Last time we talked about a simple sorting algorithm: selection sort. The algorithm is simple, however it is extremely inefficient: its runtime scales with the square of the input size. Today we will use a more indirect technique to derive a better sorting algorithm.

Suppose that, instead of doing the work yourself, you asked two of your friends to help you sort the array of N cards. You cut the deck in half and give one half to each friend. Each of them goes somewhere, sorts the cards, and returns their individual stacks back to you.

You now have two stacks of sorted cards, but what you really need is one sorted array. However, this operation can be done in linear time (where the runtime is proportional to the array sizes). Arrange the two decks such that you scan each of them from lowest to highest.

Think of it like placing them so the lowest cards are on top. Now, you can look at the top card of each stack. Whichever one is lower, you place that onto a third stack to the side.

Over time, you will slowly exhaust the two subarrays and combine them into a third singular array in linear time (linear time is often denoted O(n)), each step places the lower of two topmost cards onto the new pile. But how do we know this always works?

Once again the proof is by induction. In the first step, the lowest card is one of the two top ones, namely the lower top card. That is the first card inserted into the final array. Now, suppose the lowest k cards were already sorted.

This means that the highest (N - k) cards are still unsorted. It can be realized that the next card, the (k + 1)th lowest card, is once again on the top of one of the two stacks, since otherwise there would be a card lower than it. It must also be the lowest visible (top) card.

Therefore by induction the lower of the two top cards on the stacks are inserted into the array each time. In code, you don't need to use a stack structure. Instead, you can use two pointers on the arrays that start low and move upwards.

Each step, you find the lower of the two values at the pointers, place that value into the final array, and increment the corresponding pointer. When one pointer reaches the end, all that's left to do is exhaust the other array.

The implementation of this merge function is as follows:

```
MERGE(A, B): // Merges A and B, assuming both are sorted (0-indexed)
    a = 0 // The pointers
    b = 0
    c = 0
    C = new array[A.length + B.length]
    while (c < C.length)
        if (a ≥ A.length) // If one array is done do the other
            C[c++] = B[b++] // Notice how "used" ptrs are incremented
            continue
        else if (b ≥ B.length)
            C[c++] = A[a++]
            continue
        // Do the thing
        if (A[a] < B[b]) C[c++] = A[a++]
        else C[c++] = B[b++]
    endwhile
    return C
```

Now we know that we can combine two sorted arrays in O(n) linear time. But what about sorting the two arrays themselves?

Well, the two friends that you tasked with sorting the two arrays, they themselves found two more friends each and asked those friends to sort half of their arrays. And those (2nd-degree) friends asked more (3rd-degree) friends to sort their arrays in halves. And so on. Some (d-th degree) friends saw that their array was small enough, and sorted it directly.

In the same way, the code that does a merge sort can sort the array recursively:

```
MERGE-SORT(A):
    X = Left half of A
    Y = Right half of A
    // Sort the array
    X = MERGE-SORT(X)
    Y = MERGE-SORT(Y)
    A = MERGE(X, Y)
    return A
```

It splits the array in half, sorts each one using this same method, and linearly merges the sorted subarrays. But what is its runtime? Well in this case, we cannot simply add a few things up.

Since this method calls a smaller version of itself a few times, the runtime must now be measured in terms of itself. We know that MERGE is $O(n)$ runtime. Let $T(n)$ be the runtime of MERGE-SORT on a size n array.

Therefore, going line by line, we see that $T(n) = T(n/2) + T(n / 2) + O(n) = 2T(n / 2) + O(n)$. $T(n)$ is now recursively defined.

For now we shall leave this discussion here, and in the next article we will prove that $T(n)$ is more efficient than $O(n^2)$.