CS125 : Introduction to Computer Science

Lecture Notes #40
Advanced Sorting Analysis

Lecture 40 : Advanced Sorting Analysis

Analyzing `MergeSort`

Mergesort can be described in this manner:

```
time to sort n values = time to sort left half +
                        time to sort right half +
                          time to merge sorted halves
```

which is commonly written in the form of a *recurrence relation*, i.e. a recursive mathematical formula:

```
T(n) is the time to run MergeSort on n values

T(n) = T(n/2) + T(n/2) + time to merge
T(1) = 1     / base case is constant time
```

or after doing some algebra:

```
T(n) = 2 * T(n/2)  + time to merge
T(1) = 1
```

You'll learn about recurrence relations in a different course, and learn how to solve them. For the moment, it simply makes for a convenient notation for the mergesort running time, before we get our final result.

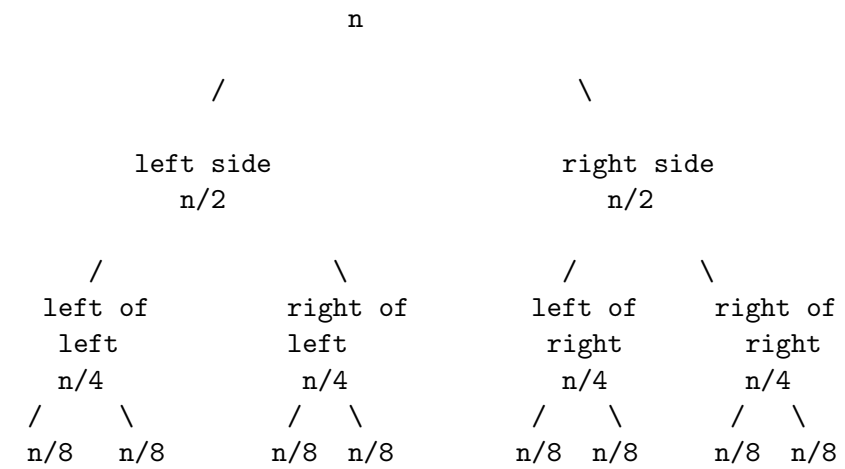
Now, what is the running time for merge? Well, if we have n total values among the two sorted collections we are merging, and we spend constant time on each step to write one of those values into the temporary array, then we have linear time total. And then, when we write the temporary array back into the original array, likewise, that takes constant time to write one value on each step, so that is likewise, linear time. So, overall, for a given step of MergeSort, the merge work is linear; each value is written to the temporary array once and then written back once.

So, the recurrence relation looks like this:

```
T(n) = 2 * T(n/2) + linear
T(1) = 1
```

That describes mergesort – two recursive calls on an array half the size, plus linear work to merge them. Now, you don't have the mathematical tools to solve that yet; eventually, you'll learn all about solving recurrence relations.

But in this case, we can still analyze Mergesort – we just will use a different technique than the recurrence relation. Each step breaks the array in half, and then we sort each of those halves before trying to merge the entire array. So if we start out with n elements, consider the following diagram, where the number we list (n, n/2, etc.) is the number of cells in the array at that level of recursion:



Notice that each level adds up to n. The first level is indicating the Merge of the entire array that we do at the end. On the second level, we have two recursive calls (the two we made from our first call to Mergesort). Each one eventually runs Merge on an array of size n/2, but we double that time because we have two such recursive calls. So that level, too, adds up to n.

The next level involves the third level of recursion – the calls we make, from the calls we made, from our first call. Since we had already recursively called on half the array on the level above, when we made two calls on half of THAT array, we end up with two recursive calls on an array one fourth the size of the original. So the Merge for such a call takes time n/4, and we have four such calls, so again, the total time for all the 3-levels-deep recursive calls is n.

And so on. Each row will total to n, in terms of time. And we have a logarithmic number of rows, since we keep cutting the array size in half as we move from one level, to the level below it. So if each row has time n, and there are a logarithmic number of rows, then the result is that the running time for Mergesort has an order of growth equal to:

```
linear times logarithmic
```

OR

```
n * log-base-2 of n
```

OR

```
n * lg n   // lg is a shorthand for log-base-2
```

but "linear times logarithmic" is good enough for our purposes here. It means the algorithm takes longer than linear time – which makes sense, since the last merge *alone* is linear time – but it's not quite as long as quadratic time. (just as constant is less than logarithmic is less than linear, likewise n*constant is less than n*lg n is less than n * n, i.e. linear is less than n * lg n is less than quadratic).

This turns out to be exactly the result we'd get if we mathematically solved the recurrence relation we listed earlier. Our diagram above was just an alternate way of arriving at the same result.

And note Mergesort does not take any advantage of the array being already sorted – the same work gets done regardless. So it's n *lg n for any case – worst, average, or even best.

Analyzing `Quicksort`

Partition is linear – each step takes constant time to place one value beyond the "left wall" or "right wall" so to partition an array of n values takes time that is linear in n.

So, in the best case, we get the best possible pivot for quicksort and the two halves of the array are equal in size. In that case, we have the same recurrence relation as for Mergesort:

```
T(n) = linear + 2 * T(n/2)
T(1) = 1
```

So quicksort in the best case is n*lg n. And on average, we expect quicksort to get a good pivot most of the time, so we can expect the average case to be similar to the best case, and thus to *also* be n*lg n. (That's not a proof, of course, but the proof that the average case is basically equal to the best case is beyond this course. I just want to give you a gut feeling that it's true, so that you remember it better.)

The worst case for quicksort is when the pivot is always as low as possible, and in that case, basically everything, or everything but one value, in the partition, ends up on the right side of the pivot. So each recursive call to sort the right side, is basically only sorting only one or two fewer cells than the previous call...so this is basically the same as SelectionSort at that point, and is thus quadratic.

Summary:

- Quicksort
  - best case: n * lg n
  - average case: n * lg n
  - worst case: quadratic
  - advantage: on average, fastest known comparison-based sorting algorithm; i.e. the only faster algorithms we know of are designed for specific kinds of data, rather than working on anything we can compare. Both quicksort and mergesort have the same order of growth, but in terms of constant factors of that n * lg n term, quicksort's constants are lower.
  - disadvantage: the quadratic worst case. It's very unlikely to occur but it *can* happen, making quicksort not the best choice if you *need* to be at n * lg n time.
- Mergesort
  - best case: n * lg n
  - average case : n * lg n
  - worst case: n * lg n
  - advangtage: n * lg n in all cases, even worst case
  - disadvantage: uses linear extra memory (the temporary array needed by merge)...this is especially a problem if you have a VERY large array to sort, since you might not even *have* enough memory left over to create another array that size. Also, not quite as fast as quicksort's average case, when you consider the constant factors (since they both have n * lg n order of growth)
- InsertionSort
  - best case: linear
  - average case: quadratic, though about half the time of worst case
  - worst case: quadratic
  - advantage: works well for partially sorted or completely sorted arrays; also good for small arrays since quicksort and mergesort tend to be overkill for such arrays
  - disadvantage: quadratic for any randomly arranged array, i.e. it's not better than quadratic unless the array *is* sorted a lot already
- SelectionSort
  - best case: quadratic
  - average case: quadratic
  - worst case: quadratic
  - advantage: fewest number of swaps...in situations (sorting integers is not one of them) where swapping our data can take a very long time due to very large amounts of it to move back and forth, swapping might have such a LARGE constant on it that it would overshadow everything else. In that case, we might want selection sort.

- disadvantage: quadratic in all cases, can't even be improved upon if the array is partially sorted. Works okay for small arrays but insertionsort works better for those arrays...so in general, SelectionSort is not generally useful. It's only useful in the one case listed as an advantage above.