 | 86 | 7 | 14 | 221 | 19 | 30 |
|---|---|---|---|---|---|---|---|---|---|---|---|

first pass (buckets for the ones digit)

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | |


second pass (buckets for the tens digit)

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | |

| 41 | 326 | 18 | 1 | 117 | 56 | 86 | 7 | 14 | 221 | 19 | 30 |
|---|---|---|---|---|---|---|---|---|---|---|---|

first pass (buckets for the ones digit)

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | |

second pass (buckets for the tens digit)

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | |

third pass (buckets for the hundreds digit)

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | |

# Radix sort

- Keeping in mind that radix sort processes its data as a sequence of *m* quantities with *k* possible values, what do *m* and *k* represent in our example?

## Radix sort

- How many operations did our example above require? How many operations would the example above have required if the elements were already in sorted order? If they were in reverse order?

## Radix sort

- Which sorting method would have been more efficient for sorting the above array: radix sort or merge sort?