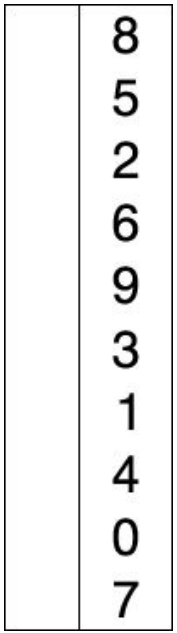
2023-03-22

10.2.2: Merge Sort

Review: Previous Sort Methods

Selection Sort:



Insertion Sort:

6 5 3 1 8 7 2 4

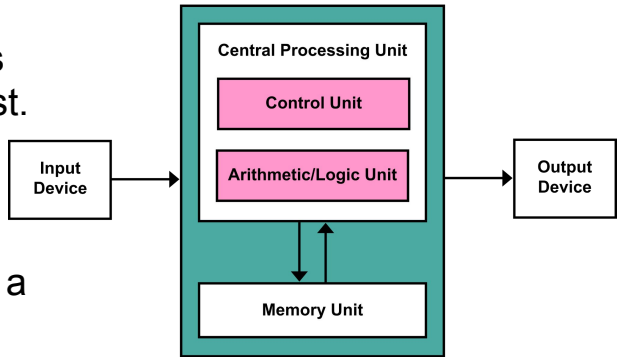
Merge Sort

Merge Sort is a **divide and conquer** sorting algorithm, much like Binary Search is a divide and conquer search algorithm.

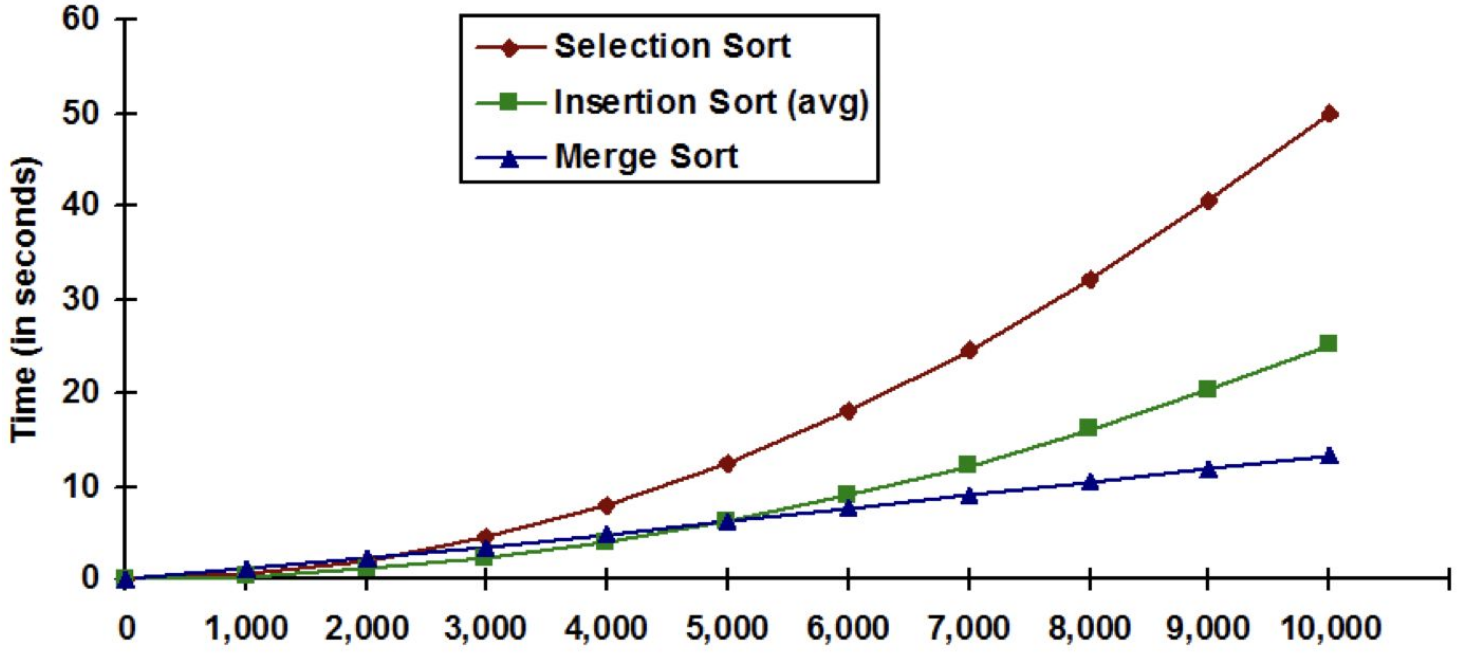
Merge Sort divides the input array in half, and recursively Merge Sorts the halves.

This algorithm was invented in 1945 (!) by John Von Neumann (1903-1957), a genius mathematician and early computer scientist.

Among other things, Von Neumann invented the Von Neumann Architecture, which is still the conceptual model for how a computer works.



Sorting Algorithm Efficiency



Merge sort worst case performance is O(n log n)

Ω(N log N)

A **lower bound** for a problem is the worst-case running time of the best possible algorithm for that problem.

"Big Omega" (Ω) is used to designate a lower bound.

A proof exists that the lower bound for comparison-based sorting is Ω(N log N).

That means that no algorithm, however smart, could possibly be faster, in the worst-case, then N log N comparisons.

Merge Sort: an O(N log N) sort

Merge Sort has O(N log N) running time for best case, worst case, and average case.

This means that Merge Sort is an efficient sorting algorithm. In the worst case, it's proven that one can't conceive a sorting algorithm that does less comparisons!

There are other factors in how well a sorting algorithm performs though: How much additional memory it needs, how well it uses computer cache memory, how it performs on different kinds of input data (randomly ordered, partially ordered, etc.)

Another popular divide and conquer sorting algorithm is Quicksort (1951), which is faster than Merge Sort in some situations and slower in others.

Merge Sort: O(N) Space Complexity

Merge Sort, as it is typically implemented, requires temporary space.

To do its merges, it needs an extra array the same size as the source.

There are ways to reduce the space overhead, but they come with trade-offs, such as making the algorithm more complex or increasing the amount of data copied.

It can actually be reduced to O(1) using linked lists, which we saw on Monday!

Merge Sort: Copying Data

There's copying of data between the main array and the temporary array.

This can be minimized, but at the cost of making the algorithm more complicated.

The code given in CS Awesome just copies the data a lot... that is fine and that's what we'll work through, for simplicity.

The Wikipedia article gives fancier code that avoids the copying.

Merge Sort: A Stable Sort

A sort algorithm is said to be stable if it always gives the same results.

When would a sort algorithm ever not give the same result??

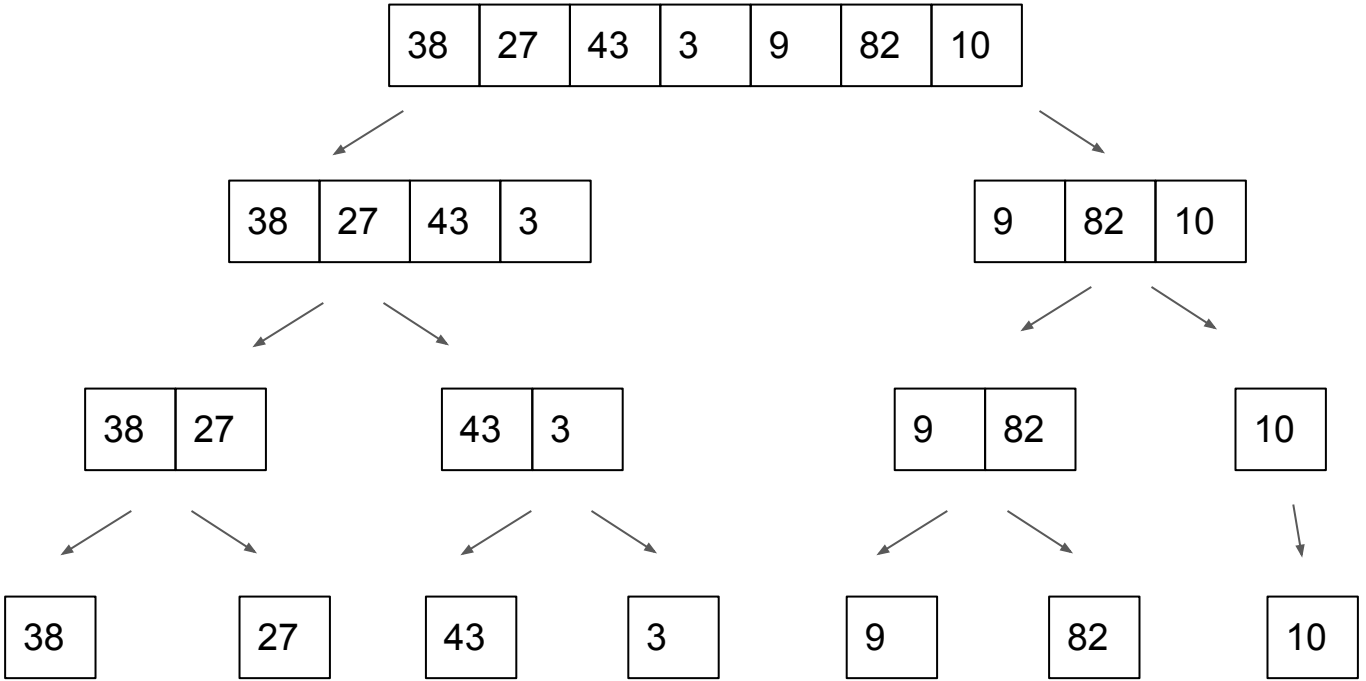
When there are duplicate keys, multiple sorted orderings of the same input are possible.

Why is a stable sort important? Sometimes, it's not. But the boss at a big company may not like it if you run the same report twice and get ever-so-slightly different results!

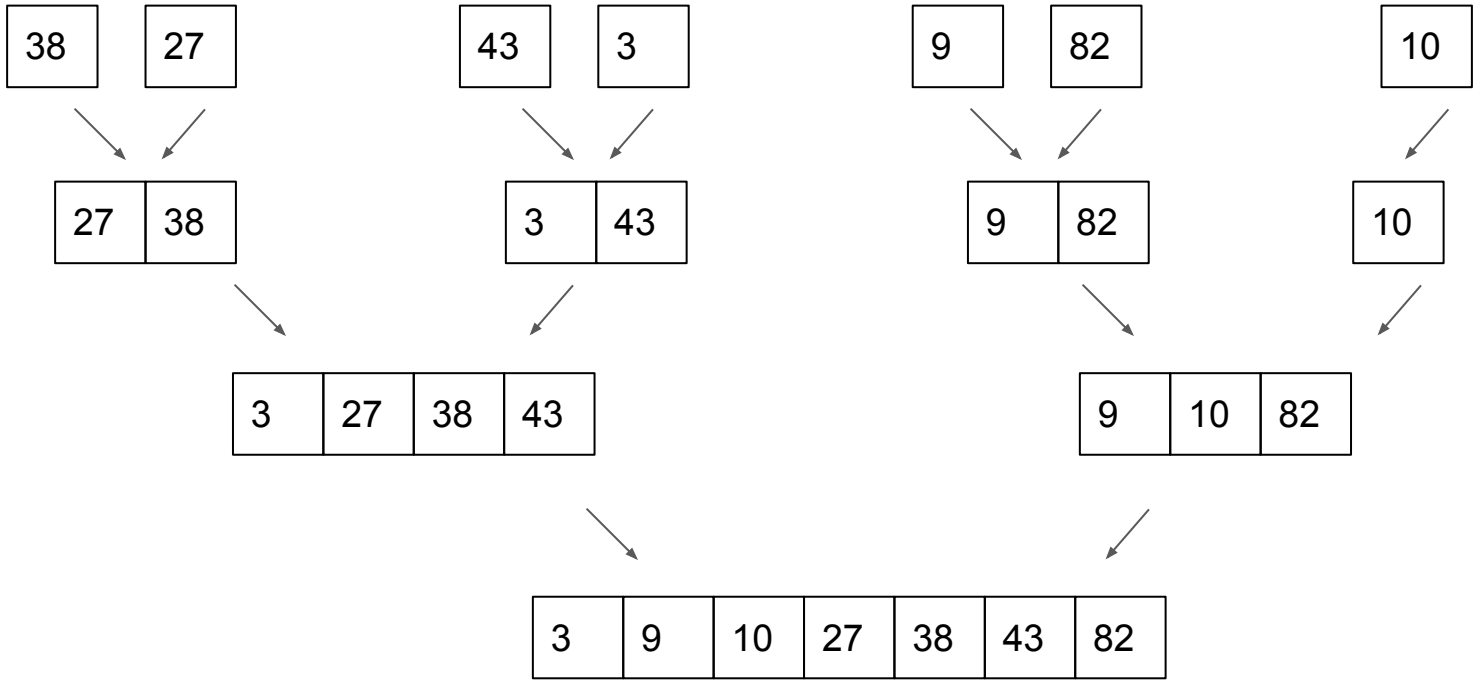
Name (key)	Score
Jill Student	100
Joe Student	96
Joe Student	95

Name (key)	Score
Jill Student	100
Joe Student	95
Joe Student	96

Merge Sort: Split the array into halves down to one element



Merge Sort: Sort the halves and merge them together



Merge Sort Pseudocode

1. If the list's size is 0 or 1, just return the original list (as it is sorted)
2. Split the list parameter int two lists of roughly equal size
3. Recursively Merge Sort both split lists, list 1 and list 2
4. Merge the two sorted lists and return the result

Merge Sort "Lists"

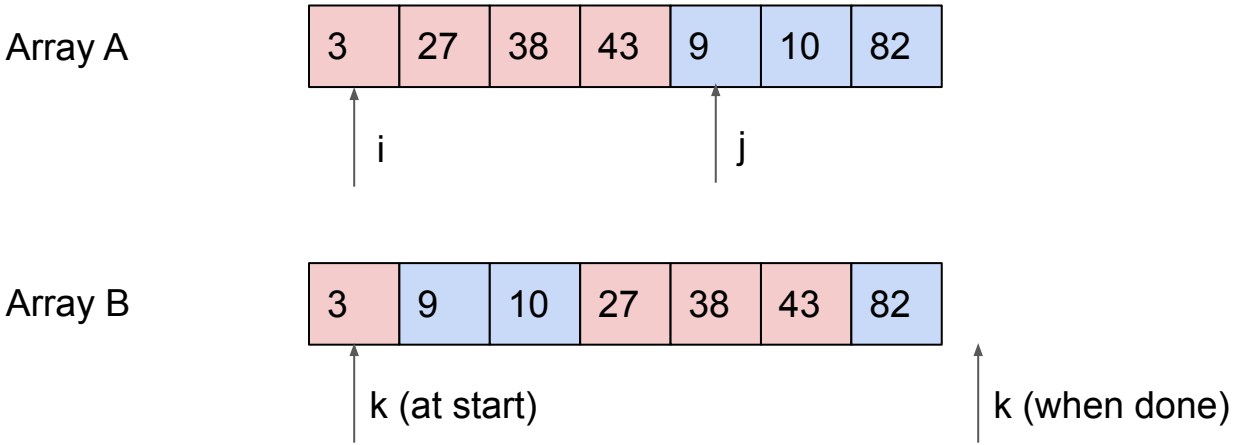
Conceptually, you can think of Merge Sort as splitting the input into a bunch of smaller sub-lists, and recursively merging all the sub-lists together.

In practice, we don't want to create tons of lists! We want to allocate as little memory as possible, and copy as little data as possible. So we want to work as much as possible in the array being sorted.

This means that a "sub-list" is really a "view" into the original array, tracked with two indices, begin and end.

Because of Merge Sort's merge operation, we do need some temporary storage, but that comes out to one additional temporary array that is the same size as the array being sorted.

Merge Sort: The Merge

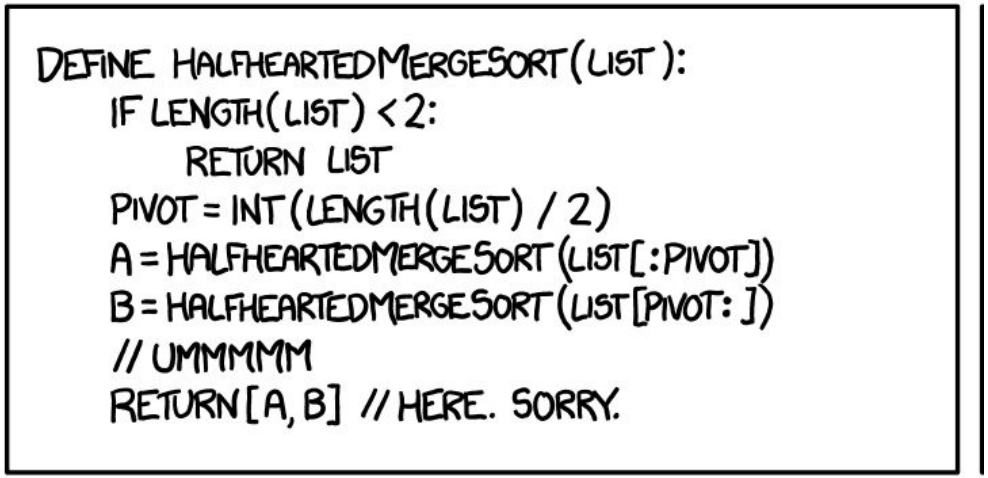


i = current index in left half, j = current index in right half. k = index in output array.

While i <= ending index of left half and j <= ending index of right half:
  B[k++] = smaller of A[i] and A[j].
  Advance i if we used A[i], or advance j if we used A[j].

After, one of the halves may still have something left... output anything left over into B.

Show me the code!



https://xkcd.com/1185/

CSAwesome's Merge Sort (edited for brevity) [1/2]

```
public static void mergeSort(int[] elements) {
   int n = elements.length;
   int[] temp = new int[n];
   mergeSortHelper(elements, 0, n - 1, temp);
}

private static void mergeSortHelper(int[] elements,
                                    int from, int to, int[] temp) {
    if (from < to) {
       int middle = (from + to) / 2;
       mergeSortHelper(elements, from, middle, temp);
       mergeSortHelper(elements, middle + 1, to, temp);
       merge(elements, from, middle, to, temp);
    }
}
```

CSAwesome's Merge Sort implementation [2/2]

```
private static void merge(int[] elements, int from, int mid, int to, int[] temp) {
   int i = from, j = mid + 1, k = from;

   while (i <= mid && j <= to) {
      if (elements[i] < elements[j]) {
         temp[k++] = elements[i++];
      } else {
         temp[k++] = elements[j++];
      }
   }

   while (i <= mid) {
      temp[k++] = elements[i++];
   }

   while (j <= to) {
      temp[k++] = elements[j++];
   }

   for (k = from; k <= to; k++) {
      elements[k] = temp[k];
   }
}
```

Here, we're comparing the current elements from each half, and taking the smaller one. We only advance the pointer for the side we took from!

If we got here, one or both halves are done. Output anything left over from the left half.

Output anything left over from the right half. (Only one of these loops will run.)

Copy everything back from the temp array. (This can be avoided with some extra fanciness.)

CSAwesome's Merge Sort implementation [2/2]

```
private static void merge(int[] elements, int from, int mid, int to, int[] temp) {
   int i = from, j = mid + 1, k = from;

   while (i <= mid && j <= to) {
      if (elements[i] < elements[j]) {
         temp[k++] = elements[i++];
      } else {
         temp[k++] = elements[j++];
      }
   }

   while (i <= mid) {
      temp[k++] = elements[i++];
   }

   while (j <= to) {
      temp[k++] = elements[j++];
   }

   for (k = from; k <= to; k++) {
      elements[k] = temp[k];
   }
}
```


Here, we're comparing the current elements from each half, and taking the smaller one. We only advance the pointer for the side we took from!

If we got here, one or both halves are done. Output anything left over from the left half.

Output anything left over from the right half. (Only one of these loops will run.)

Copy everything back from the temp array. (This can be avoided with some extra fanciness.)

CSAwesome's Merge Sort implementation [2/2]

```
private static void merge(int[] elements, int from, int mid, int to, int[] temp) {
   int i = from, j = mid + 1, k = from;

   while (i <= mid && j <= to) {
      if (elements[i] < elements[j]) {
         temp[k++] = elements[i++];
      } else {
         temp[k++] = elements[j++];
      }
   }

   while (i <= mid) {
      temp[k++] = elements[i++];
   }

   while (j <= to) {
      temp[k++] = elements[j++];
   }

   for (k = from; k <= to; k++) {
      elements[k] = temp[k];
   }
}
```


Here, we're comparing the current elements from each half, and taking the smaller one. We only advance the pointer for the side we took from!

If we got here, one or both halves are done. Output anything left over from the left half.

Output anything left over from the right half. (Only one of these loops will run.)

Copy everything back from the temp array. (This can be avoided with some extra fanciness.)

CSAwesome's Merge Sort implementation [2/2]

```
private static void merge(int[] elements, int from, int mid, int to, int[] temp) {
   int i = from, j = mid + 1, k = from;

   while (i <= mid && j <= to) {
      if (elements[i] < elements[j]) {
         temp[k++] = elements[i++];
      } else {
         temp[k++] = elements[j++];
      }
   }

   while (i <= mid) {
      temp[k++] = elements[i++];
   }

   while (j <= to) {
      temp[k++] = elements[j++];
   }

   for (k = from; k <= to; k++) {
      elements[k] = temp[k];
   }
}
```

Here, we're comparing the current elements from each half, and taking the smaller one. We only advance the pointer for the side we took from!

If we got here, one or both halves are done. Output anything left over from the left half.

Output anything left over from the right half. (Only one of these loops will run.)

Copy everything back from the temp array. (This can be avoided with some extra fanciness.)

CSAwesome's Merge Sort implementation [2/2]

```
private static void merge(int[] elements, int from, int mid, int to, int[] temp) {
   int i = from, j = mid + 1, k = from;

   while (i <= mid && j <= to) {
      if (elements[i] < elements[j]) {
         temp[k++] = elements[i++];
      } else {
         temp[k++] = elements[j++];
      }
   }

   while (i <= mid) {
      temp[k++] = elements[i++];
   }

   while (j <= to) {
      temp[k++] = elements[j++];
   }

   for (k = from; k <= to; k++) {
      elements[k] = temp[k];
   }
}
```

Here, we're comparing the current elements from each half, and taking the smaller one. We only advance the pointer for the side we took from!

If we got here, one or both halves are done. Output anything left over from the left half.

Output anything left over from the right half. (Only one of these loops will run.)

Copy everything back from the temp array. (This can be avoided with some extra fanciness.)

Avoiding the copying

Here's the extra fanciness...

This is the implementation in the Wikipedia article for Merge Sort.

It avoids the copying of data that the CS Awesome algorithm does, by alternately merging into array A or array B.

It does do a one-time copy at the beginning of the sort.

```
// Array A[] has the items to sort; array B[] is a work array.
void TopDownMergeSort(A[], B[], n)
{
    CopyArray(A, 0, n, B);           // one time copy of A[] to B[]
    TopDownSplitMerge(B, 0, n, A);   // sort data from B[] into A[]
}

// Split A[] into 2 runs, sort both runs into B[], merge both runs from B[] to A[]
// iBegin is inclusive; iEnd is exclusive (A[iEnd] is not in the set).
void TopDownSplitMerge(B[], iBegin, iEnd, A[])
{
    if (iEnd - iBegin <= 1)                     // if run size == 1
        return;                                 //   consider it sorted
    // split the run longer than 1 item into halves
    iMiddle = (iEnd + iBegin) / 2;              // iMiddle = mid point
    // recursively sort both runs from array A[] into B[]
    TopDownSplitMerge(A, iBegin,  iMiddle, B);  // sort the left  run
    TopDownSplitMerge(A, iMiddle,    iEnd, B);  // sort the right run
    // merge the resulting runs from array B[] into A[]
    TopDownMerge(B, iBegin, iMiddle, iEnd, A);
}

//  Left source half is A[ iBegin:iMiddle-1].
// Right source half is A[iMiddle:iEnd-1   ].
// Result is            B[ iBegin:iEnd-1   ].
void TopDownMerge(A[], iBegin, iMiddle, iEnd, B[])
{
    i = iBegin, j = iMiddle;

    // While there are elements in the left or right runs...
    for (k = iBegin; k < iEnd; k++) {
        // If left run head exists and is <= existing right run head.
        if (i < iMiddle && (j >= iEnd || A[i] <= A[j])) {
            B[k] = A[i];
            i = i + 1;
        } else {
            B[k] = A[j];
            j = j + 1;
        }
    }
}

void CopyArray(A[], iBegin, iEnd, B[])
{
    for (k = iBegin; k < iEnd; k++)
        B[k] = A[k];
}
```

Your turn!

Go through the exercises for chapter **10.2.2 Merge Sort** in CSAwesome

Other than that, one of the best exercises is to work through a merge sort on paper. Just like we did on the whiteboard, make an array of ~10 elements and merge sort it yourself.