Merge sort

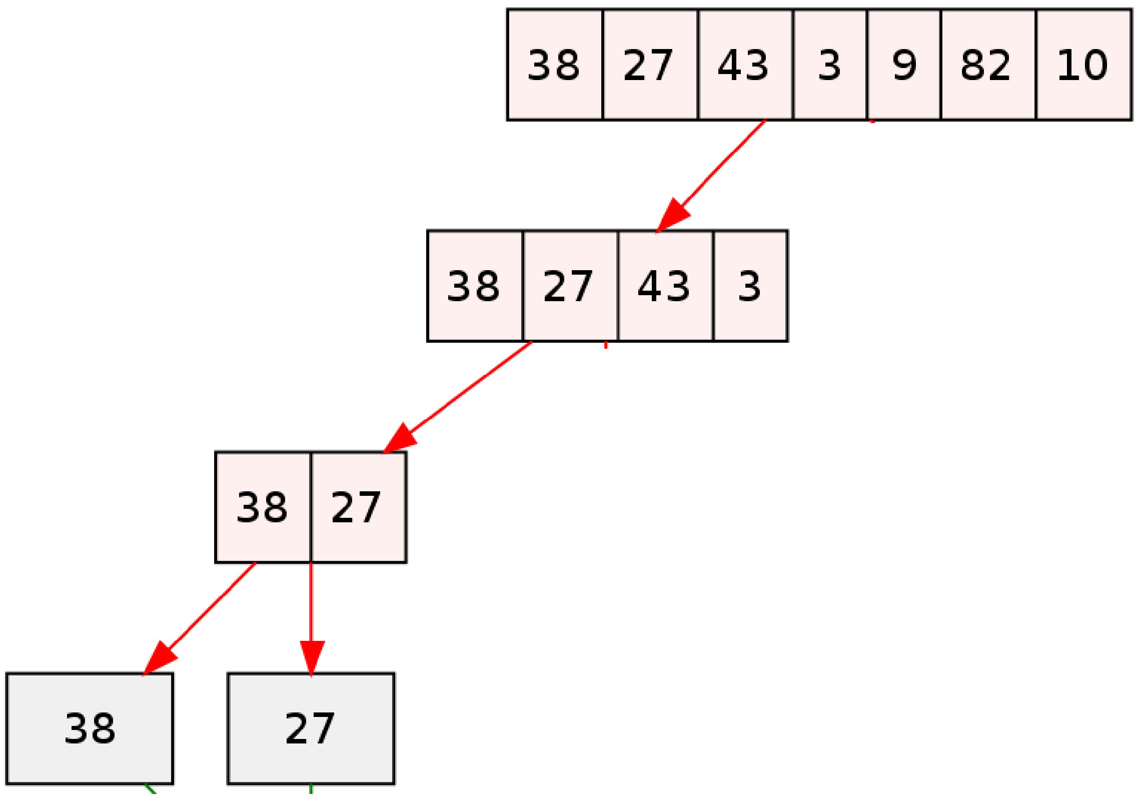
Adam Chebil

1 core

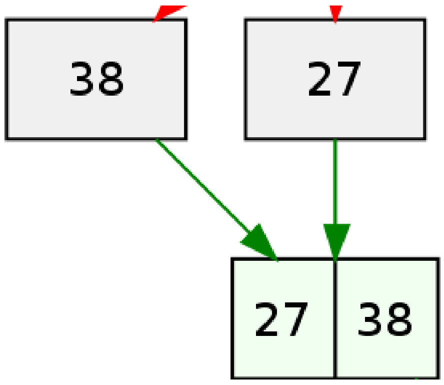
In principle, the algorithm should barely have any change in its working when using 1 or above 1 thread. The algorithm uses a pointer array and recursion to sort the array. It starts with an unsorted array. The array is then fully passed to the first function. Here the size is also given in order to know where the middle is. We are working with an array, so the size must be remembered and passed at all times separately, for it cannot be calculated afterwards [1]. After the first function, half the array is passed to the same function recursively. This is where the code in 1 thread continues, but to simply explain it, the next line does the same but for the second half.

So after passing the first half to its own function, the process starts again. This happens until the size of the array is 1. If it is one, the function returns to the previous recursion step, this is where the next half gets passed. Often this second half would then also be only have a size of 1. So it also returns here right away.

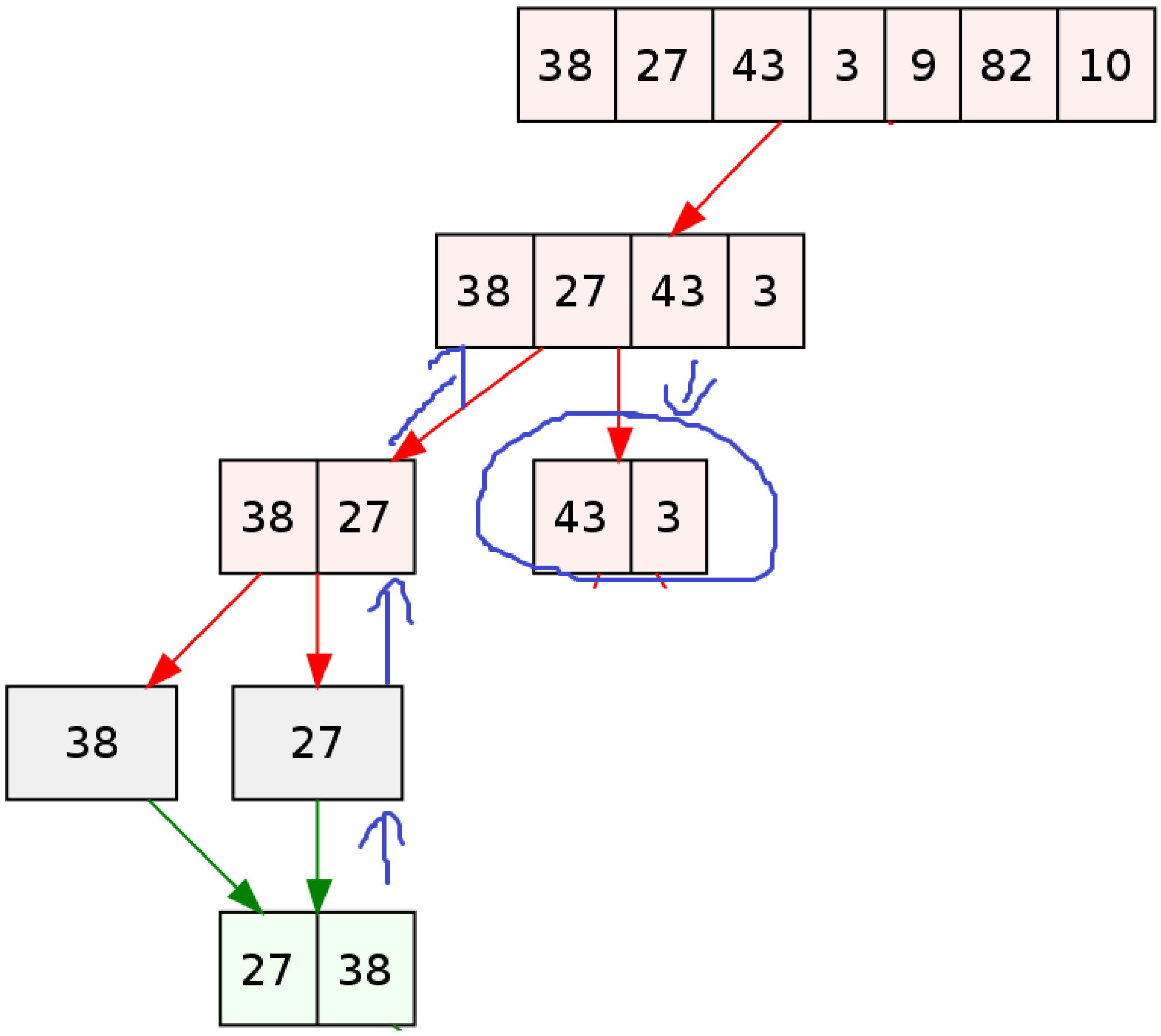
This has happened so far:



After this, merging will start with the now 2 sorted arrays in the merge function. To do this, we make an empty array with the same size as the unsorted array. Then we sort the location of each element one by one. We start with the first half of the array one by one. We look with a linear search what that elements location would be if placed in the other half, and add that to its current location. This is the location in the merged list, and is thus placed here in the empty array. This same algorithm happens for each element.



Due to the recursive nature of the script, we now go back to the other unsorted arrays before the last function call:



Nothing changes with the algorithm from here, until the array with [43, 3] also gets sorted. From there the script returns all the way to the array with 4 elements. With 2 elements, its splits, find out its an array with size of 1, return, and merge. The array with 4 had to wait until the recursive function returned. After sorting the 2 arrays of 2, it finally does return and thus will sort this array as well. This array now contains 2 sorted arrays, but is not sorted itself. But the execution is not changed, the array is passed to the merge function, seen as 2 arrays by determining its middle, and sorted the same way. No code changes.

This then returns to the array of 8, who has been waiting to recurse into its second half. This half goes through the same process and ends with a sorted half. After this the original array, now containing 2 sorted halves, gets sorted itself.

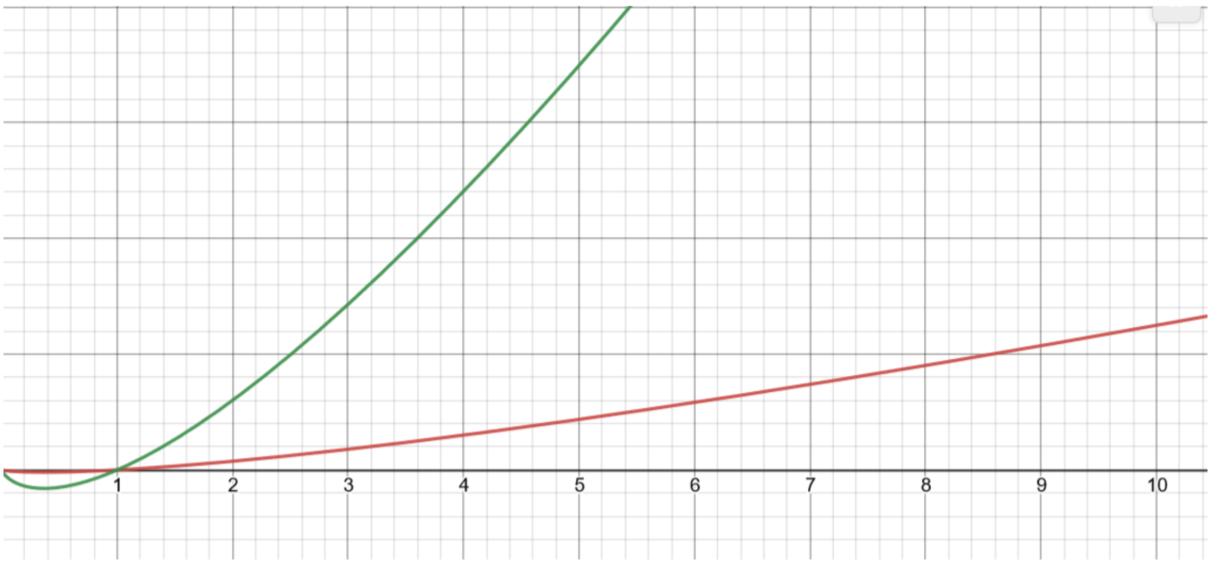
2 or more cores

With a parallel algorithm, only one small change has to happen. It splits at the call where a recursive function call happens, and asks to join the threads right after. The join happens before the sorting of the array. With 2 cores this would mean that each half of the 8 sized original array does the merge sort algorithm in parallel, but waits to join in order to do the last step of sorting the size 8 array, which would at that point contain 2 sorted array halves. With 4 cores this same principle would happen, but with one layer further down. The maximum possible threads is therefore the same amount as your cores. Some systems allow you to do more threads than owned cores by using the system scheduler. It then is not really going any faster because the system just switches between tasks (like how you can have 9 tabs open without having 9 cores in your cpu). It will have the effect of more threads, but not the speed. This algorithm allows for that with no problem because of the small to no difference in execution between single threaded and parallel.

Space and time complexity.

As seen in the description, at every merge there is a extra empty array made to sort the arrays in. The algorithm does make use of a pointer array, meaning that no extra array is made at any function call. This means that the space complexity is at most 2N.

Single threaded merge sort has a time complexity of **O(n*Log n).** The parallel system of this algorithm halves the array in all its functionality except for the last merge. This last step is a constant sized step and thus removable form the formula. The new time complexity then becomes **O(n*Log n)/P** with P being the amount of threads and bigger than 0.



n*Log n (green) and (n*Log n)/8 (red)