As you arrive:

Snarf Sorter, and complete the mySort method.

Thou shalt not:
0. Use Arrays.sort.
1. Use Collections.sort.
2. Stick everything into a PriorityQueue then take it all out again.
3. Engage in other trickery.

Instead, come up with your very own sorting algorithm! *Don't worry about efficiency for now.*

You should:
0. Work in a group of size >= 3 and <= 4.
1. Pick somebody in your group to be spokesperson.
2. *Test your code!*

Today: *Sorting*

An IBM card sorter
(thanks Wikipedia!)

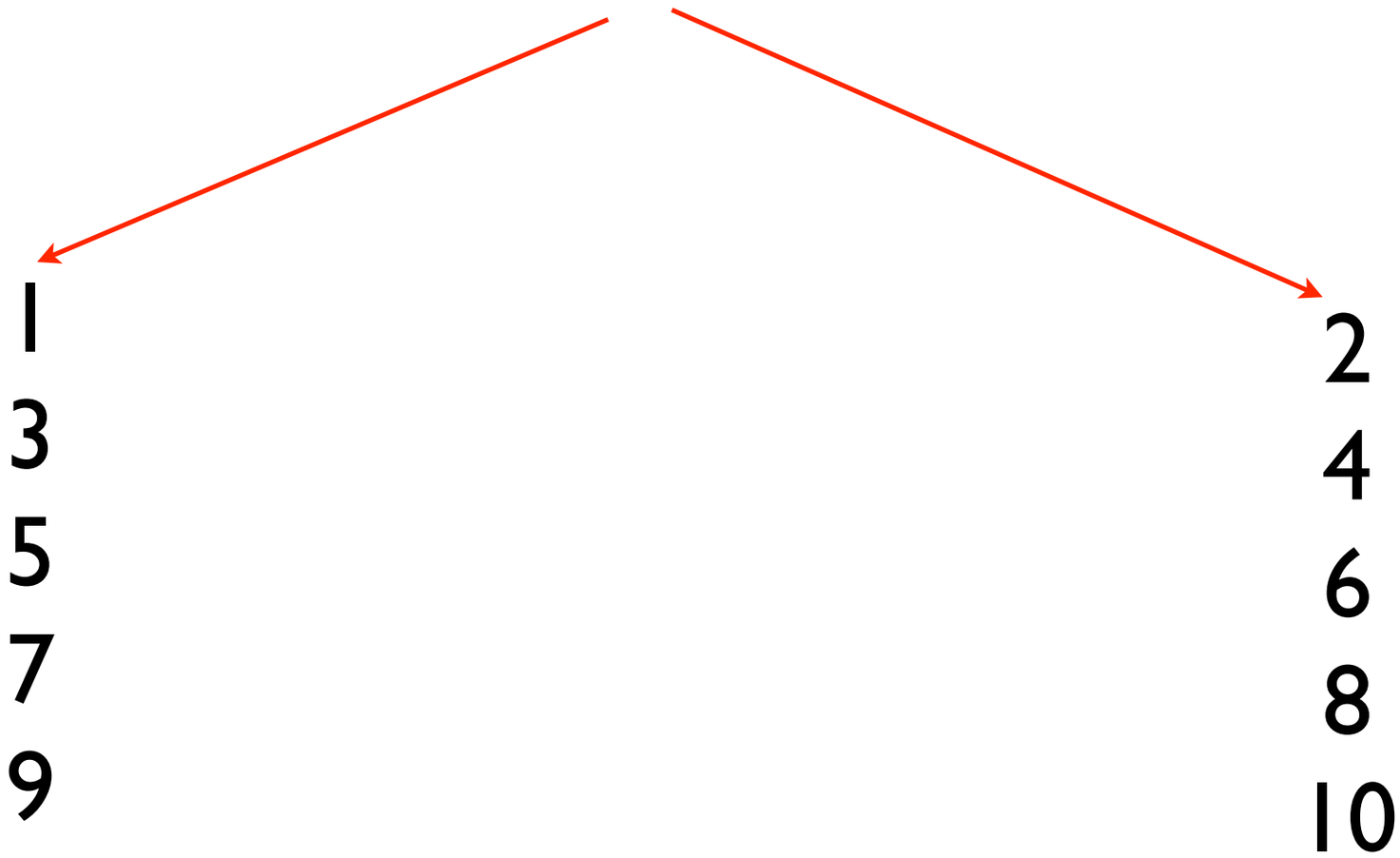
Wikipedia lists *thirteen* sorting algorithms.

I know of at least two more.

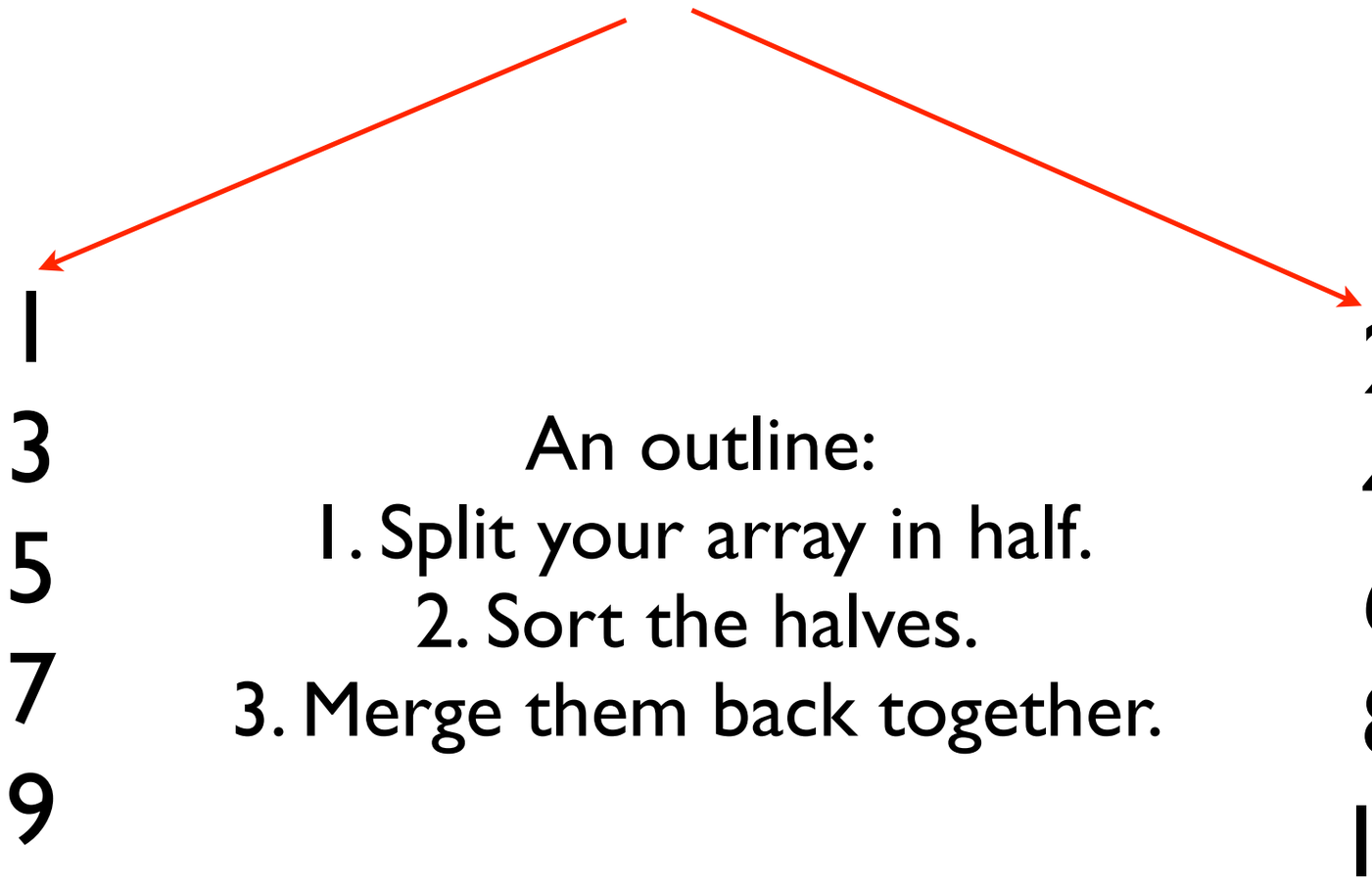
But there are only two that you need to know.

A true story

A true story



A true story



An outline:

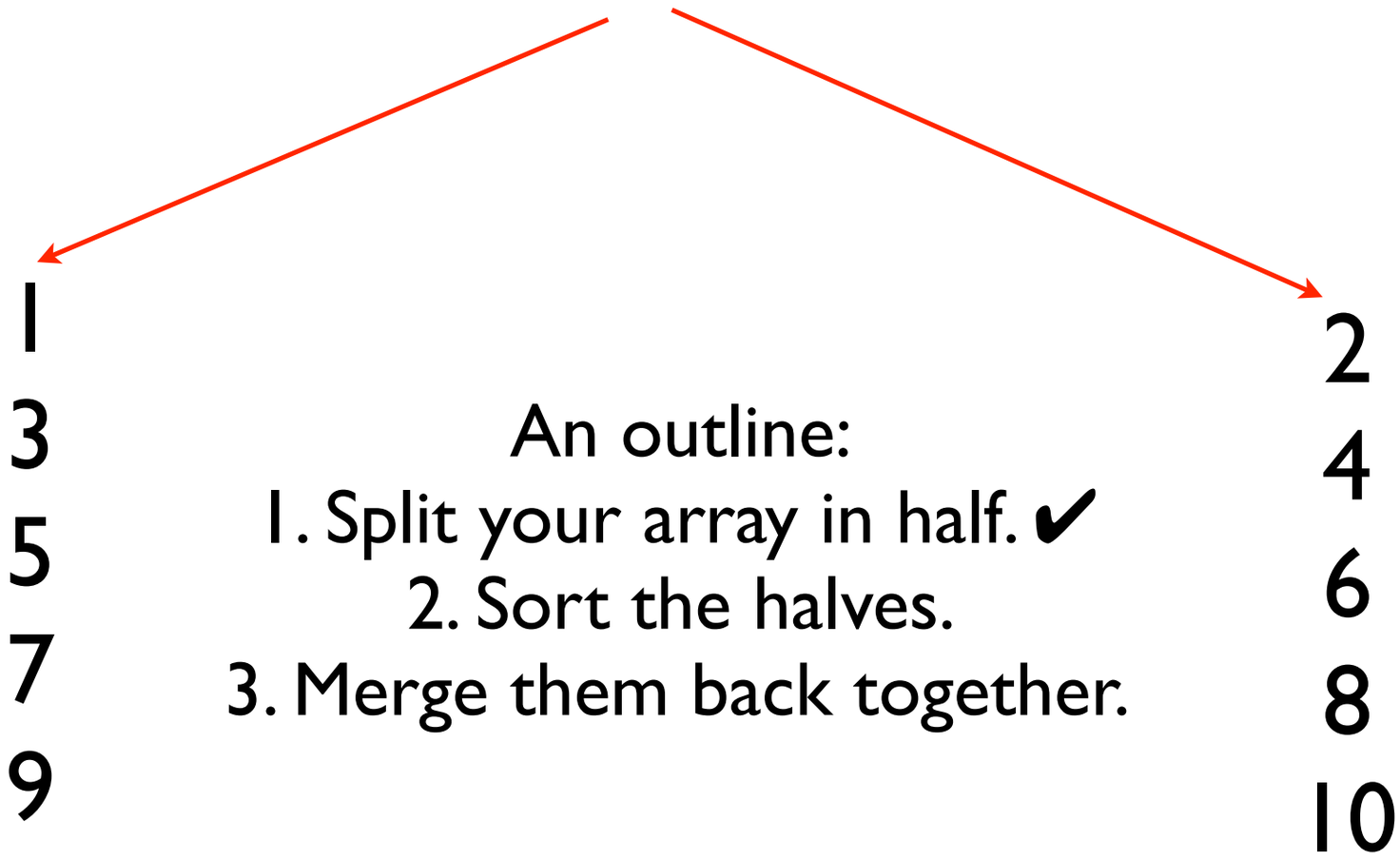
1. Split your array in half.
2. Sort the halves.
3. Merge them back together.

Implement copyLeftHalf & copyRightHalf with your group.

Make sure to write code in main to test them!

Complexity of step 1

A true story



An outline:

1. Split your array in half. ✔
2. Sort the halves.
3. Merge them back together.

Implement merge with your group.

Make sure to write code in main to test it!

Complexity of step 3

Almost there...

An outline:

1. Split your array in half. ✔
2. Sort the halves.
3. Merge them back together. ✔

```
public static int[] mergeSort(int[] input) {
   // We need a base case!

   // Non-base case.
   int[] leftHalf = copyLeftHalf(input);
   int[] rightRight = copyRightHalf(input);

   // Fill some stuff in...

   return merge(something, somethingElse);
}
```

Almost there...

An outline:

1. Split your array in half. ✔
2. Sort the halves.
3. Merge them back together. ✔

```
public static int[] mergeSort(int[] input) {
   // We need a base case!

   // Non-base case.
   int[] leftHalf = copyLeftHalf(input);
   int[] rightRight = copyRightHalf(input);

   // Fill some stuff in...

   return merge(something, somethingElse);
}
```

Implement mergeSort with your group.
Test it. Then, compute its Big-O complexity.

How's your constant factor?

```
public static int[] mergeSort(int[] input) {
   // We need a base case!

   // Non-base case.
   int[] leftHalf = copyLeftHalf(input);
   int[] rightRight = copyRightHalf(input);

   // Fill some stuff in...

   return merge(something, somethingElse);
}
```

Sorting in-place

1 2 6 8 5 4 9 3 0

Sorting in-place

1 2 6 8 5 4 9 3 0

pivot

1 2 0 3 4 5 9 8 6

Sorting in-place

1 2 6 8 5 4 9 3 0

pivot

1 2 0 3 4 5 9 8 6

Less-thans (unsorted)

Greater-thans (unsorted)

Note that this one element will never have to move again!

Sorting in-place

1 2 6 8 5 4 9 3 0

pivot

1 2 0 3 4 5 9 8 6

Less-thans (unsorted) *Greater-thans (unsorted)*

Note that this one element will never have to move again!

```
public static int[] quickSort(int[] input, int low, int high) {
      // We need a base case!

      // Non-base case.
      pivot(input, low, high, 0);
      // Make sure recursive calls...
}
```

Lower bounds

Sorting is $\Omega(n \log n)$

This is a *lower bound:* we can't hope to sort better, no matter the algorithm. Whoa!

QuickSort is $O(n \log n)$ if your pivot is well-chosen.

MergeSort is $O(n \log n)$

=>

MergeSort is $\Theta(n \log n)$

Caveat: can't hope to do better *with a comparison-based sort.* See radix sort or counting sort for ways of sorting certain kinds of data faster.

Finally

Be sure you've submitted your Sorting code.

Put everybody's NetIDs at the top of the file!

http://goo.gl/cmVl