Merge Sort - Cprogramming.com

Merge sort is the second guaranteed O(nlog(n)) sort we'll look at. Like heap sort, merge sort requires additional memory proportional to the size of the input for scratch space, but, unlike heap sort, merge sort is stable, meaning that "equal" elements are ordered the same once sorting is complete.

Merge sort works using the principle that if you have two sorted lists, you can merge them together to form another sorted list. Consequently, sorting a large list can be thought of as a problem of sorting two smaller lists and then merging those two lists together. For instance, if you have the list

```
1 9 7 6
```

you could divide it into two lists,

```
1 9
and
7 6
```

Once those two lists are sorted:

```
1 9
and
6 7
```

They could be merged back together easily by starting at the left end of each list and then picking the smaller value. This process is illustrated below for those who find a visual approach helpful:

```
New list: 1
9
and
6 7
```

```
New list: 1 6
9
and
7
```

```
New list: 1 6 7
9
and
empty list
```

```
New list: 1 6 7 9
```

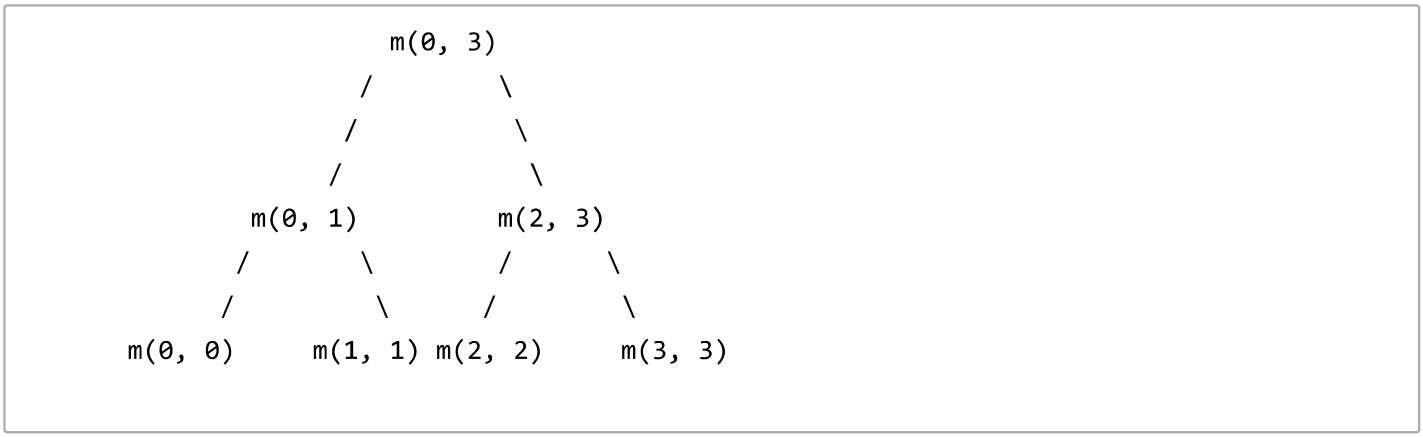
Here's the key to merge sort: once you've broken the problem in a problem of sorting and then merging two smaller lists, you can then apply merge sort to each of those smaller lists. This is a recursive process, so it will need a base case. Specifically, once we've reached a single element array, we know it's sorted (it has only one element, which must be in the right position) and we can just merge it with its neighbor to produce a new, sorted two-element array. This array, again, can be recombined with a neighbor, and so on until the entire array is sorted.

Merge sort is also our first divide-and-conquer sort. The term divide and conquer refers to breaking the problem into simpler sub-problems, each of which is then solved by applying the same approach, until the sub-problems are small enough to be solved immediately.

Merge sort guarantees O(nlog(n)) complexity because it always splits the work in half. In order to understand how we derive this time complexity for merge sort, consider the two factors involved: the number of recursive calls, and the time taken to merge each list together.

First, let's consider the number of recursive calls, as this will shed some light onto our understanding of the list merge operation. Each recursive call will either be a base case, or will result in two future recursive calls. The first call starts off by making two calls; each of those makes four, and so forth. What does this sound like?

If you thought, "a binary tree", then you're absolutely right. An easy way to visualize merge sort is as a tree of recursive calls. To save a bit of space, I will use m(lower, upper) to indicate merge sort called from element lower to element upper. For instance, m(0, n-1) would be the merge sort call for an array of size n in C/C++.



So we see that for an array of four elements, we have a tree of depth three. Now let's say we doubled the number of elements in the array to eight; each merge sort at the bottom of this tree would now have double the number of elements -- two rather than one. This means we'd need one additional recursive call at each element. This suggests that the total depth of the tree is log(n) + 1, the number of times we need to halve the number of elements in the array to reach the base case.

Now, what about the amount of work done at each recursive call? At first, you might think that every merge in the tree should equate to O(n) time, but this is incorrect. At each level, the number of elements is being dramatically reduced; at the bottom branch, it is certainly not taking O(n) time to perform a non-operation. At the level where

the results of the base case are being merged (at depth 1 in the above tree), each merge sort call is merging exactly half the list. At the root node is the only time the entire list is merged together at a single node.

As a result, it makes more sense to think about merge sort in terms of the number of operations performed on a single level of the tree. At each level, a total of n operations take place, and there are log(n) + 1 levels; consequently, the overall time complexity is O(n * log(n)).

Moreover, merge sort is stable -- so long as you break ties by picking from the correct list, equal elements will always end up in the same order as before. Specifically, if you split an array into a left half and a right half, you would break ties in favor of the left half, as it precedes the right half. This allows equal elements to stay ordered across merge operations.

The downside of merge sort is that it usually does require a scratch array to store the results of a merge. In place mergesort with arrays is a complex problem beyond the scope of this discussion. On the other hand, when dealing with linked lists, merge sort can be outstanding because no scratch space is needed. As an exercise, try implementing merge sort for linked lists without using any extra space save for a few extra variables.

Implementation Here is a find a sample implementation of merge sort. The code is a bit too long to post here, but you should check it out and notice one important feature: the scratch space is only allocated once. Malloc, free and other memory allocation routines (e.g., new and delete in C++) are typically fairly slow. As a consequence, reallocating the scratch space for every recursive call would be time prohibitive and would significantly increase the constant factor of merge sort.

Summary Merge sort is a fast, stable sorting routine with guaranteed O(n*log(n)) efficiency. When sorting arrays, merge sort requires additional scratch space proportional to the size of the input array. Merge sort is relatively simple to code and offers performance typically only slightly below that of quicksort.

Previous: Heap Sort
Next: Quicksort