

# Sorting with merge sort

Lecturer: Dr Anwarul Patwary

# Outline

We discuss:

- ▶ What is the idea behind merge sort?
- ▶ How is it implemented?
- ▶ How does merge sort compare with other popular sorting algorithms?

# Merge sort

- ▶ Merge sort is based on the idea that
  - ▶ it's easier to sorting short lists than long lists
  - ▶ if we have two sorted lists, it's easy to “merge” them into a longer sorted list.
- ▶ For instance, suppose we have the following two (very short) sorted lists:  $[2, 5]$  and  $[3, 4]$ .
  - ▶ It is easy to see that we can *merge* these into a longer sorted list:  
 $[2, 3, 4, 5]$

# Merging lists

- ▶ We will consider the process of *merging* sorted lists first.
- ▶ Suppose we have the following two sorted lists:  
[1, 3, 4, 7] and [2, 5, 6, 8].
- ▶ It might seem easy to us to know how to merge these – but we need to express it in a form the computer can use.

# Merging lists

- ▶ The idea is: we start at the front of each list; since they're sorted, we know *one* of those elements will be the first element in the merged list.
- ▶ Then, we “step” along each list, deciding from which list we will draw the next element to go in our merged list.
  - ▶ One incorrect idea is to just alternate between the lists – but that won't work.
  - ▶ If you try this for the lists above, you'll see it works for the first three elements, then produces a wrong result.

## Merging lists

- ▶ Let's try this for the two lists above ...

# Complexity of merging

- ▶ What is the big 'O' complexity of merging two sorted lists?
- ▶ We can describe it in terms of the length of the *output* list.
  - ▶ (Note that in previous cases, we have described complexity in terms of the *input* data set – but we can use whatever is convenient.)
- ▶ If the final list contains  $n$  numbers – we couldn't have done *more* than  $n$  comparisons.
  - ▶ So – we need at most  $n$  comparisons to correctly merge the two lists;
  - ▶ And therefore the merge algorithm runs in *linear* time, or has  $O(n)$ .

# Sorting in merge sort

- ▶ Merge sort is what is called a “divide and conquer” algorithm – it breaks a big problem into many much smaller problems.
- ▶ The major parts of the merge sort algorithm can be described as follows.
- ▶ To sort an array using a method `mergeSort`:
  - ▶ split the array in two, at the middle. (If we have an array with an odd number of elements, that's fine – we can arbitrarily decide which of the two sub-lists will be the bigger.)
  - ▶ recursively call `mergeSort` on each of the lists.
  - ▶ merge the two lists.



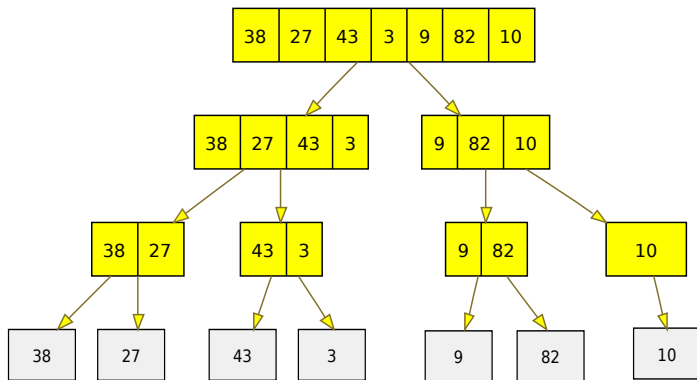
## Sorting in merge sort

Let's see how merge sort works when applied to the following list:

38	27	43	3	9	82	10
----	----	----	---	---	----	----

## Sorting in merge sort

First, the recursive calls to `mergeSort` will split the array into half, each time:<sup>1</sup>



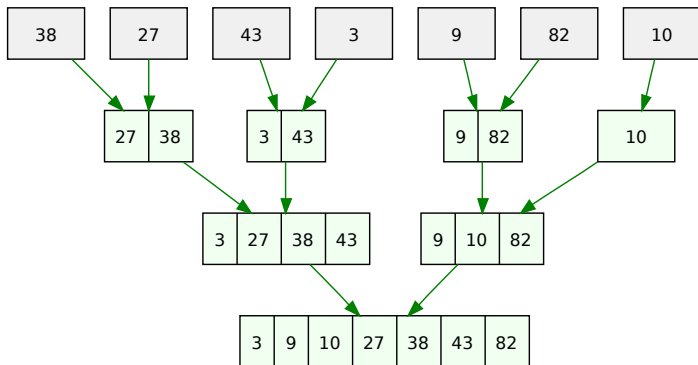
And eventually, we have a set of “lists” which only contain a single element; which means each of them can be considered sorted.

---

<sup>1</sup>Image adapted from [Wikimedia Commons](#) diagram by Vineet Kumar.

## Merging in merge sort

And once our initial list has been split into length-1 lists, it is easy to apply the “merge” algorithm to merge them:



So we end up with a completely sorted list.

## Java code for merge sort

The overall merge sort algorithm is implemented like this, in `SortingAlgorithms.java`:

```
public static void mergeSort(short[] arr) {  
    mergeSort(arr, 0, arr.length-1);  
}  
  
private static void mergeSort(short[] arr, int l, int r) {  
    if (l < r) {  
        int mid = (l + r) / 2;  
        mergeSort(arr, l, mid);  
        mergeSort(arr, mid + 1, r);  
        merge(arr, l, mid, r);  
    }  
}
```

We will discuss this implementation.

## Java code for merging

The Java code for merging is a little more complicated.

- ▶ First, we will *copy* the elements from our input lists, into two new arrays we create for this purpose.
- ▶ Then when we do the “merge”, we can write the results of the merge into the array `arr` (our original array).

```
private static void merge(short[] arr, int l, int mid, int r) {  
    int lsize = mid - l + 1;  
    int rsize = r - mid;  
    short[] left = new short[lsize];  
    short[] right = new short[rsize];  
  
    for (int i = 0; i < lsize; i++) {  
        left[i] = arr[l + i];  
    }  
    for (int j = 0; j < rsize; j++) {  
        right[j] = arr[mid + 1 + j];  
    }  
}
```

## Java code for merging

The code which actually does the merging is below.

- ▶ **while** there are still elements in *both* lists, we copy an element from either the left or the right list.
- ▶ Once this is done – there may be “left over” elements in one of the lists.

So we copy those as well.

```
int i = 0; int j = 0; int k = 1;
while (i < lsize && j < rsize) {
    if (left[i] < right[j]) {
        arr[k++] = left[i++];
    } else {
        arr[k++] = right[j++];
    }
}
while ( i < lsize ) { // Copy rest of first half
    arr[k++] = left[i++];
}
while( j < rsize ) { // Copy rest of second half
    arr[k++] = right[j++];
}
}
```

## Performance of merge sort

- ▶ Recall that for insertion sort, sorting had a worst-case running time of  $O(n^2)$  – the algorithm contained a nested loop.
- ▶ How does merge sort compare?

## Performance of merge sort

- ▶ It turns out that merge sort has a worst-case running time of  $O(n \log n)$ .
- ▶ Why is this?
- ▶ The “splitting” part of merge sort will take  $\log n$  running time – as we have seen, it requires  $\log_2 n$  steps to repeatedly divide a number into two until you reach 1.
- ▶ And at each “level” – i.e., for each of those  $\log_2 n$  steps – we will have to do a merge
  - ▶ and we have seen that the run-time of the merge algorithm is linear – that is,  $O(n)$  – with respect to the size of the merged list.
- ▶ The way in which these combine is a little complicated, and we will not cover it – but it turns out that the merge sort algorithm as a whole ends up with  $O(n \log n)$  complexity.



## Performance of merge sort

- ▶ This turns out to be the *best possible* “big ‘O’ ” running time can have for sorting a list (in the worst case).
- ▶ This doesn't mean merge sort will *always* be faster than insertion sort.
- ▶ It could well be that some other sorting algorithms might perform better on lists up to a certain size.

## Costs of merge sort

- ▶ The good performance of merge sort does come with some disadvantages.
- ▶ For insertion sort, we could sort the array “in place”: we didn’t have to allocate any extra arrays besides the one we were sorting.
- ▶ But for merge sort, when we do the “merge” step, we *did* have to allocate extra arrays – we copied elements from the original array, into a “left” and a “right” temporary array.
- ▶ So although merge sort has a better run-time complexity than insertion sort, it will use more memory.
- ▶ Usually, however, the cost of the extra memory is not unreasonable.

## Use of merge sort

- ▶ Merge sort uses the fewest number of comparisons of the popular sorting algorithms.
- ▶ So, it is often a good choice for a general-purpose sorting algorithm.
- ▶ Merge sort was the sorting algorithm used for sorting arrays of objects in versions of Java [up until Java version 7](#)
  - ▶ From version 7 of Java onwards, the sorting algorithm used is a more complex one invented by [Tim Peters](#).
  - ▶ However, that algorithm still uses general ideas taken from merge sort.