MergeSort

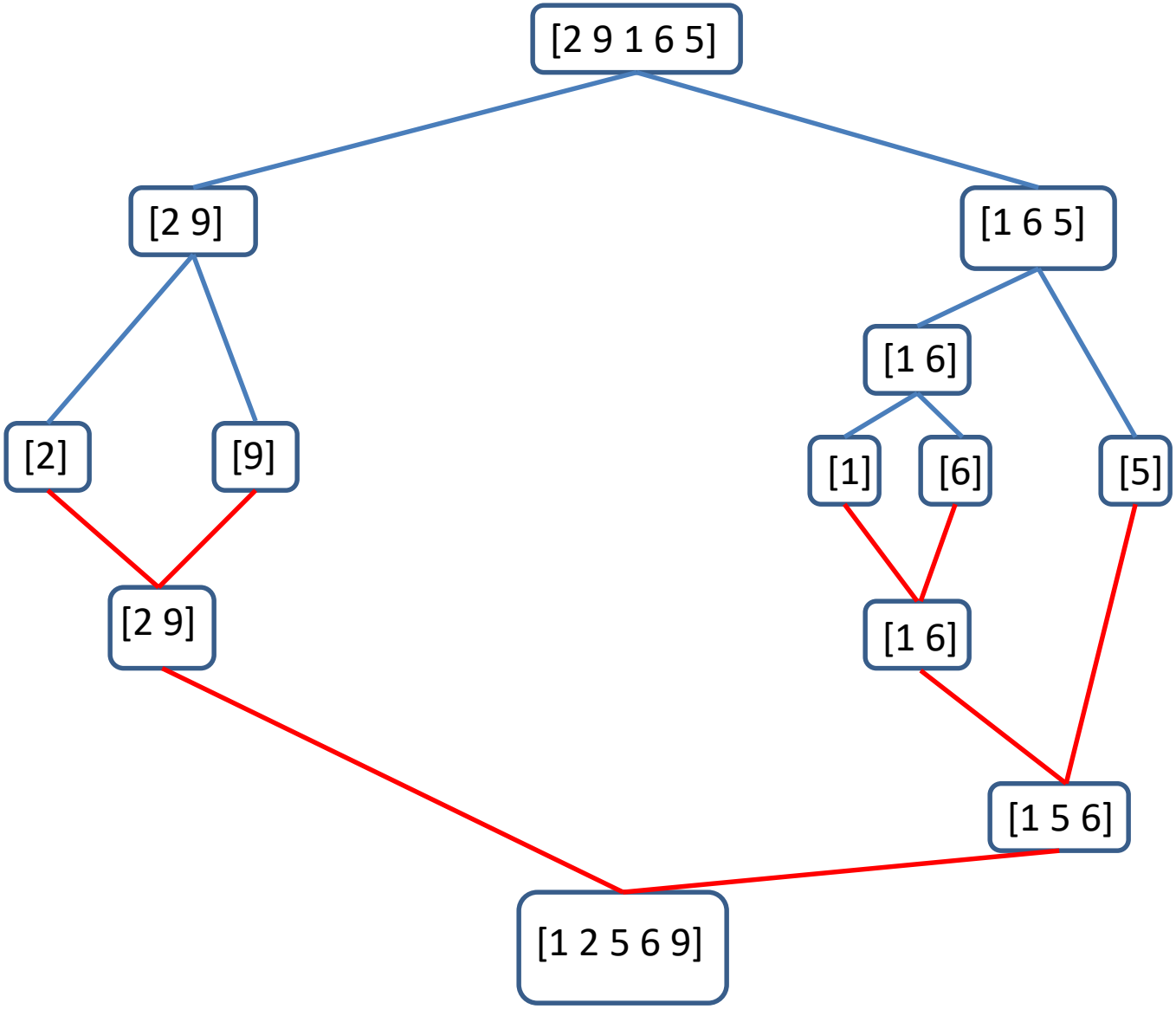
This algorithm is based on a simple observation. Suppose you have n names on pieces of paper divided into two piles, and each pile is already sorted. You can merge those two sorted piles into one sorted with no more than n comparisons. At each step you compare the top (smallest) element from each pile and pick up the smaller of the two. When one of the piles is empty you pick up all of the rest in one step.

In data structures terms we can merge two sorted lists whose sizes sum to n into one sorted list in time O(n).

The MergeSort algorithm is an easy recursion:

To sort a list of n items split it into two halves, recursively sort the halves, and merge them back into one sorted list. The base case for the recursion is having only one element in the list, in which case there is nothing to sort.

Before we give code for this, think about how it works:


[2 9 1 6 5]
[2 9]
[1 6 5]
[2]
[9]
[1 6]
[1]
[6]
[5]
[2 9]
[1 6]
[1 5 6]
[1 2 5 6 9]

The tree with blue edges represents the recursive call; the list is broken down into small and smaller pieces until the pieces are of size 1.

The tree with red edges represents the calls to merge. The small lists are merged into larger and larger sorted lists.

All of the work happens in the calls to merge. How many comparisons are there? Start with the row of n lists of size 1. It takes a total of n comparisons to merge all of these into n/2 lists of size 2. Each element is in one of these. It then takes n comparisons to merge all of these lists of size 2 into n/4 lists of size 4. And so we work our way up the red tree, using n comparisons to go from one level to the next.

How many levels are there? Whether we look at the red tree or the blue one there are log(n) levels -- you can only divide a list of size n in half log(n) times until it gets down to size 1. Alternatively, if you start with lists of size 1, you can only double their sizes log(n) times until you get a list of size n. Either way, there are log(n) levels and we do n comparisons on each level, so that gives us n*log(n) comparisons for the entire sort.

Result: MergeSort uses no more than n*log(n) comparisons to sort n items and runs in time Θ(n*log(n)).

Implementing MergeSort is not hard.

First, the recursive version needs more parameters than just the array being sorted, so the top-level call just sets up the arguments for the recursive call:

```
public static <E extends Comparable<? super E>>void
        MergeSort(E[] array ) {
                Object[] temp = new Object[array.length];
                MergeSort(array, 0, array.length-1, temp);
}
```

The arguments are:

- The array being sorted.
- The first and last indices of the region being sorted.
- A temporary array to help with the merging. By passing this as one of the arguments we avoid the expense of allocating arrays every time we recurse.

As we said, the recursive MS method splits the region being sorted into two pieces, recursively sorts them, and then merges them back together:

```
private static <E extends Comparable<? super E>> void
        MergeSort(E[] A, int first, int last, Object[] temp) {
                if (first < last) {
                        int mid = (first+last)/2;
                        MergeSort(A, first, mid, temp);
                        MergeSort(A, mid+1, last,temp);
                        merge(A, first, mid, last, temp);
                }
}
```

The merge method has more code but is conceptually quite simple. We have two consecutive regions of the array; both are sorted and we want to merge them into one sorted list.

We keep and index variable at the next element of each portion. We compare the values stored at these two indices and copy the smaller of them into the next open slot in the temp array.

This continues until we run out of one portion or the other. When that happens we just add the remaining portion to the temp array. Finally, after everything is merge onto temp in the correct order, we copy temp back over the original data.

```
public static <E extends Comparable<? super E>> void
        merge(E[] A, int first, int mid, int last, Object[] temp) {
                int p1 = first;
                int p2 = mid+1;
                int p = first;
                while (p1 <=mid && p2 <= last) {
                        if (A[p1].compareTo(A[p2]) < 0){
                                temp[p] = A[p1];
                                p1 += 1;
                        }
                        else {
                                temp[p] = A[p2];
                                p2 += 1;
                        }
                        p += 1;
                }
```

```
// Here is the "cleanup" portion of the merging, after we have run
// out of elements in one of the two arrays being merged:

        while (p1 <= mid) {
                temp[p] = A[p1];
                p1 += 1;
                p += 1;
        }
        while(p2 <= last) {
                temp[p] = A[p2];
                p2 += 1;
                p += 1;
        }
        for (int i = first; i <= last; i++)
                A[i] = (E) temp[i];
}
```