MERGE SORT

THIS FOLLOWS THE DIVIDE AND CONQUER APPROACH TO CREATE SMALLER SUB-PROBLEMS

A LIST IS BROKEN DOWN INTO SMALLER AND SMALLER PARTS RECURSIVELY

AT SOME POINT THERE WILL BE A LIST OF LENGTH ONE

WE CAN CONSIDER THAT A SORTED LIST

THEN MERGE THE SORTED LISTS TOGETHER TO GET THE FULLY SORTED LIST

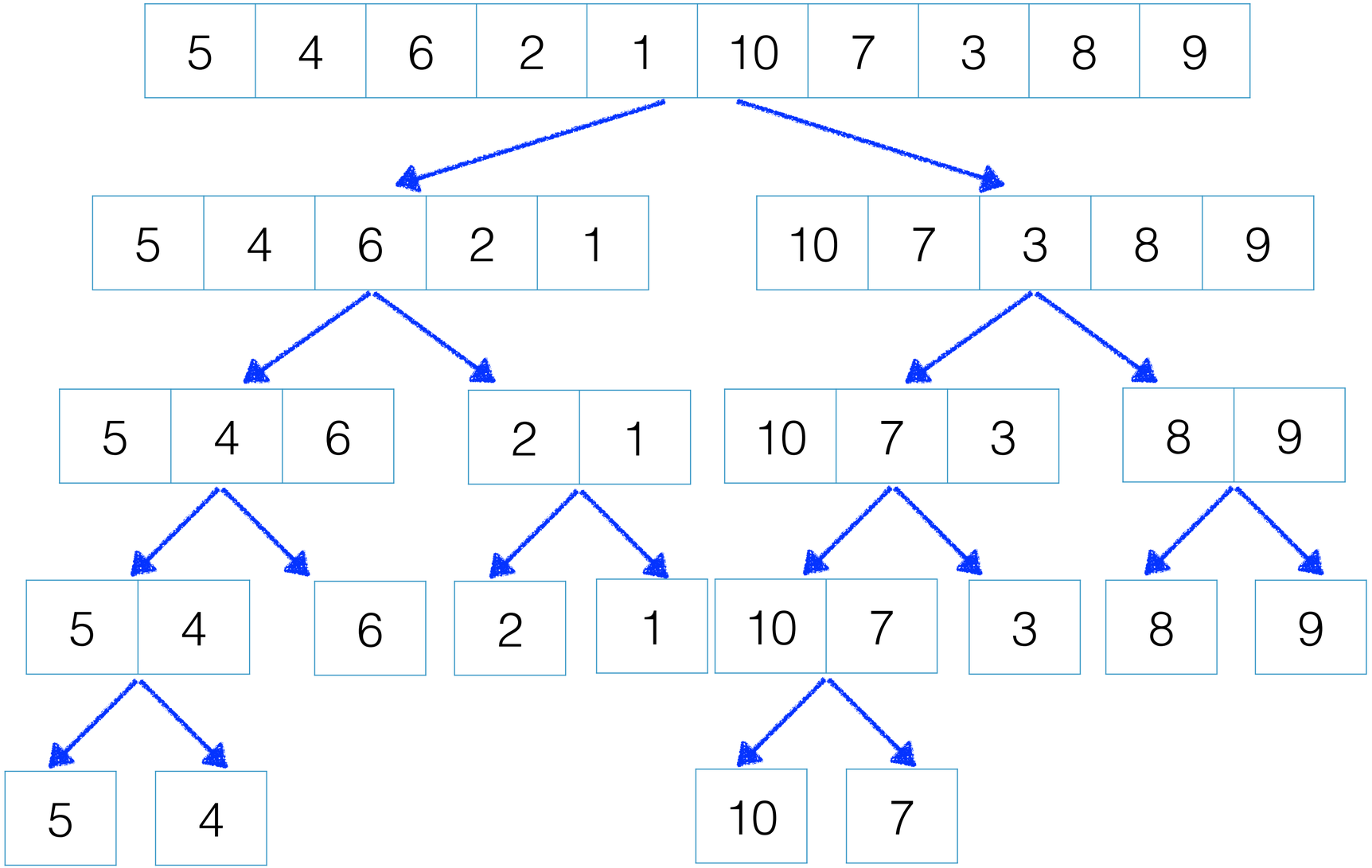
DIVIDE AND CONQUER - THE WEAPON? RECURSION

THIS IS A CLASSIC RECURSION BASED ALGORITHM, DIVIDE TILL THE PROBLEM IS SO SMALL AS TO BE TRIVIAL

SOLVE FOR THE TRIVIAL CASE AND THEN BUILD UP THE COMPLETE SOLUTION AS RECURSION UNWINDS

RECURSION CAN SEEM LIKE MAGIC, SO MAKE SURE YOU UNDERSTAND RECURSION BEFORE TACKLING THIS

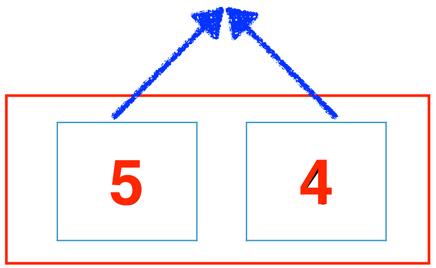
MERGE SORT

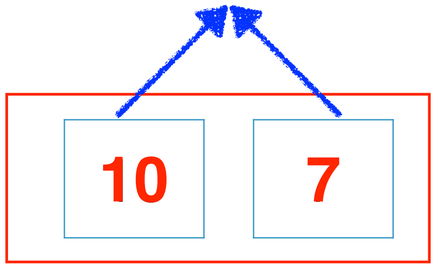


MERGE SORT

MERGE SORT

MERGE THE LISTS
KEEPING THE SORTED
ORDER

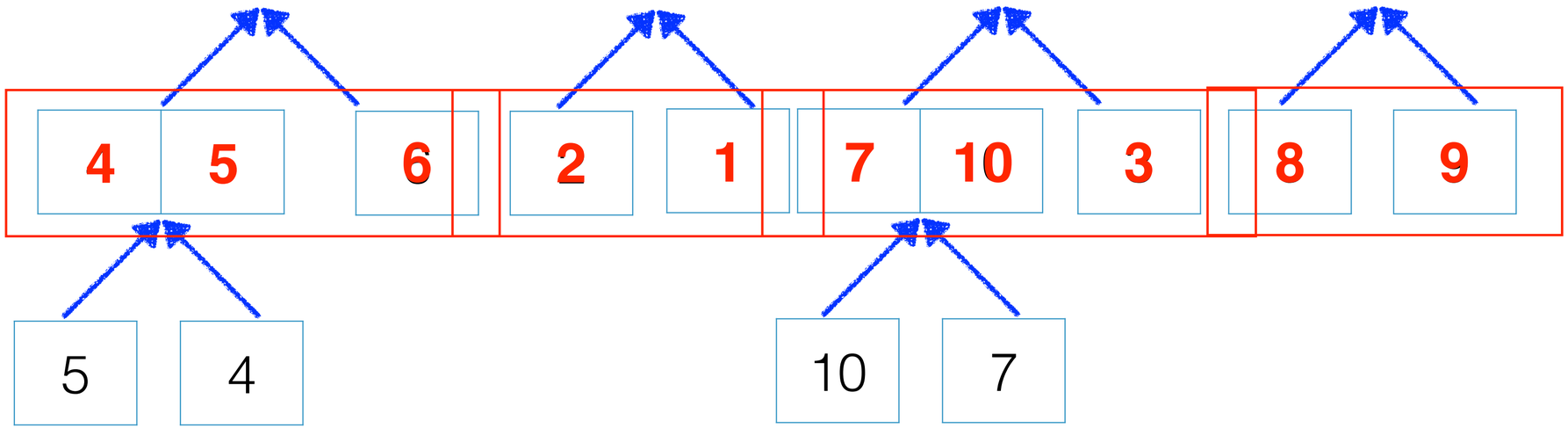




MERGE SORT

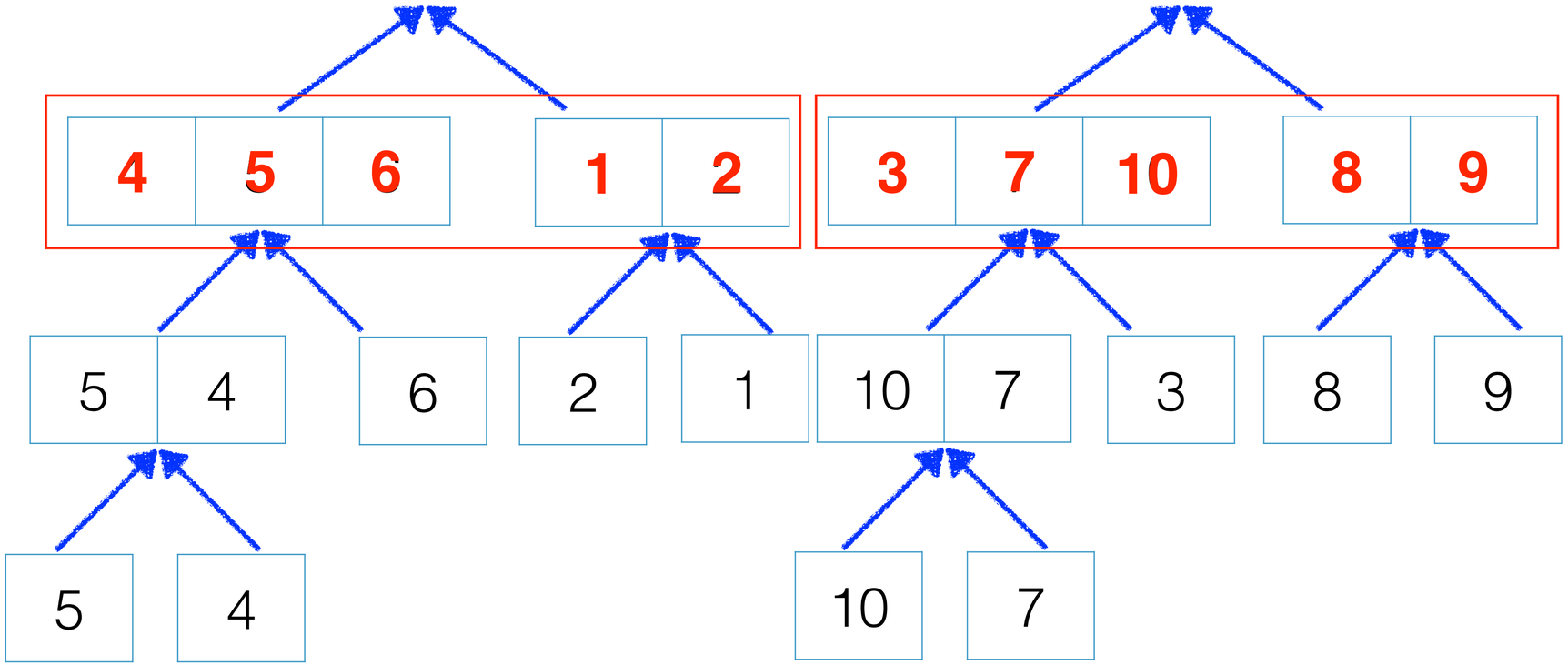
EACH LIST IS SORTED WHETHER IT IS A SINGLE ELEMENT LIST OR LISTS WITH 2 ELEMENTS

MERGE THE LISTS TOGETHER KEEPING THE SORTED ORDER



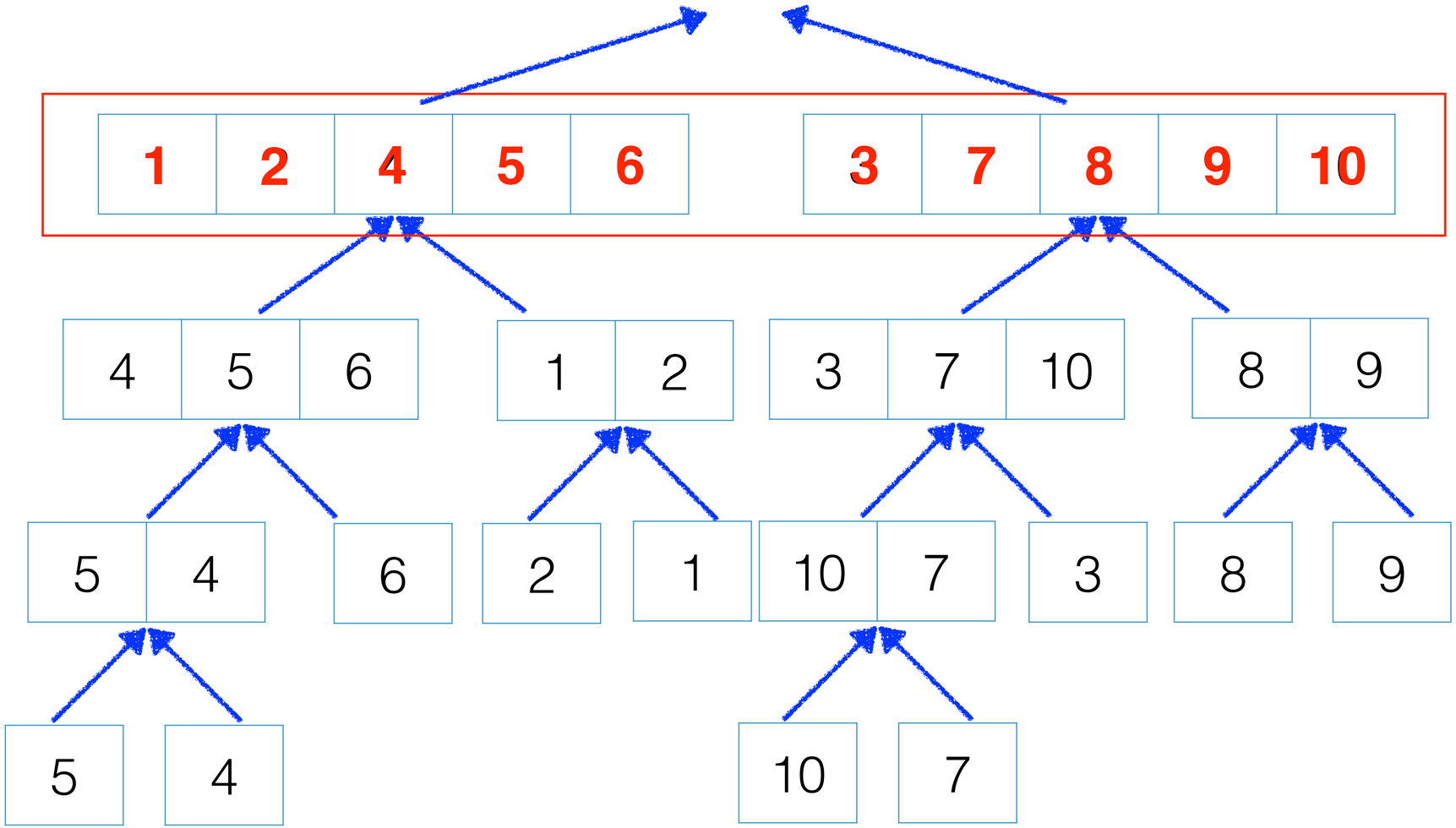
MERGE SORT

OUR LISTS ARE LONGER AND STILL SORTED, CONTINUE MERGING



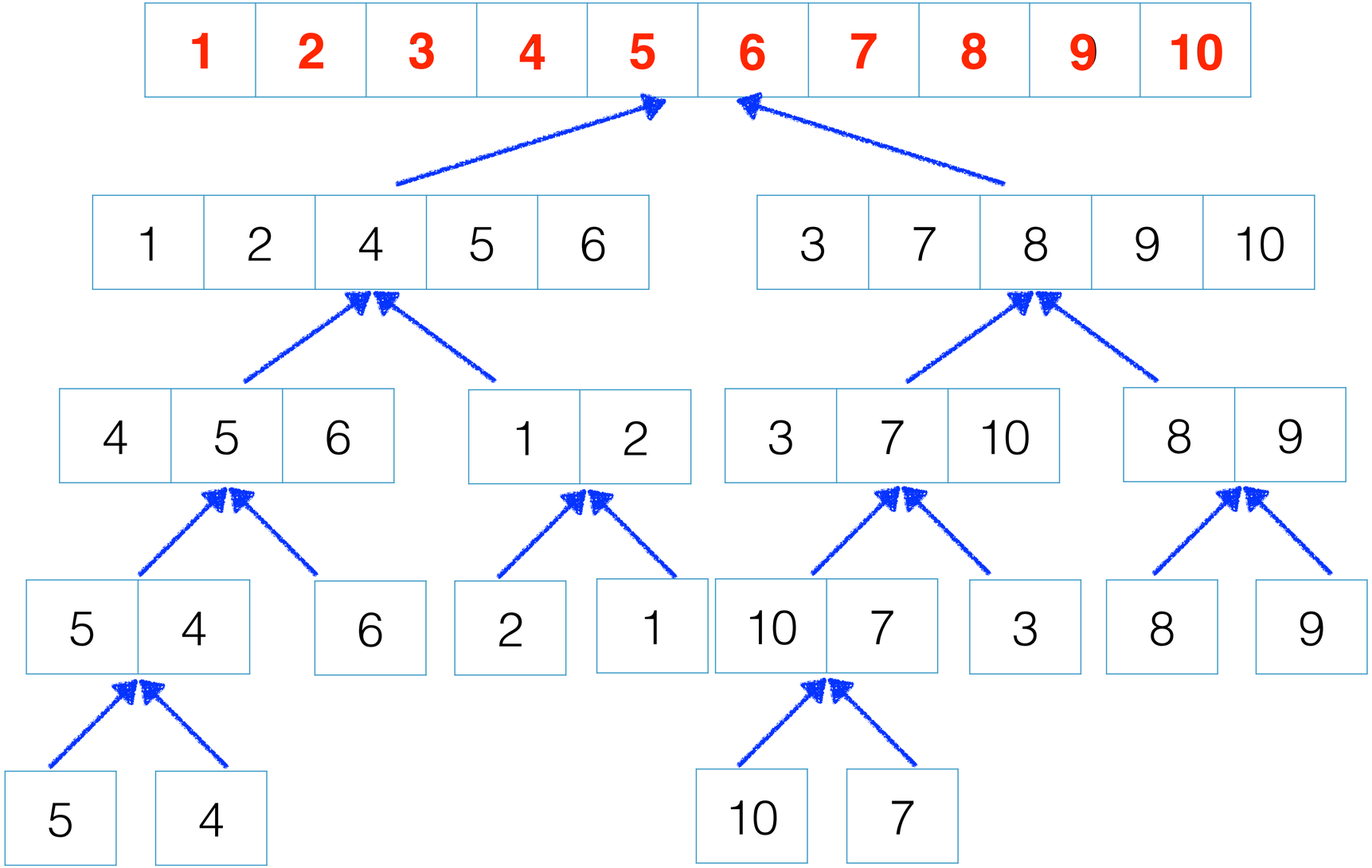
MERGE SORT

ONE FINAL MERGE TO GET THE ORIGINAL LIST IN SORTED ORDER



MERGE SORT

A SORTED LIST!



NOW FOR SOME CODE...

MERGE SORT USES A NUMBER OF HELPER METHODS

WE'LL SEE 2 FUNCTIONS IMPLEMENTED:

1: THE "SPLIT" METHOD TO SPLIT THE LIST INTO 2 SUB-LISTS

2: THE "MERGE" METHOD TO MERGE 2 SORTED LISTS INTO ONE SORTED LIST

3: THE "MERGESORT" METHOD WHICH DOES THE FINAL RECURSIVE SORT

THE "SPLIT"

THE TWO LISTS FOR THE FIRST AND SECOND HALVES HAVE BEEN SETUP AND SPLIT SIMPLY COPIES THE ELEMENTS FROM THE FIRST LIST OVER

```
public static void split(int[] listToSort, int[] listFirstHalf, int[] listSecondHalf) {
    int index = 0;
    int secondHalfStartIndex = listFirstHalf.length;
    for (int elements : listToSort) {
        if (index < secondHalfStartIndex) {
            listFirstHalf[index] = listToSort[index];
        } else {
            listSecondHalf[index - secondHalfStartIndex] = listToSort[index];
        }
        index++;
    }
}
```

THE SECOND HALF HOLDS THE REMAINDER OF THE ELEMENTS

THE FIRST HALF HOLDS EVERYTHING UP TO THE MID-POINT

THE "MERGE"

SET UP INDICES INTO THE FINAL MERGED LIST AND THE TWO HALVES WHICH ARE TO BE MERGED TOGETHER

```
public static void merge(int[] listToSort, int[] listFirstHalf, int[] listSecondHalf) {
    int mergeIndex = 0;
    int firstHalfIndex = 0;
    int secondHalfIndex = 0;

    while (firstHalfIndex < listFirstHalf.length && secondHalfIndex < listSecondHalf.length) {
        if (listFirstHalf[firstHalfIndex] < listSecondHalf[secondHalfIndex]) {
            listToSort[mergeIndex] = listFirstHalf[firstHalfIndex];
            firstHalfIndex++;
        } else if (secondHalfIndex < listSecondHalf.length) {
            listToSort[mergeIndex] = listSecondHalf[secondHalfIndex];
            secondHalfIndex++;
        }
        mergeIndex++;
    }

    if (firstHalfIndex < listFirstHalf.length) {
        while (mergeIndex < listToSort.length) {
            listToSort[mergeIndex++] = listFirstHalf[firstHalfIndex++];
        }
    }
    if (secondHalfIndex < listSecondHalf.length) {
        while (mergeIndex < listToSort.length) {
            listToSort[mergeIndex++] = listSecondHalf[secondHalfIndex++];
        }
    }
}
```

COPY OVER THE REMAINING ELEMENTS LEFT IN EITHER ONE OF THE LISTS

COMPARE THE ELEMENT AT THE CURRENT INDEX OF EACH OF THE LISTS AND CHOOSE THE SMALLER ONE TO GO INTO THE FINAL LIST

FINALLY "MERGESORT"

A LIST OF LENGTH 1 IS A SORTED LIST! THIS IS THE BASE CASE OF RECURSION

```
public static void mergeSort(int[] listToSort) {
    if (listToSort.length == 1) {
        return;
    }

    int midIndex = listToSort.length / 2 + listToSort.length % 2;
    int[] listFirstHalf = new int[midIndex];
    int[] listSecondHalf = new int[listToSort.length - midIndex];
    split(listToSort, listFirstHalf, listSecondHalf);

    mergeSort(listFirstHalf);
    mergeSort(listSecondHalf);

    merge(listToSort, listFirstHalf, listSecondHalf);
    print(listToSort);
}
```

GET THE MID-POINT AT WHICH TO SPLIT THE LIST

MERGE THE SORTED LIST TO GET THE ORIGINAL LIST IN SORTED ORDER

RECURSIVE CALL, MERGE SORT THE TWO SMALLER SUB-LISTS CREATED

MERGE SORT USES DIVIDE AND CONQUER TO CREATE SMALLER PROBLEMS WHICH ARE EASIER TO TACKLE

TO CALCULATE THE COMPLEXITY WE NEED TO CONSIDER THE RECURSIVE STEP WHERE THE PROBLEM IS DIVIDED INTO 2 AND THE MERGE OF TWO LISTS OF N/2 LENGTH

THE EXACT DERIVATION IS NOT REALLY RELEVANT TO PROGRAMMING INTERVIEWS

THE COMPLEXITY OF MERGE SORT IS O(NLOGN)

MERGE SORT IS NOT ADAPTIVE

IT TAKES O(N) EXTRA SPACE WHEN WE USE ARRAYS (ALL THE SMALLER LISTS WE CREATE IN THE DIVIDE PHASE)

IT IS A STABLE SORT