Section 4

CSCI E-22

Will Begin Shortly

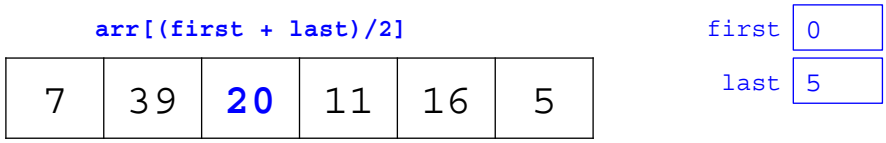
Quicksort

- 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?

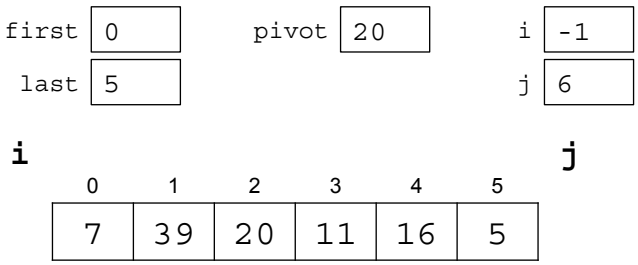
7	39	20	11	16	5

Quicksort

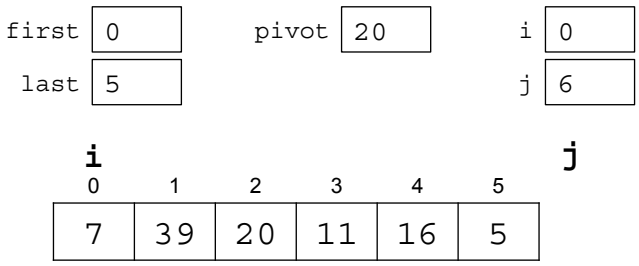
- Recursive, divide & conquer
- Achieves better average-case time complexity than O(n^2)
- Elements are partitioned into two *subarrays*
  - Elements in left subarray are *less than or equal to* elements in right subarray
- To partition, pick a pivot value, then repeatedly swap elements between subarrays 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?



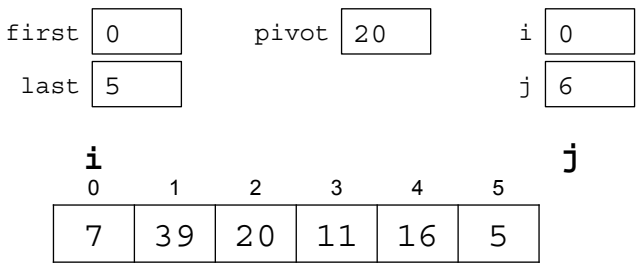

```
public static int partition(
  int[] arr, int first, int last
) {
    int pivot = arr[(first + last)/2];
    int i = first - 1;
    int j = last + 1;

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

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

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



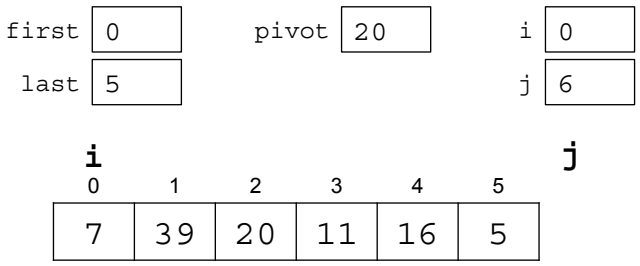
```java
public static int partition(
  int[] arr, int first, int last
) {
    int pivot = arr[(first + last)/2];
    int i = first - 1;
    int j = last + 1;

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

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

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



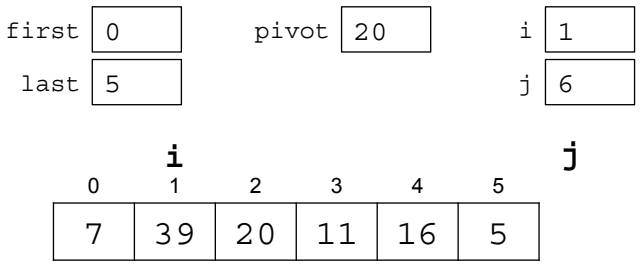

```java
public static int partition(
  int[] arr, int first, int last
) {
    int pivot = arr[(first + last)/2];
    int i = first - 1;
    int j = last + 1;

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

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

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



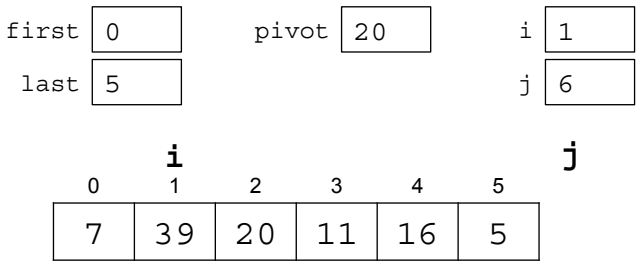
```
public static int partition(
  int[] arr, int first, int last
) {
    int pivot = arr[(first + last)/2];
    int i = first - 1;
    int j = last + 1;

    while (true) {
        do {
            i++;
        } while (arr[i] < pivot);
                 arr[0] < pivot
        do {          7 < 20
            j--;
        } while (arr[j] > pivot);

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



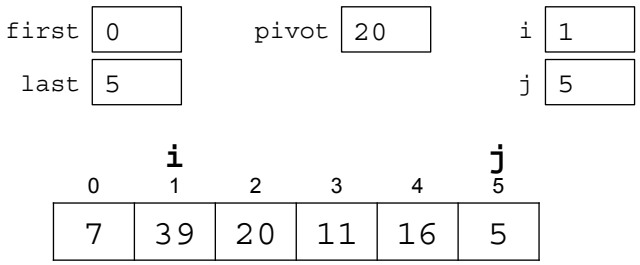

```
public static int partition(
  int[] arr, int first, int last
) {
    int pivot = arr[(first + last)/2];
    int i = first - 1;
    int j = last + 1;

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

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

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



```
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);
                 arr[1] < pivot
        do {         39 < 20
            j--;
        } while (arr[j] > pivot);

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

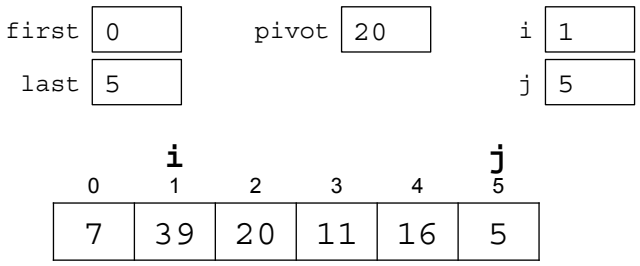



```
public static int partition(
  int[] arr, int first, int last
) {
    int pivot = arr[(first + last)/2];
    int i = first - 1;
    int j = last + 1;

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

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

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

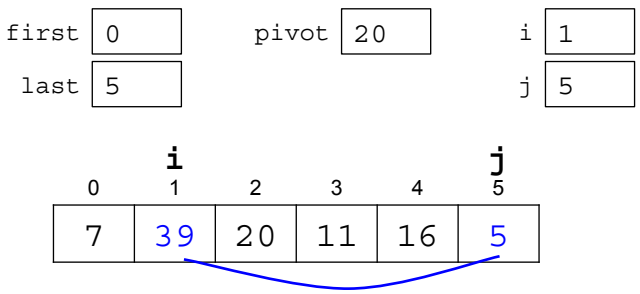


```java
public static int partition(
  int[] arr, int first, int last
) {
    int pivot = arr[(first + last)/2];
    int i = first - 1;
    int j = last + 1;

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

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

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

arr[5] > pivot
5 > 20



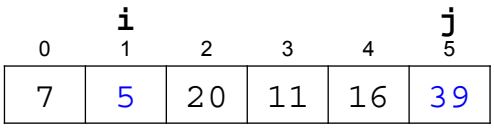

```java
public static int partition(
  int[] arr, int first, int last
) {
    int pivot = arr[(first + last)/2];
    int i = first - 1;
    int j = last + 1;

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

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

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



```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;
        }
    }
}
```

first 0 pivot 20 i 1

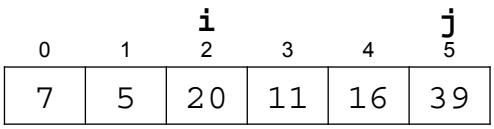
last 5 j 5




```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;
        }
    }
}
```

first 0 pivot 20 i 2

last 5 j 5



```
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);
                arr[2] < pivot
        do {        20 < 20
            j--;
        } while (arr[j] > pivot);

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

first	0	pivot	20	i	2
last	5			j	5

		i			j
0	1	2	3	4	5
7	5	20	11	16	39

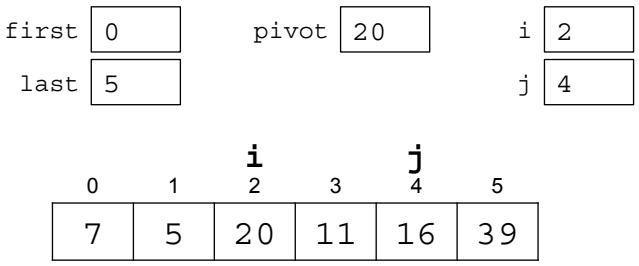
```
public static int partition(
  int[] arr, int first, int last
) {
    int pivot = arr[(first + last)/2];
    int i = first - 1;
    int j = last + 1;

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

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

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

first	0	pivot	20	i	2
last	5			j	4

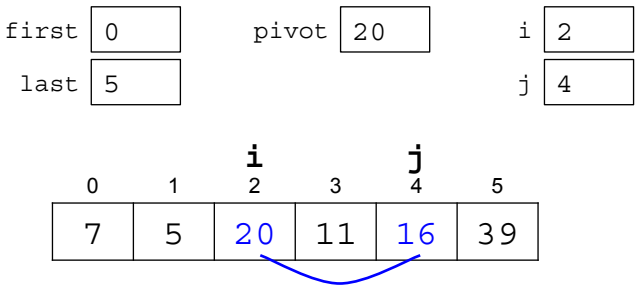
		i		j	
0	1	2	3	4	5
7	5	20	11	16	39

```
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;
        }
    }
}
```

arr[4] > pivot
16 > 20




```
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;
        }
    }
}
```

```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;
        }
    }
}
```

first	0	pivot	20	i	2
last	5			j	4

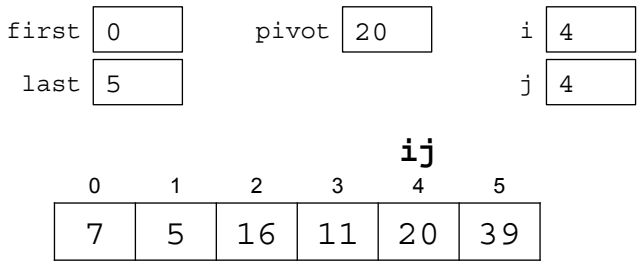
		i		j	
0	1	2	3	4	5
7	5	16	11	20	39

```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;
        }
    }
}
```

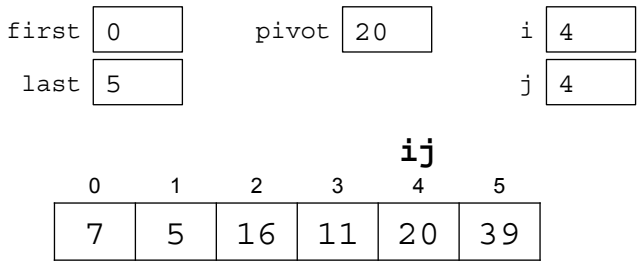
first	0	pivot	20	i	3
last	5			j	4

			i	j	
0	1	2	3	4	5
7	5	16	11	20	39

```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;
        }
    }
}
```




```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);
                arr[4] < pivot
        do {        20 < 20
            j--;
        } while (arr[j] > pivot);

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

```
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;
        }
    }
}
```

first	0	pivot	20	i	4
last	5			j	3

			j	i	
0	1	2	3	4	5
7	5	16	11	20	39


```
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;
        }
    }
}
```


arr[3] > pivot
11 > 20

first	0	pivot	20	i	4
last	5			j	3

			j	i	
0	1	2	3	4	5
7	5	16	11	20	39

```
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;
        }
    }
}
```

index of last element in left subarray

first	0	pivot	20	i	4
last	5			j	3

			j	i	
0	1	2	3	4	5
7	5	16	11	20	39

```
private static void qSort(
  int[] arr, int first, int last
) {
    int split = partition(arr, first, last);

    if (first < split) {
        qSort(arr, first, split);
    }
    if (last > split + 1) {
        qSort(arr, split + 1, last);
    }
}
```

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

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

index of last element in left subarray

first	0	split	3		
last	5				

			split		
0	1	2	3	4	5
7	5	16	11	20	39

`qSort(0, 5)`

```
private static void qSort(
  int[] arr, int first, int last
) {
    int split = partition(arr, first, last);

    if (first < split) {
        qSort(arr, first, split);
    }
    if (last > split + 1) {
        qSort(arr, split + 1, last);
    }
}
```

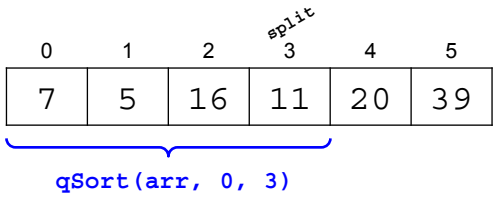
0 < 3

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

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

index of last element in left subarray

first 0 split 3

last 5



qSort(arr, 0, 3)

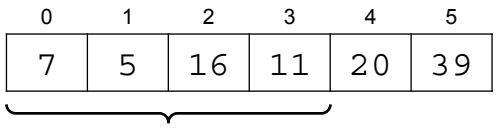
qSort(0, 5)

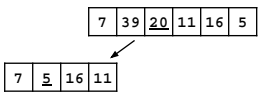
```
public static int partition(
  int[] arr, int first, int last
) {
    int pivot = arr[(first + last)/2];
    int i = first - 1;
    int j = last + 1;

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

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

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

first 0 pivot ? i

last 3 j

what is the pivot value for this subarray?





partition(0, 3)

qSort(0, 3)

qSort(0, 5)

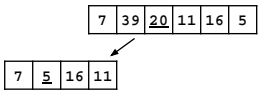
```
public static int partition(
  int[] arr, int first, int last
) {
    int pivot = arr[(first + last)/2];
    int i = first - 1;
    int j = last + 1;

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

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

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

first 0 pivot 5 i

last 3 j

what are the initial values of i and j?

0	1	2	3	4	5
7	5	16	11	20	39



partition(0, 3)
qSort(0, 3)
qSort(0, 5)

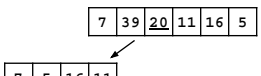

```
public static int partition(
  int[] arr, int first, int last
) {
    int pivot = arr[(first + last)/2];
    int i = first - 1;
    int j = last + 1;

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

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

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


first 0 pivot 5 i -1

last 3 j 4

i j

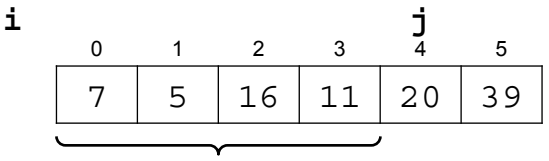
0	1	2	3	4	5
7	5	16	11	20	39



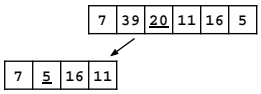
partition(0, 3)
qSort(0, 3)
qSort(0, 5)

```
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;
        }
    }
}
```

first	0	pivot	5	i	-1
last	3			j	4



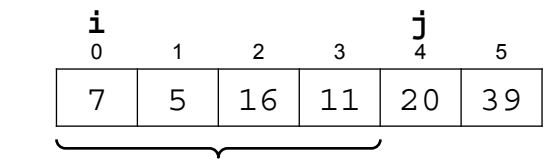
where does `i` stop?



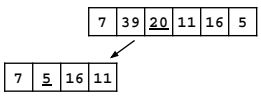
partition(0, 3)
qSort(0, 3)
qSort(0, 5)

```
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;
        }
    }
}
```

first	0	pivot	5	i	0
last	3			j	4

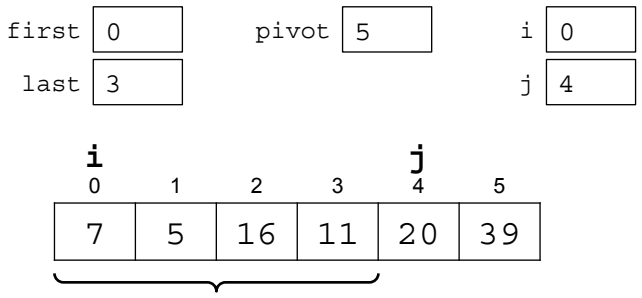


where does `i` stop? *on the **first** element that is **not** less than the pivot value (stopping condition of do-while loop)*

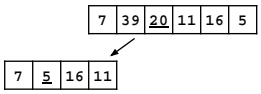


partition(0, 3)
qSort(0, 3)
qSort(0, 5)

```
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);
                 arr[0] < pivot
        do {           7 < 5
            j--;
        } while (arr[j] > pivot);

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

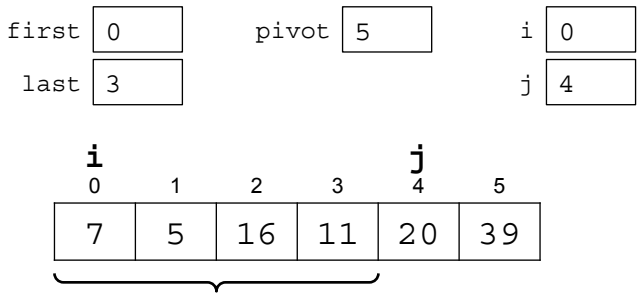


where does `i` stop? — *on the **first** element that is **not** less than the pivot value (stopping condition of do-while loop)*



partition(0, 3)
qSort(0, 3)
qSort(0, 5)


```
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;
        }
    }
}
```

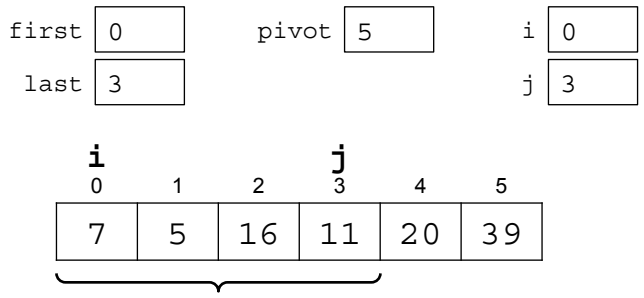


where does `i` stop? — *on the **first** element that is **not** less than the pivot value (stopping condition of do-while loop)*

where does `j` stop?

partition(0, 3)
qSort(0, 3)
qSort(0, 5)

```
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;
        }
    }
}
```



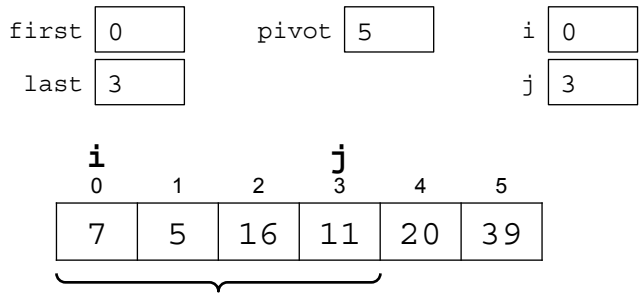
where does `i` stop? — *on the **first** element that is **not** less than the pivot value (stopping condition of do-while loop)*

where does `j` stop? — *on the **first** element that is **not** greater than the pivot value (stopping condition of do-while loop)*

partition(0, 3)
qSort(0, 3)
qSort(0, 5)


```
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;
        }
    }
}
```


arr[3] > pivot
11 > 5

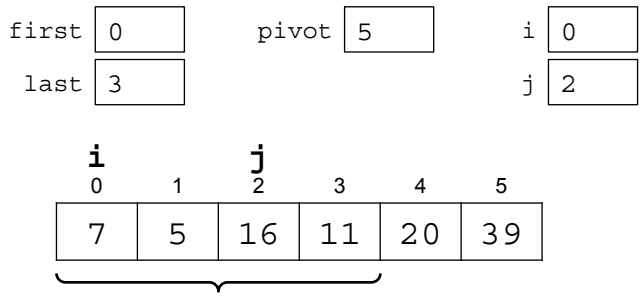


where does `i` stop? — *on the **first** element that is **not** less than the pivot value (stopping condition of do-while loop)*

where does `j` stop? — *on the **first** element that is **not** greater than the pivot value (stopping condition of do-while loop)*

partition(0, 3)
qSort(0, 3)
qSort(0, 5)

```
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;
        }
    }
}
```



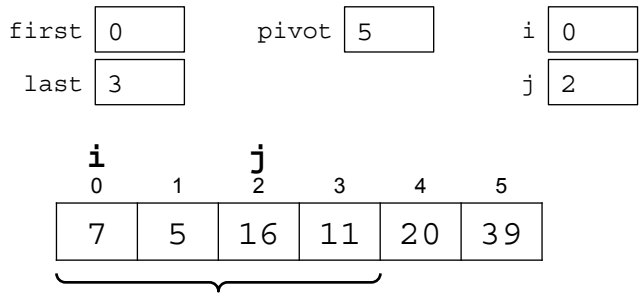
where does `i` stop? — *on the **first** element that is **not** less than the pivot value (stopping condition of do-while loop)*

where does `j` stop? — *on the **first** element that is **not** greater than the pivot value (stopping condition of do-while loop)*

partition(0, 3)
qSort(0, 3)
qSort(0, 5)


```
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;
        }
    }
}
```


arr[2] > pivot
16 > 5

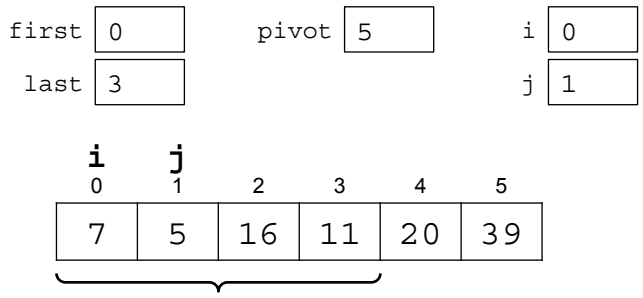


where does `i` stop? — *on the **first** element that is **not** less than the pivot value (stopping condition of do-while loop)*

where does `j` stop? — *on the **first** element that is **not** greater than the pivot value (stopping condition of do-while loop)*

partition(0, 3)
qSort(0, 3)
qSort(0, 5)

```
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;
        }
    }
}
```



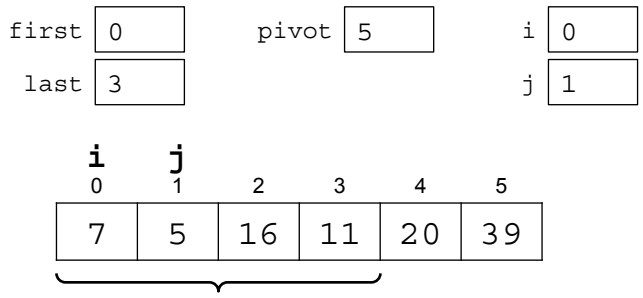
where does `i` stop? — *on the **first** element that is **not** less than the pivot value (stopping condition of do-while loop)*

where does `j` stop? — *on the **first** element that is **not** greater than the pivot value (stopping condition of do-while loop)*

partition(0, 3)
qSort(0, 3)
qSort(0, 5)


```
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;
        }
    }
}
```


arr[1] > pivot
5 > 5

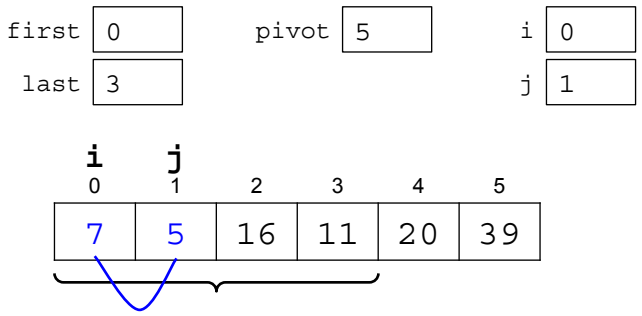


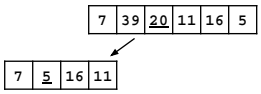
where does `i` stop? — *on the **first** element that is **not** less than the pivot value (stopping condition of do-while loop)*

where does `j` stop? — *on the **first** element that is **not** greater than the pivot value (stopping condition of do-while loop)*

partition(0, 3)
qSort(0, 3)
qSort(0, 5)

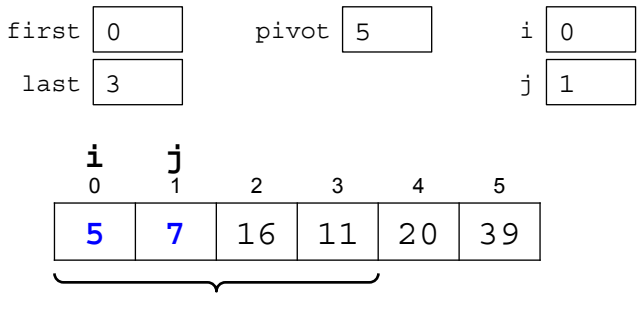
```
public static int partition(
  int[] arr, int first, int last
) {
    int pivot = arr[(first + last)/2];
    int i = first - 1;
    int j = last + 1;

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

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

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

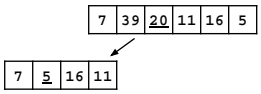




partition(0, 3)
qSort(0, 3)
qSort(0, 5)


```
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;
        }
    }
}
```



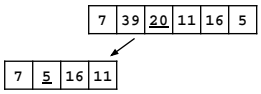
partition(0, 3)
qSort(0, 3)
qSort(0, 5)

```
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;
        }
    }
}
```

first	0	pivot	5	i	1
last	3			j	1

ij

0	1	2	3	4	5
5	7	16	11	20	39



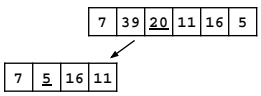
partition(0, 3)
qSort(0, 3)
qSort(0, 5)

```
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;
        }
    }
}
```

first	0	pivot	5	i	1
last	3			j	0

j i

0	1	2	3	4	5
5	7	16	11	20	39

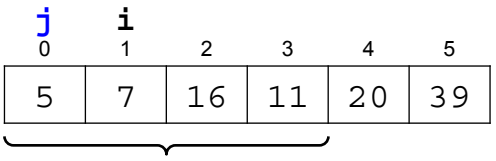


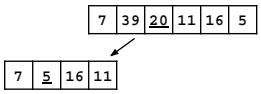
partition(0, 3)
qSort(0, 3)
qSort(0, 5)

```
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;
        }
    }
}
```

first 0 pivot 5 i 1

last 3 j 0





partition(0, 3)
qSort(0, 3)
qSort(0, 5)

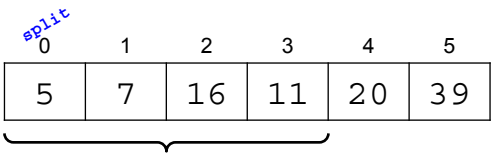
```
private static void qSort(
  int[] arr, int first, int last
) {
    int split = partition(arr, first, last);

    if (first < split) {
        qSort(arr, first, split);
    }
    if (last > split + 1) {
        qSort(arr, split + 1, last);
    }
}
```

0 < 0

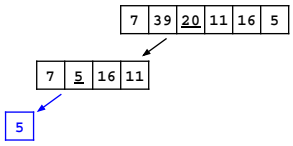
```
            j--;
        } while (arr[j] > pivot);

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

first 0 split 0

last 3



no call to `qSort()` for subarrays with one element



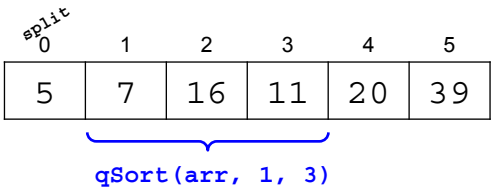
qSort(0, 3)
qSort(0, 5)

```
private static void qSort(
  int[] arr, int first, int last
) {
    int split = partition(arr, first, last);

    if (first < split) {
        qSort(arr, first, split);
    }
    if (last > split + 1) {
        qSort(arr, split + 1, last);
    }
}
```

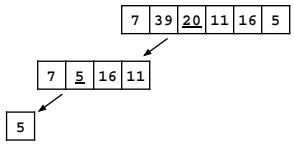
```
            j--;
        } while (arr[j] > pivot);

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

first 0 split 0

last 3



recurse on the right!

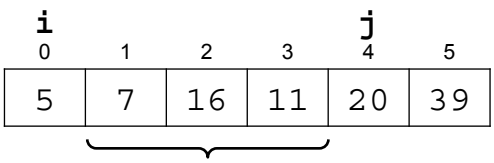


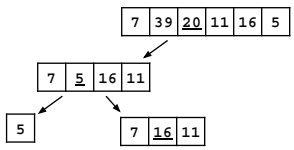
qSort(0, 3)
qSort(0, 5)

```
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;
        }
    }
}
```

first 1 pivot 16 i 0

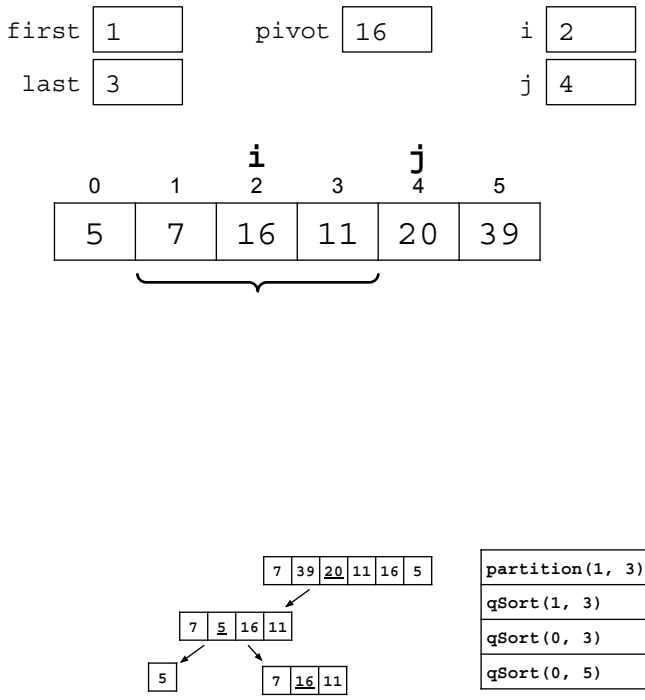
last 3 j 4



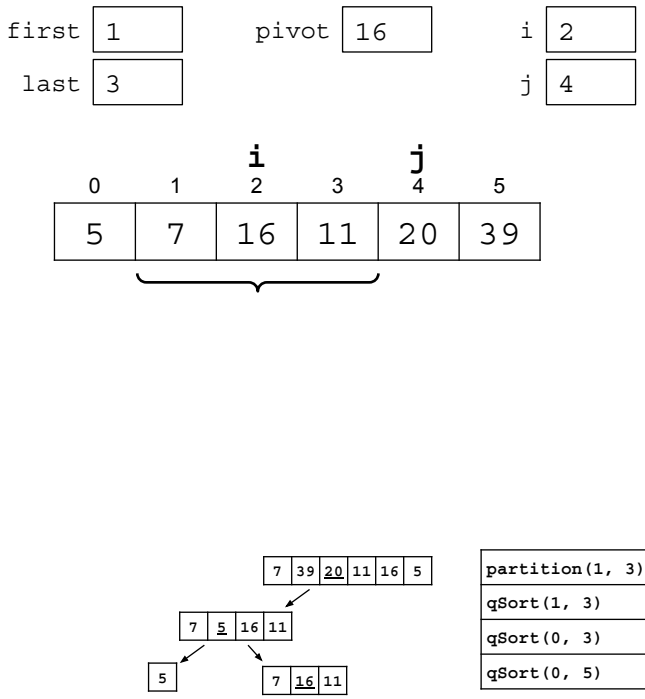


partition(1, 3)
qSort(1, 3)
qSort(0, 3)
qSort(0, 5)

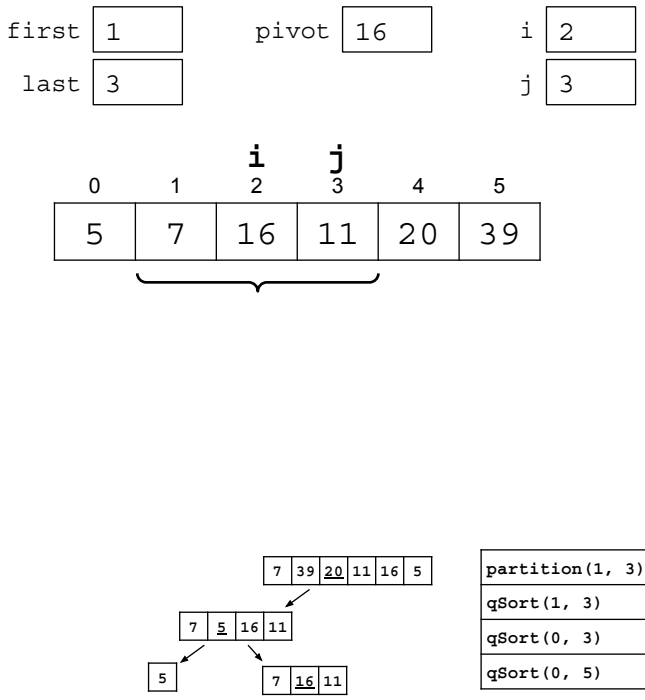
```java
public static int partition(
  int[] arr, int first, int last
) {
    int pivot = arr[(first + last)/2];
    int i = first - 1;
    int j = last + 1;

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

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

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



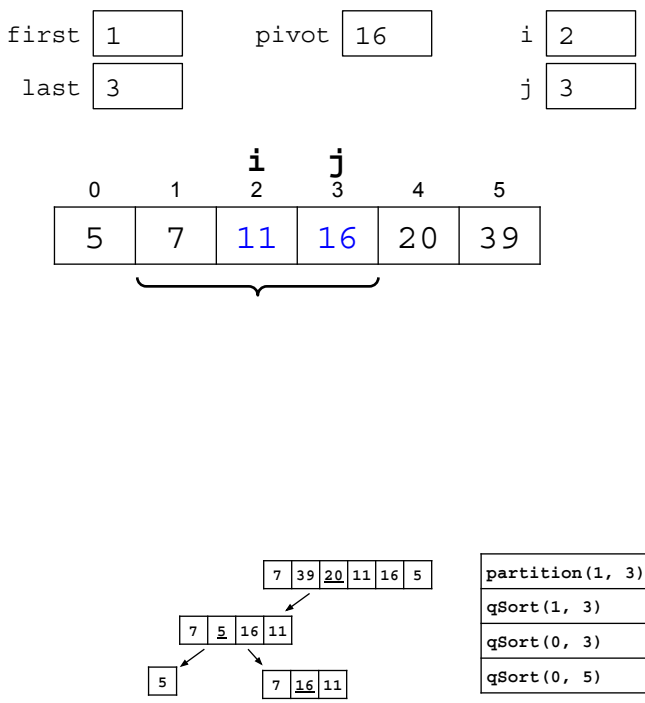

```java
public static int partition(
  int[] arr, int first, int last
) {
    int pivot = arr[(first + last)/2];
    int i = first - 1;
    int j = last + 1;

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

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

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



```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;
        }
    }
}
```

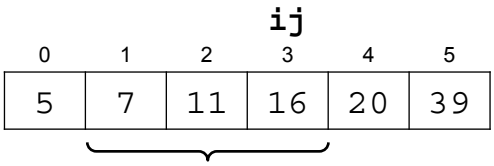


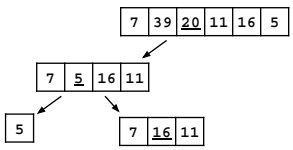

```java
public static int partition(
  int[] arr, int first, int last
) {
    int pivot = arr[(first + last)/2];
    int i = first - 1;
    int j = last + 1;

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

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

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



```
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;
        }
    }
}
```

first	1	pivot	16	i	3
last	3			j	3

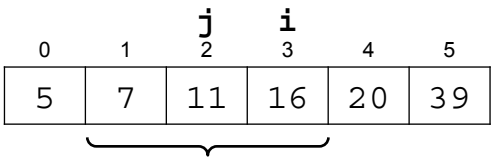


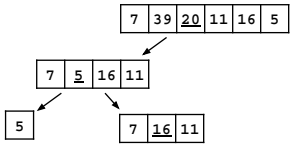


partition(1, 3)
qSort(1, 3)
qSort(0, 3)
qSort(0, 5)

```
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;
        }
    }
}
```

first	1	pivot	16	i	3
last	3			j	2



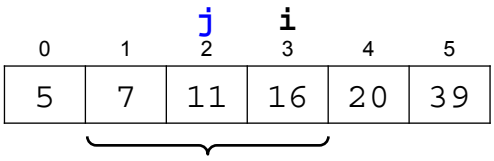


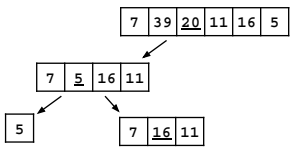
partition(1, 3)
qSort(1, 3)
qSort(0, 3)
qSort(0, 5)

```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;
        }
    }
}
```

first 1 pivot 16 i 3

last 3 j 2





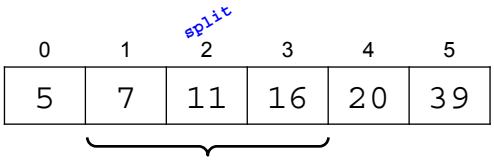
partition(1, 3)
qSort(1, 3)
qSort(0, 3)
qSort(0, 5)

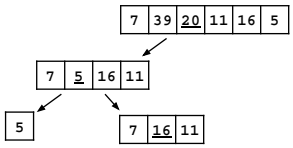
```java
private static void qSort(
  int[] arr, int first, int last
) {
    int split = partition(arr, first, last);

    if (first < split) {
        qSort(arr, first, split);
    }
    if (last > split + 1) {
        qSort(arr, split + 1, last);
    }
}
```

1 < 2

first 1 split 2

last 3





qSort(1, 3)
qSort(0, 3)
qSort(0, 5)

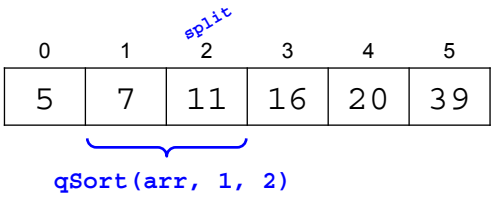
```
private static void qSort(
  int[] arr, int first, int last
) {
    int split = partition(arr, first, last);

    if (first < split) {
        qSort(arr, first, split);
    }
    if (last > split + 1) {
        qSort(arr, split + 1, last);
    }
}
```

1 < 2

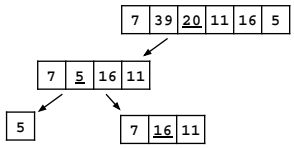
```
            j--;
        } while (arr[j] > pivot);

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

first 1 split 2

last 3



recurse on the left!

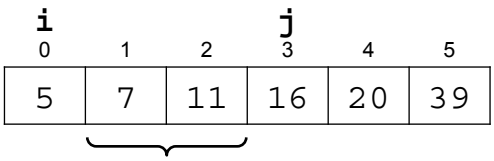


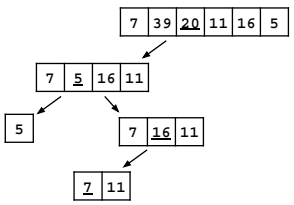
qSort(1, 3)
qSort(0, 3)
qSort(0, 5)

```
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;
        }
    }
}
```

first 1 pivot 7 i 0

last 2 j 3

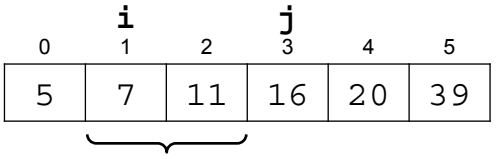


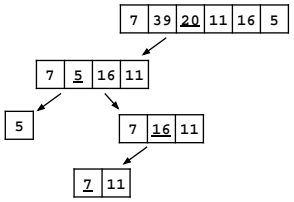


partition(1, 2)
qSort(1, 2)
qSort(1, 3)
qSort(0, 3)
qSort(0, 5)

```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;
        }
    }
}
```

first	1	pivot	7	i	1
last	2			j	3

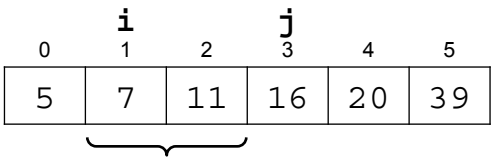


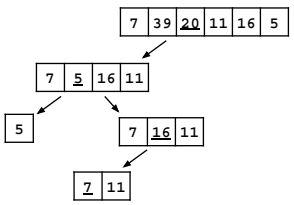


partition(1, 2)
qSort(1, 2)
qSort(1, 3)
qSort(0, 3)
qSort(0, 5)


```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;
        }
    }
}
```

first	1	pivot	7	i	1
last	2			j	3





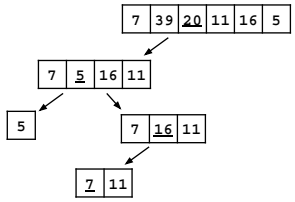
partition(1, 2)
qSort(1, 2)
qSort(1, 3)
qSort(0, 3)
qSort(0, 5)

```
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;
        }
    }
}
```

first	1	pivot	7	i	1
last	2			j	1

ij

0	1	2	3	4	5
5	7	11	16	20	39



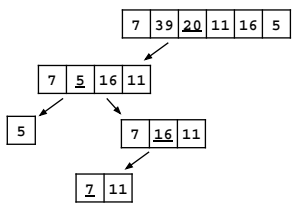
partition(1, 2)
qSort(1, 2)
qSort(1, 3)
qSort(0, 3)
qSort(0, 5)

```
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;
        }
    }
}
```

first	1	pivot	7	i	1
last	2			j	1

ij

0	1	2	3	4	5
5	7	11	16	20	39



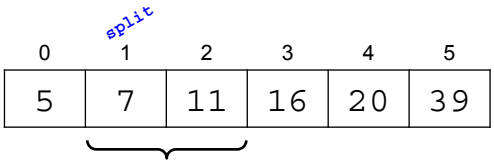
partition(1, 2)
qSort(1, 2)
qSort(1, 3)
qSort(0, 3)
qSort(0, 5)

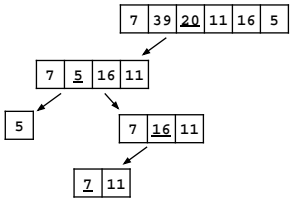
```
private static void qSort(
  int[] arr, int first, int last
) {
    int split = partition(arr, first, last);

    if (first < split) {
        qSort(arr, first, split);
    }
    if (last > split + 1) {
        qSort(arr, split + 1, last);
    }
}
```

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

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

first 1 split 1

last 2





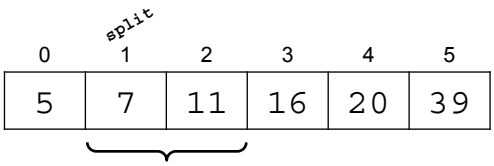
qSort(1, 2)
qSort(1, 3)
qSort(0, 3)
qSort(0, 5)


```
private static void qSort(
  int[] arr, int first, int last
) {
    int split = partition(arr, first, last);

    if (first < split) {
        qSort(arr, first, split);
    }
    if (last > split + 1) {
        qSort(arr, split + 1, last);
    }
}
```

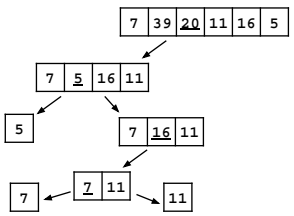
```
            j--;
        } while (arr[j] > pivot);

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

first 1 split 1

last 2



both of these checks are false, so the one-element subarrays with 7 and 11 are sorted, and we return back to `qSort(1, 3)`



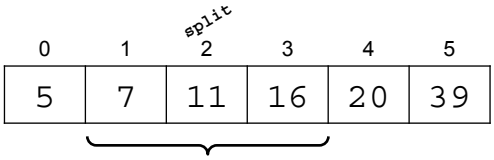
qSort(1, 2)
qSort(1, 3)
qSort(0, 3)
qSort(0, 5)

```
private static void qSort(
  int[] arr, int first, int last
) {
    int split = partition(arr, first, last);

    if (first < split) {
        qSort(arr, first, split);
    }
    if (last > split + 1) {
        qSort(arr, split + 1, last);
    }
}
```

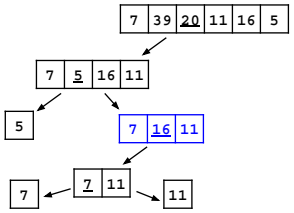
```
            j--;
        } while (arr[j] > pivot);

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

first 1 split 2

last 3



in `qSort(1, 3)` we were waiting on the left recursive call... should we do the right recursive call now?



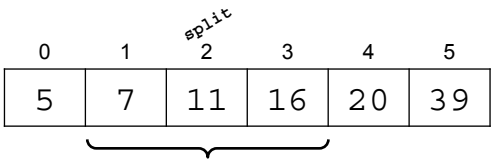
qSort(1, 3)
qSort(0, 3)
qSort(0, 5)

```
private static void qSort(
  int[] arr, int first, int last
) {
    int split = partition(arr, first, last);

    if (first < split) {
        qSort(arr, first, split);
    }                                  3 > 3
    if (last > split + 1) {
        qSort(arr, split + 1, last);
    }
}
```

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

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

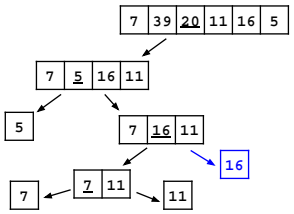
first 1 split 2

last 3



in `qSort(1, 3)` we were waiting on the left recursive call... should we do the right recursive call now?

***no**, the right subarray only contains 16, so return from `qSort(1, 3)`*



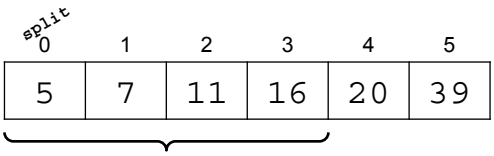
qSort(1, 3)
qSort(0, 3)
qSort(0, 5)

```
private static void qSort(
  int[] arr, int first, int last
) {
    int split = partition(arr, first, last);

    if (first < split) {
        qSort(arr, first, split);
    }
    if (last > split + 1) {
        qSort(arr, split + 1, last);
    }
}
```

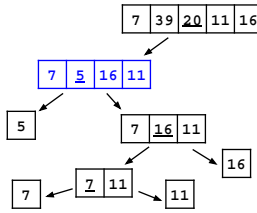
```
            j--;
        } while (arr[j] > pivot);

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

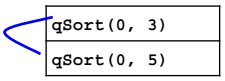
first 0 split 0

last 3



in `qSort(0, 3)` we were waiting on the right recursive call, no more work to do! return back to `qSort(0, 5)`



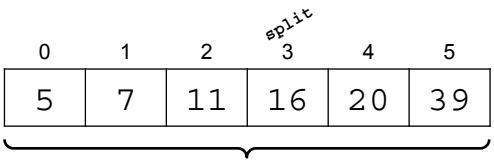



```
private static void qSort(
  int[] arr, int first, int last
) {
    int split = partition(arr, first, last);

    if (first < split) {
        qSort(arr, first, split);
    }
    if (last > split + 1) {
        qSort(arr, split + 1, last);
    }
}
```

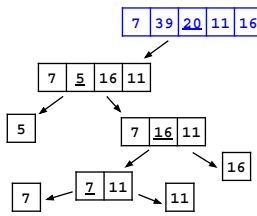
```
            j--;
        } while (arr[j] > pivot);

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

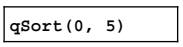
first 0 split 3

last 5



in `qSort(0, 5)` we were waiting on the left recursive call, now recurse right



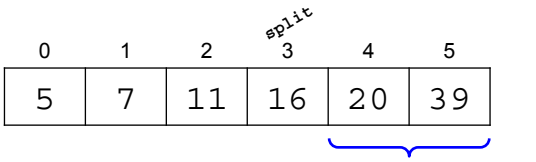


```
private static void qSort(
  int[] arr, int first, int last
) {
    int split = partition(arr, first, last);

    if (first < split) {
        qSort(arr, first, split);
    }
    if (last > split + 1) {
        qSort(arr, split + 1, last);
    }
}
```

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

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

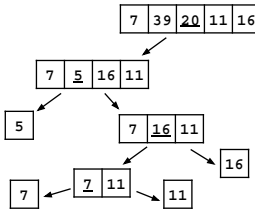
first 0 split 3

last 5



qSort(arr, 4, 5)

in `qSort(0, 5)` we were waiting on the left recursive call, now recurse right

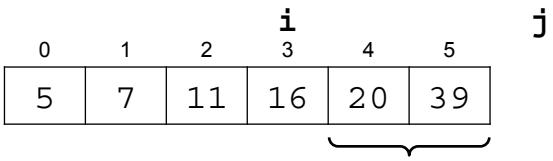


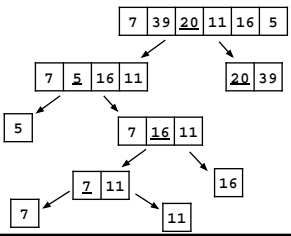
qSort(0, 5)

```
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;
        }
    }
}
```

first 4 pivot 20 i 3

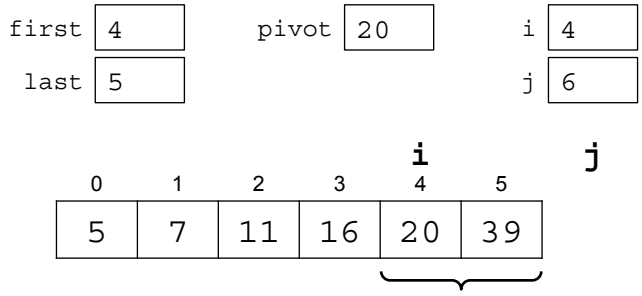
last 5 j 6

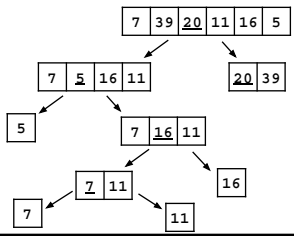




partition(4, 5)
qSort(4, 5)
qSort(0, 5)

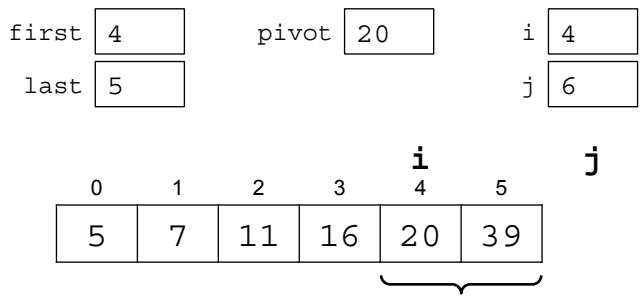
```java
public static int partition(
  int[] arr, int first, int last
) {
    int pivot = arr[(first + last)/2];
    int i = first - 1;
    int j = last + 1;

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

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

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

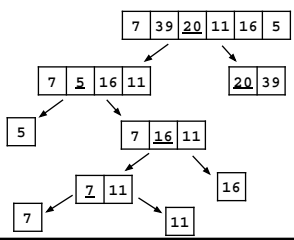




partition(4, 5)
qSort(4, 5)
qSort(0, 5)


```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;
        }
    }
}
```

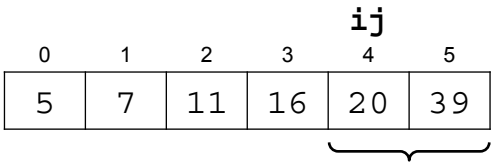



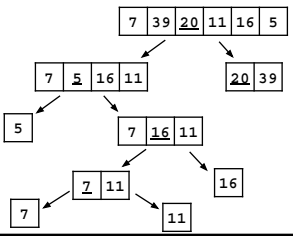


partition(4, 5)
qSort(4, 5)
qSort(0, 5)

```
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;
        }
    }
}
```

first	4	pivot	20	i	4
last	5			j	4

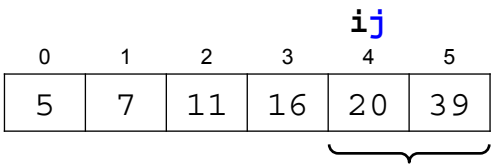


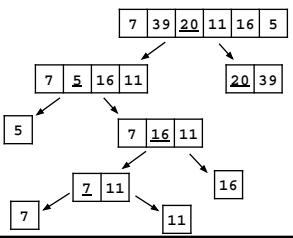


partition(4, 5)
qSort(4, 5)
qSort(0, 5)


```
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;
        }
    }
}
```


first	4	pivot	20	i	4
last	5			j	4





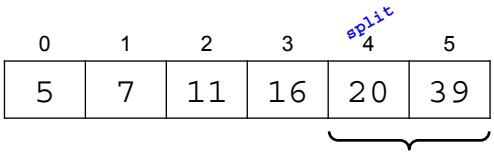
partition(4, 5)
qSort(4, 5)
qSort(0, 5)

```
private static void qSort(
  int[] arr, int first, int last
) {
    int split = partition(arr, first, last);

    if (first < split) {
        qSort(arr, first, split);
    }
    if (last > split + 1) {
        qSort(arr, split + 1, last);
    }
}
```

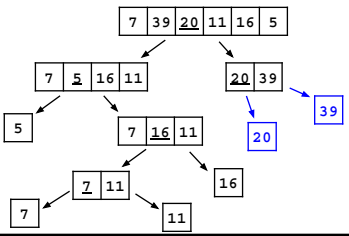
```
            j--;
        } while (arr[j] > pivot);

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

first 4 split 4

last 5



both of these checks are false, so the one-element subarrays with 20 and 39 are sorted, and we return back to `qSort(0, 5)`



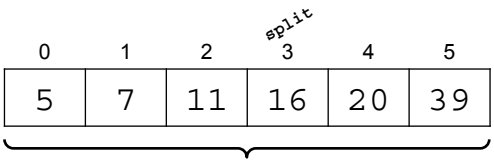
qSort(4, 5)
qSort(0, 5)

```
private static void qSort(
  int[] arr, int first, int last
) {
    int split = partition(arr, first, last);

    if (first < split) {
        qSort(arr, first, split);
    }
    if (last > split + 1) {
        qSort(arr, split + 1, last);
    }
}
```

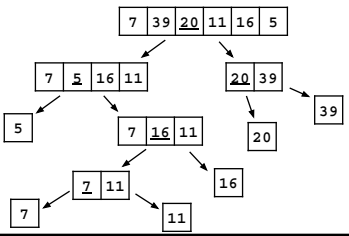
```
            j--;
        } while (arr[j] > pivot);

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

first 0 split 3

last 5



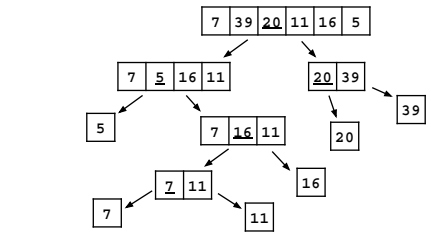
`qSort(0, 5)` was waiting on the right recursive call, so it has no more work to do, and the array is sorted



qSort(0, 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;
        }
    }
}
```

```java
private static void qSort(
  int[] arr, int first, int last
) {
    int split = partition(arr, first, last);

    if (first < split) {
        qSort(arr, first, split);
    }
    if (last > split + 1) {
        qSort(arr, split + 1, last);
    }
}
```



5 7 11 16 20 39

0	1	2	3	4	5
7	39	20	11	16	5

first

last

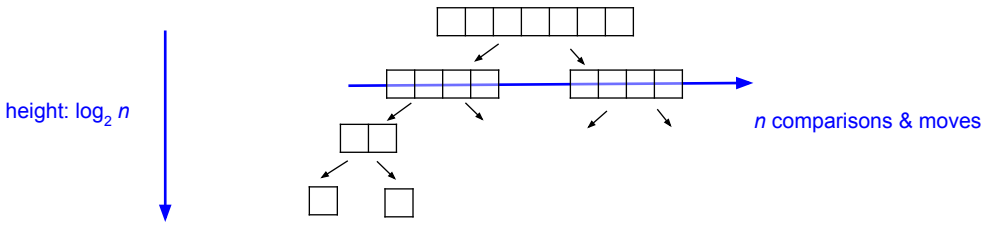
split

Quicksort

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

Quicksort

- What is the time complexity of quicksort
  - in the best case?
    Each time we partition, we divide the current subarray in half. The call tree will then have a height of $\log_2 n$, since $\log_2 n$ is an upper bound on the number of times we can divide n in half before reaching 1, and the recursion stops when we get down to subarrays of size 1. At each of the $\log_2 n$ levels of the call tree, we perform n comparisons. The number of moves per level will vary, but is still O(n). The overall best-case time complexity is therefore $\mathbf{O(n \cdot \log_2 n)}$.

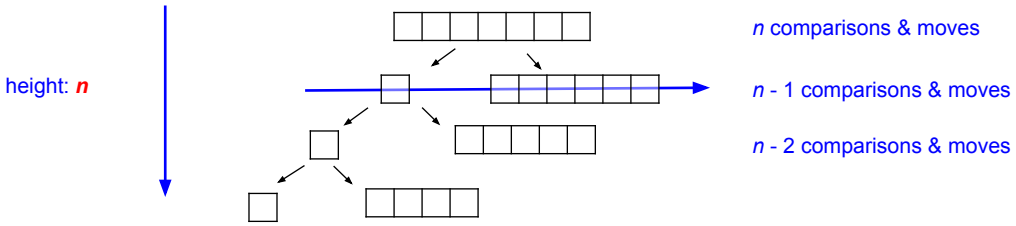


Quicksort

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

Quicksort

- What is the time complexity of quicksort
  - in the worst case?
    Each time we partition, we divide the current subarray into a subarray containing just a single element and another subarray containing all the remaining elements. In this case, the call tree will have a height of *n*, since we're reducing the problem size by **only 1 element at a time**. The number of comparisons we perform starts at *n* and goes down by 1 for each level, giving us $O(n^2)$ comparisons. We perform at most 1 swap per level, so there are $O(n)$ moves, but the overall worst-case complexity is still $\mathbf{O(n^2)}$.

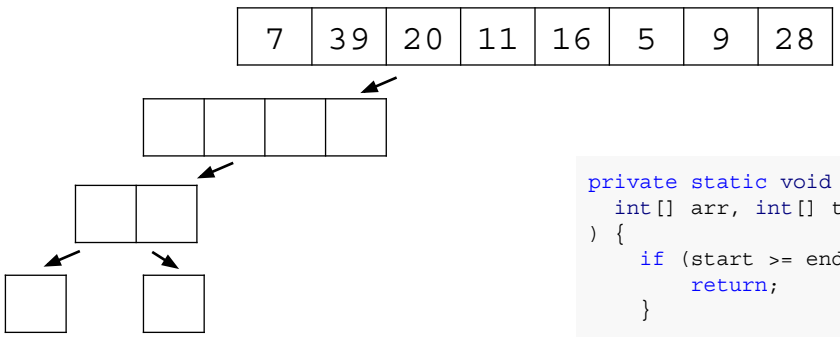


Quicksort

- What is the time complexity of quicksort
  - in the average case?
    We get a mix of balanced and unbalanced partitionings, and so the height of our call tree will be > $\log_2 n$ but < n. It can be proved that quicksort is still $O(n \cdot \log_2 n)$ on average, and that its average case is much closer to its best case than its worst 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?
  - Our input size n is 6 and the height of our call tree was 4. $\log_2(6) \approx 2.5$, so it is best described as an example of average performance.

Merge sort

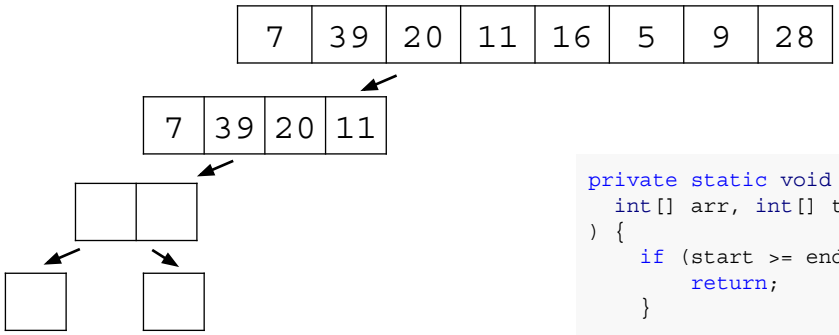
- Like quicksort, recursive divide & conquer
- Unlike quicksort, merge sort does not modify the array during the “division” phase of the algorithm
- Instead, sorting is done as subarrays are merged together into a fully sorted subarray
- `merge()` method takes two already sorted subarrays and merges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

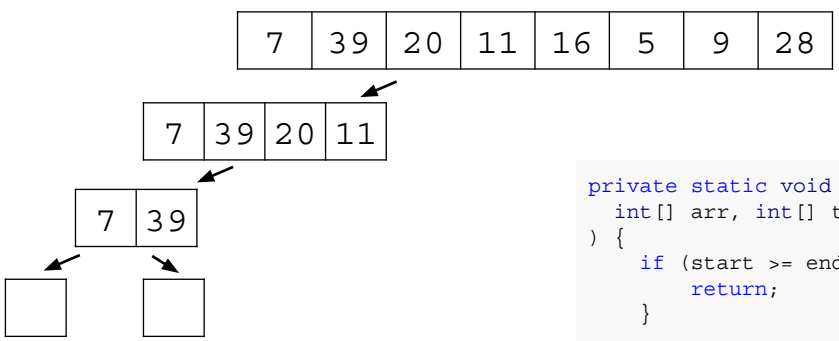
```
private static void mSort(
  int[] arr, int[] tmp, int start, int end
) {
    if (start >= end) {
        return;
    }

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



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, 3)
mSort(0, 7)

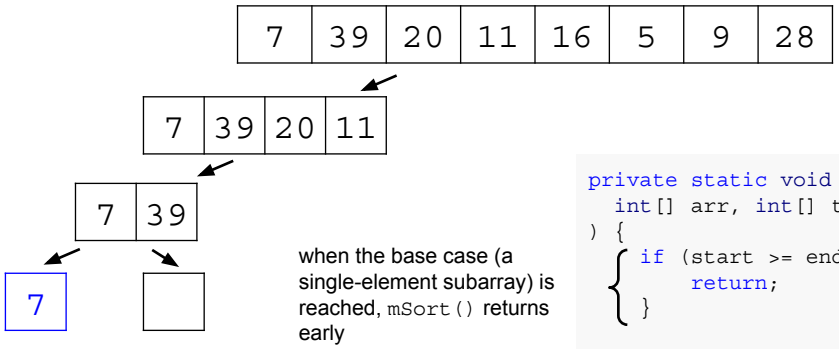
```
private static void mSort(
  int[] arr, int[] tmp, int start, int end
) {
    if (start >= end) {
        return;
    }

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



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

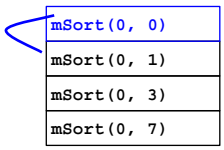
mSort(0, 1)
mSort(0, 3)
mSort(0, 7)

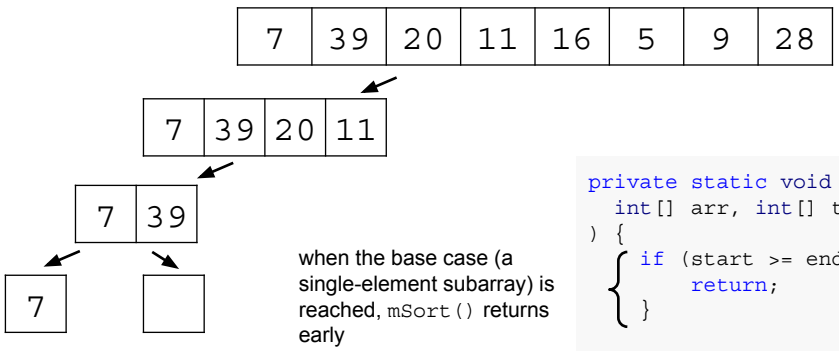
```
private static void mSort(
  int[] arr, int[] tmp, int start, int end
) {
    if (start >= end) {
        return;
    }

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



when the base case (a single-element subarray) is reached, `mSort()` returns early

```
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
  );
}
```

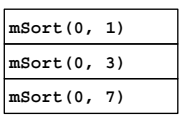


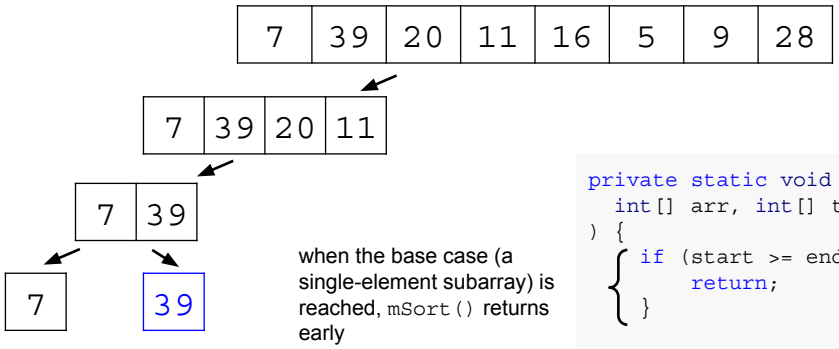


when the base case (a single-element subarray) is reached, `mSort()` returns early

what happens next?

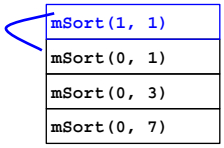
```
private static void mSort(
  int[] arr, int[] tmp, int start, int end
) {
  if (start >= end) {
      return;
  }

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



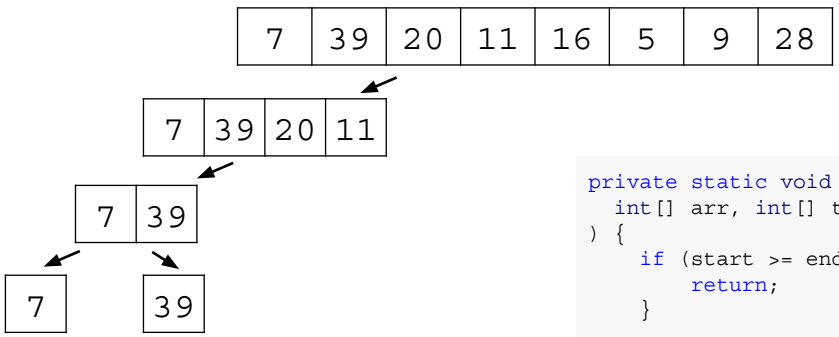


when the base case (a single-element subarray) is reached, `mSort()` returns early

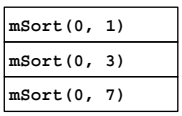
`mSort(1, 1)` reaches the base case and returns early



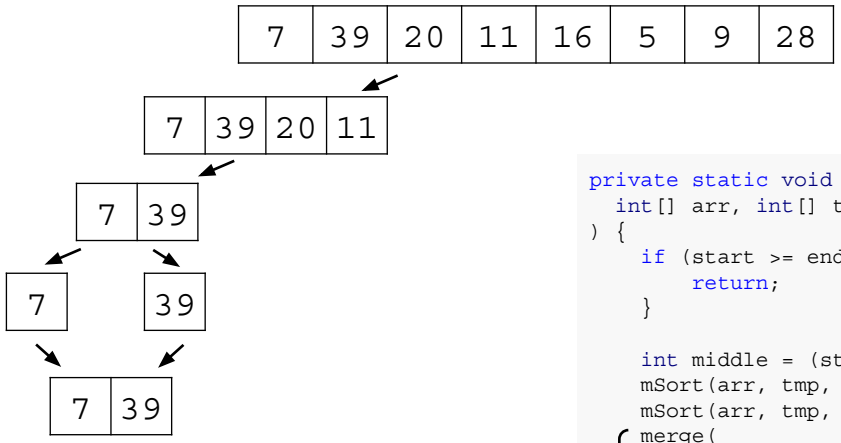

```
private static void mSort(
  int[] arr, int[] tmp, int start, int end
) {
  if (start >= end) {
      return;
  }

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



having made both left and right recursive calls, `mSort(0, 1)` calls `merge(0, 0, 1, 1)`

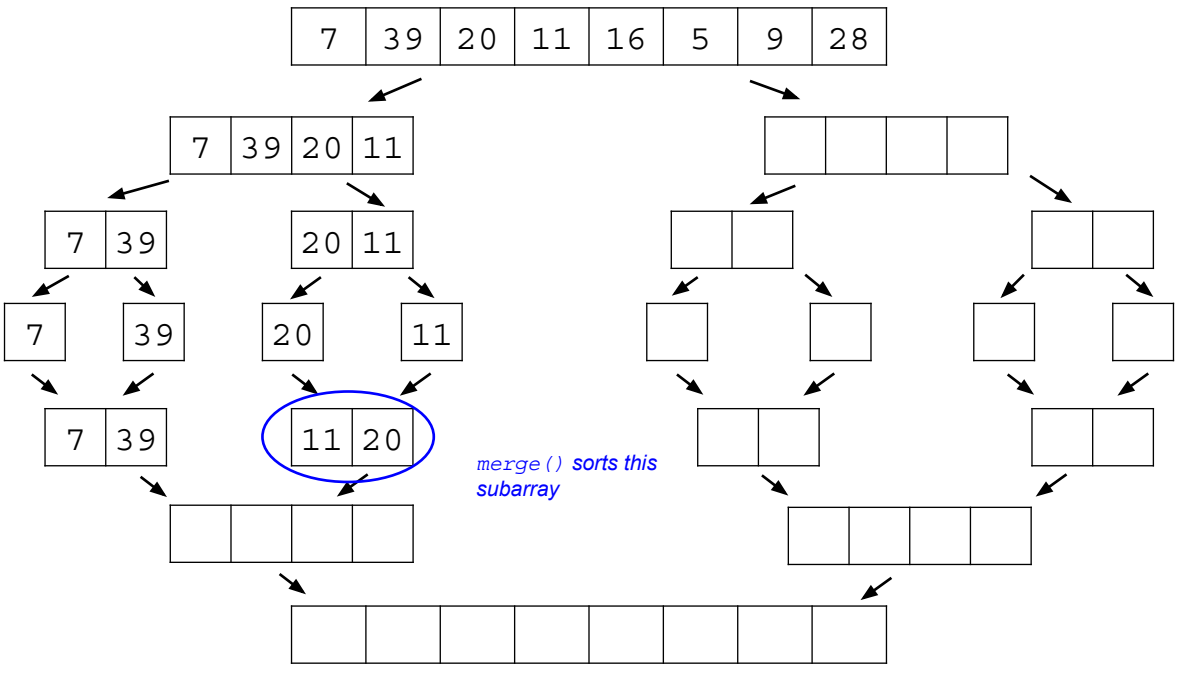


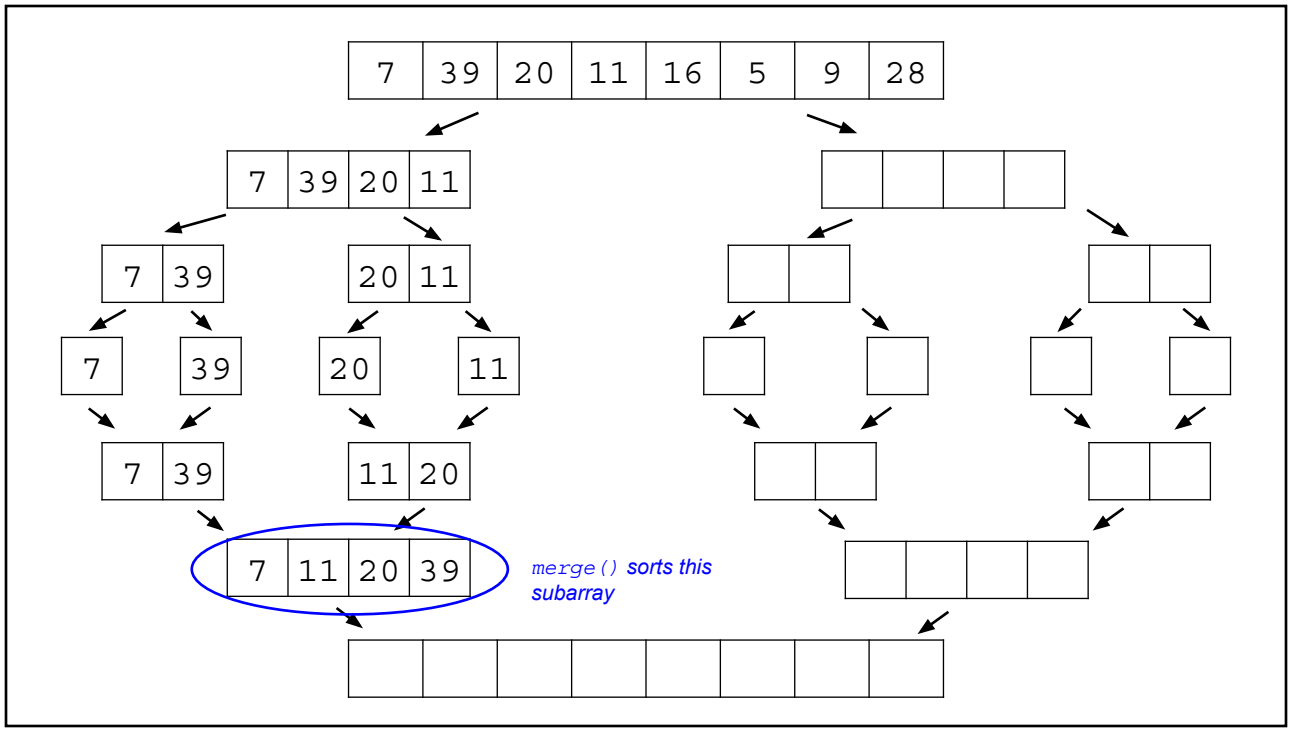

```
private static void mSort(
  int[] arr, int[] tmp, int start, int end
) {
  if (start >= end) {
      return;
  }

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

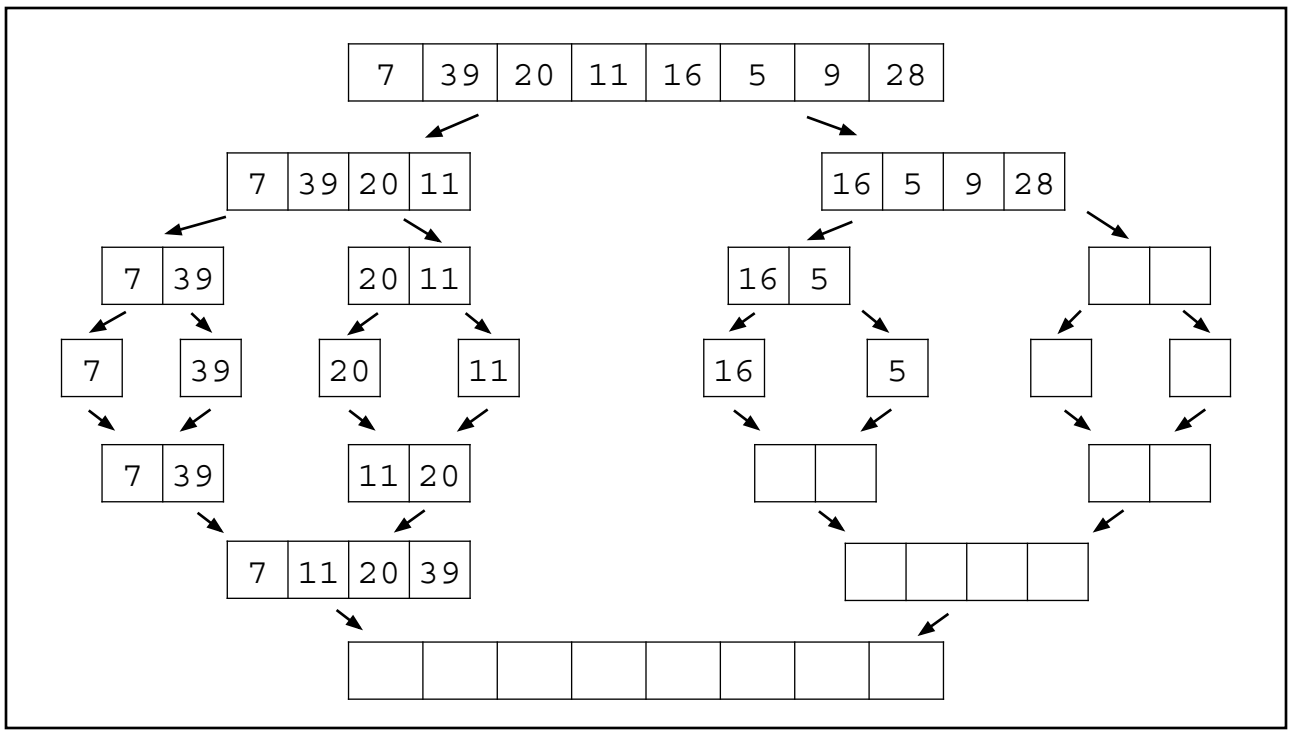


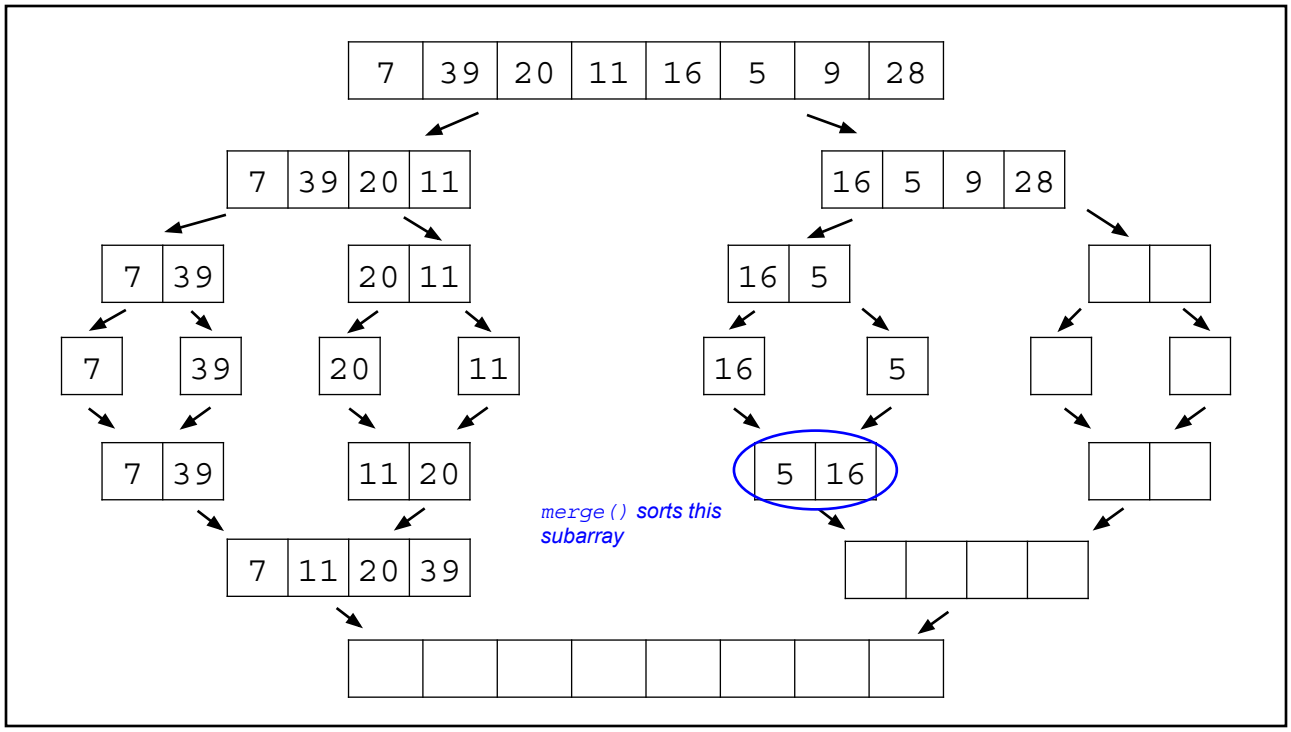

```
private static void mSort(
  int[] arr, int[] tmp, int start, int end
) {
    if (start >= end) {
        return;
    }

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

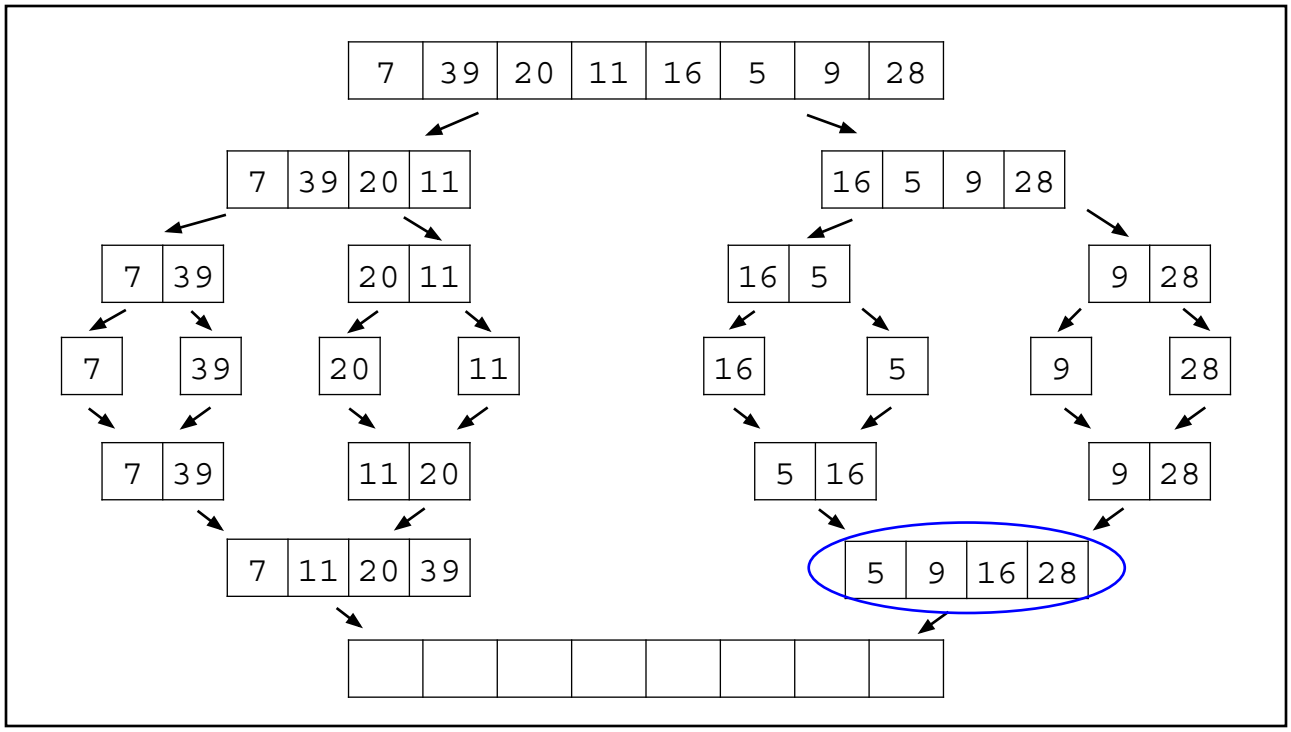
in the actual code, a second temporary array is used for space efficiency; this is not shown here

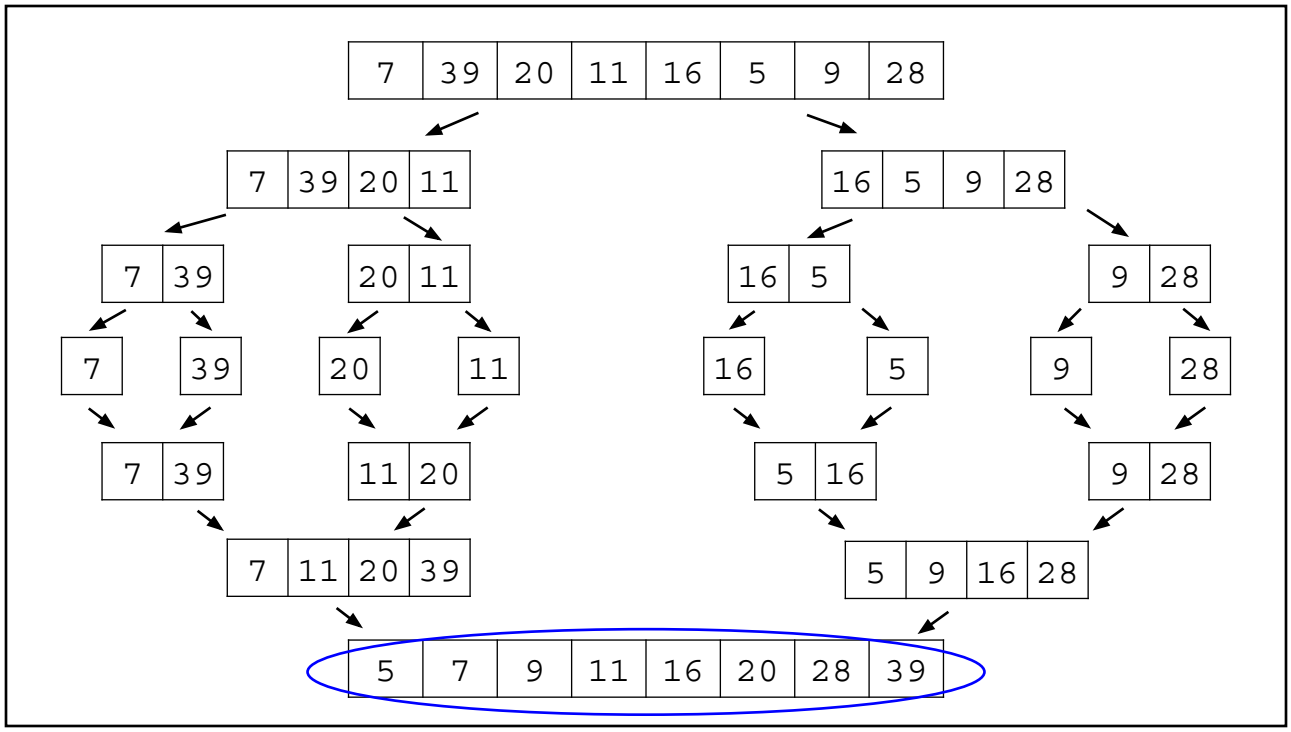



7 39 20 11 16 5 9 28
7 39 20 11
7 39
20 11
7
39
20
11
7 39
11 20
7 11 20 39
merge() sorts this subarray


7 39 20 11 16 5 9 28
7 39 20 11
16 5 9 28
7 39
20 11
16 5
7
39
20
11
16
5
7 39
11 20
7 11 20 39


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


7 39 20 11 16 5 9 28
7 39 20 11
16 5 9 28
7 39
20 11
16 5
9 28
7
39
20
11
16
5
9
28
7 39
11 20
5 16
9 28
7 11 20 39
5 9 16 28



Merge sort

- What major **advantage** does merge sort have over quicksort with respect to **time** complexity?
  - Unlike quicksort, merge sort always divides the current subarray evenly in half, and so its call tree is always perfectly balanced, with height proportional to $\log_2 n$. Therefore, merge sort gives $O(n \cdot \log_2 n)$ performance even in the worst case, whereas quicksort can degenerate into $O(n^2)$ performance if it does not partition evenly and its call tree approaches a height of n.
- What major **disadvantage** does merge sort have compared to quicksort with respect to **space** complexity?
  - Merge sort requires $O(n)$ additional memory on top of the array itself, while quicksort uses only $O(1)$ additional memory.

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
	41								

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**			

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**	

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 **1**					326		18	

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 1					326	**117**	18	

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 1					326 **56**	117	18	

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 1					326 56 **86**	117	18	

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 1					326 56 86	117 **7**	18	

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 1			**14**		326 56 86	117 7	18	

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 1 **221**			14		326 56 86	117 7	18	

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 1 221			14		326 56 86	117 7	18	**19**

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

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

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

second pass (buckets for the tens digit)

0	1	2	3	4	5	6	7	8	9
			30						

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

second pass (buckets for the tens digit)

0	1	2	3	4	5	6	7	8	9
			30	**41**					

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

second pass (buckets for the tens digit)

0	1	2	3	4	5	6	7	8	9
1			30	41					

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

second pass (buckets for the tens digit)

0	1	2	3	4	5	6	7	8	9
1		**221**	30	41					

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

second pass (buckets for the tens digit)

0	1	2	3	4	5	6	7	8	9
1	**14**	221	30	41					

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

second pass (buckets for the tens digit)

0	1	2	3	4	5	6	7	8	9
1	14	221 **326**	30	41					

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

second pass (buckets for the tens digit)

0	1	2	3	4	5	6	7	8	9
1	14	221 326	30	41	**56**				

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

second pass (buckets for the tens digit)

0	1	2	3	4	5	6	7	8	9
1	14	221 326	30	41	56			**86**	

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

second pass (buckets for the tens digit)

0	1	2	3	4	5	6	7	8	9
1	14 **117**	221 326	30	41	56			86	

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

second pass (buckets for the tens digit)

0	1	2	3	4	5	6	7	8	9
1 **7**	14 117	221 326	30	41	56			86	

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

second pass (buckets for the tens digit)

0	1	2	3	4	5	6	7	8	9
1 7	14 117 **18**	221 326	30	41	56			86	

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

second pass (buckets for the tens digit)

0	1	2	3	4	5	6	7	8	9
1 7	14 117 18 **19**	221 326	30	41	56			86	

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

third pass (buckets for the hundreds 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
1 7	14 117 18 19	221 326	30	41	56			86	

third pass (buckets for the hundreds digit)

0	1	2	3	4	5	6	7	8	9
1 7 14 18 19 30 41 56 86	117	221	326						

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?
  - In the example, *m* = 3, because there are 3 positions we're looking at: the 1's position, the 10's position, and the 100's position.
  - In general, if we have one bucket for each value of our given radix, *m* is equal to the positional index of the most significant digit of the largest element. Here, since we're sorting integers in base 10, and we'll have one bucket for each digit 0–9.
  - We can think of *k* as the **number of buckets** we have—that is, it represents the number of possible values we could have for each of *m* quantities. So, as we just mentioned, *k* = 10 in the above example, and in any case in which we're using radix sort to sort integers by significant digit.

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?
  - In general radix sort takes $n \cdot m$ steps for an array with n elements, since it iterates over all n elements of the array a total of m times.
  - Our previous example required $12 \cdot 3 = 36$ steps. It would also require 36 steps if it had already been in sorted order, or reverse-sorted order. That is, the number of steps performed by radix sort is dependent only on the values of n and m, and not on the order of the original array.

Radix sort

- Which sorting method would have been more efficient for sorting the above array: radix sort or merge sort?
  - As discussed in lecture, radix sort is $O(n \cdot m)$, whereas merge sort is $O(n \cdot \log n)$. So radix sort is more efficient than merge sort (and other comparison-based sorting algorithms in $O(n \cdot \log n)$, such as quicksort) when $m < \log n$.
  - Here, $m = 3$ and $\log_2 n = 3.585\ldots$, so radix sort would sort the above example **in fewer steps** than merge sort.

End of section.

Questions?

Lecture 5

CSCI E-22

Will Begin Shortly