This lecture explains the ninth homework, merge sort.

The merge sort has two phases: the division phase and the merge phase. In the first phase, merge sort divides an array into two halves. Divisions continue until each array has only one element or no element. In either case, there is no need to sort.

Then, merge sort enters the merge and sort phase. In this phase, merge sort takes two sorted array and merges them into one sorted array.

Let me use an example to explain how this works.

Consider an array of these elements. In the first division phase, the array is divided into two halves. In this example, the array has 8 elements and the two halves have the same number of elements. If the array has an odd number of elements, that’s all right. One half may have one more element than the other. That’s not a problem at all. The algorithm still works.

The algorithm continues dividing the arrays. The two arrays now becomes four arrays.

Division continues until each array has only one element or no element. If an array has no element, obviously, nothing needs to be done. If an array has only one element, this array is already sorted.

This is how division can be done. Consider this function for merge sort. The function has three arguments: the address of an array, the indexes for the left and right boundaries of the array.

The stop conditions check whether there is no element or only one element. In either case, sorting is already done. Otherwise, the function calculates the index in the middle. The function continues dividing the array into two halves. Please notice that the starting index for the second half is M plus 1. If you use M without plus one, the element at the middle of the array will appear in both arrays and the result will be wrong.

After the division phase, the program enters the merge phase.

How does merge work?

Merge takes two arrays and sorts them. For example, if one array has 9 and the other array is 5, the result is an array starting with 5 and the second element is 9. Similarly, if an array has 2 and the other array has 4, the result is 2 at the beginning and 4 as the second element.

The next step merges and sorts two arrays, each with two elements. The first array has 5 and 9. The second array has 2 and 4. The result is an array with 2, 4, 5, and 9.

At the other side of this example, the first array has 3 and 7. The second array has 1 and 6. The result is an array of 1, 3, 6, and 7.

The next step merges the two arrays, each with 4 elements. The final result is a sorted array.

Merge sort also uses transitivity. Let me remind you of transitivity: If ei is greater than bee and bee is greater than cee, ei must be greater than cee. It is unnecessary comparing ei and cee.

How does merge sort use transitivity?

It compares the first element in each array and picks the smaller one. In this example, merge sort compares 2 and 1. 1 is smaller and 1 becomes the first element in the final array. Please notice that merge sort does not compare 1 with 4 because this comparison is unnecessary.

After putting 1 as the first element in the final array, the program now compares 2 with 3. 2 is smaller and it becomes the second element in the final array. Please notice that 2 is not compared with 6 because this comparison is unnecessary.

Next, the program compares 3 with 4 and selects 3 as the third element in the final array.

Let me explain this further by drawing the three arrays: the first half, the second half, and the result.

We need to keep track of three indexes for these three arrays.

Let’s call these indexes X, Y, and Z. .

If the first array’s element is greater than the second array’s element, then the second array’s element is copied to the destination array. The index for the second array, namely Y, increases by one. The index for the destination array, namely Z, also increases by one.

If the first array’s element is smaller than the second array’s element, then the first array’s element is copied to the destination array. The index for the first array, namely X, increases by one. The index for the destination array, namely Z, also increases by one.

This process continues. If the first array’s element is greater than the second array’s element, then the second array’s element is copied to the destination array. The index for the second array, namely Y, increases by one. The index for the destination array, namely Z, also increases by one.

We have learned several sorting algorithms, including selection sort, quick sort, and merge sort.

Which one is better? Is there a sorting algorithm that is the best?

Actually, there is no best sorting algorithm. Depending on the evaluation metrics, some algorithms are better than the others. There are many different ways to evaluate sorting algorithms, for example. You may evaluate algorithms by the number of comparisons. This is the most commonly used method to evaluate sorting algorithms. If you use this method for evaluation, quick sort and merge sort are better than selection sort because quick sort and merge sort use the principle of transitivity.

Another method for evaluation considers how many times data gets moved. For selection sort, each number gets moved at most once. This is better than quick sort.

A property of a sorting algorithm is whether it is stable. Stability means the sorting algorithm preserves the order if two values are the same. Why is this important? Imagine that you want to sort data in a spreadsheet. You want to sort the data according to one column first and then another column. In this case, stability is important. When the program sorts the second column and two rows have the same value, then the sorting algorithm should preserve the order set by sorting the first column.

Sometimes, we care more about the best case scenarios. If the data is already sorted, what will the algorithm do? If the data is already reversely sorted, in other words the data is sorted by the descending order, what will the algorithm do? Sometimes we care more about the average case.

Sometimes, the data may be almost sorted already. What will the sorting algorithm do?

In some cases, we need to worry about the situation when the data is too large and cannot fit into memory. Reading data from hard disks is much slower than reading data from memory. Thus, if data cannot fit in memory, the priority is to reduce the number of disk accesses.

In some cases, the data may not even be stored on the same machine. The data may be distributed over multiple machines. Getting data from networks is even slower than disks.

To summarize, no sorting algorithm is better than all the other sorting algorithms when these metrics are used. Some algorithms are better in some scenarios and worse in some other scenarios. Thus, we have to learn several sorting algorithms.

If you want to become an expert in sorting, take a look of this book: The Art of computer Programming by Donald Knuth.

In case you do not know Donald Knuth, do a web search. You need to know him. His work forms the foundation of many things you have learned or will learn.