CS104: Data Structures and Object-Oriented Design (Fall 2013)
November 7, 2013: Recursive Sorting Algorithms and their Analysis
Scribes: CS 104 Teaching Team

Lecture Summary

In this lecture, we continued our analysis of Merge Sort, by focusing on its running time and deriving the $O(n \log n)$ bound. Then, we turned our attention to Quick Sort. We saw that with a good choice of pivot, it runs in $O(n \log n)$ as well, but with a bad choice, it could take $\Theta(n^2)$.

1 Analysis of Merge Sort

If we look at the Merge Sort algorithm (from last class) more closely, we notice that the running time on an array of size n accrues as follows:

1. First, we spend time $O(1)$ for computing m.
2. Then, we make two recursive calls to Merge Sort, with arrays of sizes $\lfloor (n-1)/2 \rfloor$ and $\lceil (n-1)/2 \rceil$.
3. Finally, we call `Merge`. `Merge` goes through the two subarrays with one loop, always increasing one of i and j. Thus, it takes time $\Theta(n)$.

Let's use $T(n)$ to denote the worst-case running time of Merge Sort on an array of n. We don't know yet what $T(n)$ is — after all, that's what we're trying to work out right now. But we do know that it must satisfy the following recurrence relation:

$$\begin{aligned} T(n) &= T(\lfloor (n-1)/2 \rfloor) + T(\lceil (n-1)/2 \rceil) + \Theta(n), \\ T(1) &= \Theta(1). \\ T(0) &= \Theta(1). \end{aligned}$$

The last two cases are just the base case where the recursion bottoms out. The first equation is obtained as follows: We worked out that the running time for an array of size n is the time for `Merge` (and the computation of m), which is $\Theta(n)$, plus the time for the two recursive calls. We don't know how long those two take yet, but we know that they must be the function T applied to the respective array sizes.

Having worked out the recurrence, with a bit of experience, we actually realize that in this type of recurrence, it doesn't really matter whether one rounds up or down, and whether it is $(n-1)/2$ or $n/2$. If you are worried about this, you can work out the details and verify them. There are certainly cases where this would matter, but until you start analyzing pretty intricate algorithms and data structures (at which point you'll have learned the material so well that you don't need these lecture notes), you can always disregard, for the purpose of analysis, such small details. So we get a simpler recurrence:

$$\begin{aligned} T(n) &= 2T(n/2) + \Theta(n), \\ T(1) &= \Theta(1). \end{aligned}$$

This type of recurrence pops up very frequently in the analysis of Divide&Conquer algorithms. The more general form is $T(n) = a \cdot T(n/b) + f(n)$, for some constant numbers a and b, and some function f. You will learn in CSCI270 how to solve this more general case, including by a theorem called the Master Theorem (which really isn't very hard, the name notwithstanding). But it's actually not too hard to do by hand. The typical way to solve such recurrences by hand is to draw a recursion tree, work out the amount of work on each level and the number of levels, and then add them up. This gives you a conjecture for the formula for $T(n)$. This conjecture can then be verified using (strong) induction on n. After a bit of experience, one may omit the induction proof (since it's always very similar), but for now, verifying by induction is quite important.

The recursion tree is given in Figure 1:

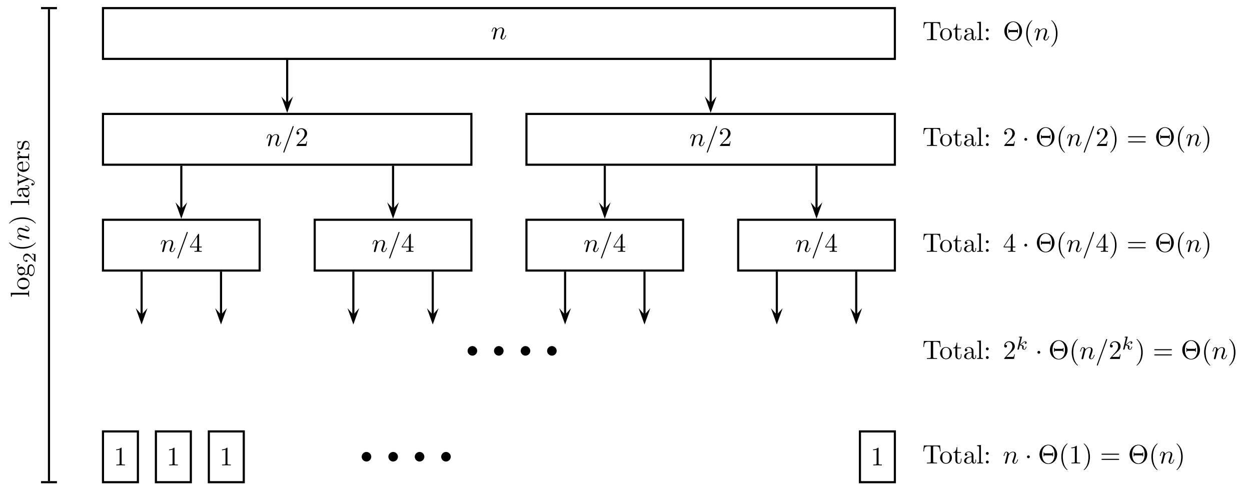
$\log_2(n)$ layers
n
$n/2$
$n/2$
$n/4$
$n/4$
$n/4$
$n/4$
1
1
1
1
Total: $\Theta(n)$
Total: $2 \cdot \Theta(n/2) = \Theta(n)$
Total: $4 \cdot \Theta(n/4) = \Theta(n)$
Total: $2^k \cdot \Theta(n/2^k) = \Theta(n)$
Total: $n \cdot \Theta(1) = \Theta(n)$

Figure 1: A recursion tree for Merge Sort. The numbers in the boxes are the array sizes, and we know that the work for `Merge` for an array of size s is $\Theta(s)$. In layer k, there are 2^k subarrays to process in the separate recursive calls, and each of them has size $n/2^k$.

Each level corresponds to one level of recursion. The top (root, level 0) is the call on an array of size n. This causes two calls with arrays of size $n/2$, at the next lower level (level 1). Each of those causes two calls with arrays of size $n/4$, for a total of 4 such calls at level 2. More generally, at level k, we have 2^k recursive calls, each on an array of size $n/2^k$. This bottoms out at the level where the array sizes are 1.

Now, if we look at the total work done by the calls to `Merge` at level k, we see that we call the function 2^k times, on arrays of size $n/2^k$ each. Since the running time for `Merge` is linear in the array size, we get $2^k\Theta(n/2^k) = \Theta(n)$. Thus, the total work at each level is $\Theta(n)$.

Next, we calculate the number of levels. Since the array size halves with each extra level, and it starts with n and finishes with 1, the number of levels is $\log_2(n)$. (Technically, it is $\lceil \log_2(n) \rceil$, but such details don't affect the running time analysis.) So the total work, summing up over all levels, is $\Theta(n \log_2(n))$. Remember that all logarithms are the same up to constant factors (base conversion), so we can just write $\Theta(n \log n)$, and leave the base unspecified.

So now, we have a pretty well-founded conjecture that $T(n) = \Theta(n \log n)$. But we want to prove this conjecture very formally. Whenever we want to formally analyze the running time of a recursive algorithm, the proof technique of choice is induction, because to prove something for bigger arrays, we'll want to rely on the same result for smaller arrays. This requires us to write a formal induction hypothesis and statement. First, to avoid mistakes we might make with $\Theta()$ notation (there are some subtleties when combining recursion and $\Theta()$), let's pretend that the recurrence is actually:

$$\begin{aligned} T(n) &= 2T(n/2) + n, \\ T(1) &= 1, \end{aligned}$$

getting rid of all $\Theta()$ notation. Then, we also want to make the precise form of the conjecture concrete, by saying that

$$T(n) \quad = \quad n(1 + \log_2(n)).$$

You may wonder where the "1+" came from suddenly. When looking at the base case $n = 1$, we notice that $T(1) = 1$ is what we want, but if we just had $n \log(n)$, because $\log(1) = 0$ (regardless of base), we would have 0. Adding "1+" is a bit of a trial-and-error result.

We will now prove the conjecture by (strong) induction on n. For the base case $n = 1$, we get that $T(1) = 1$ by the recurrence, and our formula gives us $1 \cdot (1 + \log_2(1)) = 1 \cdot (1 + 0) = 1$, so the base case worked.

Next, we state the induction hypothesis. Because we are doing *strong* induction, we get to assume that *for all* $k < n$, the statement $T(k) = k(1 + \log_2(k))$ is true. We want to prove that $T(n) = n(1 + \log_2(n))$. Notice here the difference between normal induction and strong induction. In normal induction, we would only get to assume that the hypothesis holds for $k = n - 1$, whereas using strong induction, we can assume it for *all* $k < n$, including $k = n - 1$. This is useful when, as here, the recurrence for n is based on values for parameters other than $n - 1$; in our case, this is $n/2$.

To actually prove this, we begin by using the only thing we know for sure about $T(n)$, namely that

$$T(n) \quad = \quad 2T(n/2) + n.$$

Now, we need to deal with those $T(n/2)$ terms. Here, we apply the induction hypothesis with $k = n/2$. We are allowed to do so, because $n \geq 2$ implies that $n/2 < n$. Then, we get that

$$T(n) \;=\; 2T(n/2) + n \;\overset{I.H.}{=}\; 2\frac{n}{2}(1 + \log_2(n/2)) + n \;=\; n(1 + \log_2(n) - 1) + n \;=\; n(1 + \log_2(n)).$$

This is exactly what we wanted to show, so the induction step is finished, and we have completed the induction proof.

2 Quick Sort

Merge Sort was our first illustration of a Divide&Conquer sorting algorithm. The other classic representative is Quick Sort. Merge Sort does practically no work in the Divide step — it only calculates m, the middle position of the array. Instead, the Combine step is where all the work happens. Quick Sort is at the other extreme: it does all its hard work in the Divide step, then calls itself recursively, and literally does nothing for the Combine step.

The Divide step in Quick Sort makes sure to move all of the smaller numbers to the left side of the array (but not sort them there), all of the larger numbers to the right side of the array (again, not sorted further), and then call itself recursively on the two sides. This division is accomplished with a *pivot element* `p`. The left side will contain only elements that are *at most* as large as `p`, while the right side contains only elements *at least* as large as `p`. (Depending on the implementation, elements equal to `p` may end up on either side, or the implementation may choose to put them all in one side.) Once we do this Divide step, we know that no elements of the left will ever need to end up on the right side, or vice versa. Thus, after we recursively sort the two sides, the entire array will be sorted.

Note that the left and right sides of the array will not necessarily be the same size, not even approximately. In Merge Sort, we carefully ensured to divide the array into two equal halves. In Quick Sort, we may also end up with two halves, but that would require choosing the median of the array as the pivot. More formally, the Quick Sort algorithm looks as follows. (This is slightly revised from the version in class.)

```
void QuickSort (T a[], int l, int r) {
    if (l < r) {
        int m = partition(a, l, r);
        QuickSort(a, l, m-1);
        QuickSort(a, m+1, r);
    }
}
```

As you can see, basically all the work must be done in the `partition` function, which is as follows:

```
int partition (T a[], int l, int r) {
    int i = l; // i will mark the position such that everything strictly to
               // the left of i is less than or equal the pivot.
    T p = a[r];
    for (int j = l; j < r; j++) {
        if (a[j] <= p) {  // found an element that goes to the left
            a.swap(i, j); // put it on the left
            i++;
        }
    }
    a.swap (i, r); // put the pivot in its place
    return i;
}
```

The `partition` function runs one loop through the array, and moves to the left side any element that is smaller than the pivot, keeping track of the boundary (`i`) between elements that are smaller than the pivot, and elements that may be (or are) larger. This is a perfectly adequate implementation of the `partition` function, but it uses about twice as many swaps as necessary. (In big-O notation, that does not matter, of course, but in practice, such a factor of 2 can be very important.) The slightly more clever implementation is to have two counters `i, j`, one starting at the left, the other at the right. Whenever `a[i] > p` and `a[j] < p`, the two elements at `i` and `j` are swapped. Otherwise, `i` is incremented, `j` is decremented, or both. When the counters cross each other, the function is done. One has to be a little careful with the indices `i` such that `a[i] = p`, which is why in class, we presented the simpler and slower version. But it's not too hard to work out.

Again, because Quick Sort is a recursive algorithm, the correctness statement is simply the overall correctness condition, and we don't need a loop invariant:

> After `QuickSort(a,l,r)` executes, `a` is sorted between `l` and `r`, and contains the same elements as the original array.

To prove this condition, we would again use induction over the array size $n = r + 1 - l$, just as with Merge Sort. Also, as with Merge Sort, since all the actual work is done by a "helper function" — here, the `partition` function — the correctness proof would mostly rely on a lemma about the behavior of that helper function.

Lemma 1 *When `partition(a,l,r)` returns the value `m`, the array `a` still contains the same elements, and has the following property: (1) Each position $l \leq i < m$ satisfies $a[i] \leq a[m]$, and (2) Each position $m < i \leq r$ satisfies $a[i] \geq a[m]$.*

We can visualize this lemma as in Figure 2, covering the main insight of the Quick Sort algorithm visually (like we did with the other sorting algorithms):

To prove the lemma, we would again need induction. Since `partition` consists mostly of a loop, we would need a loop invariant, capturing formally the outline of how `partition` works. We won't do that here, but it isn't too hard. Once we have the lemma, the correctness proof for Quick Sort is very straightforward. It simply uses strong induction on the array size, and applies the lemma and the induction hypothesis. This is one you should be able to work out pretty easily on your own.

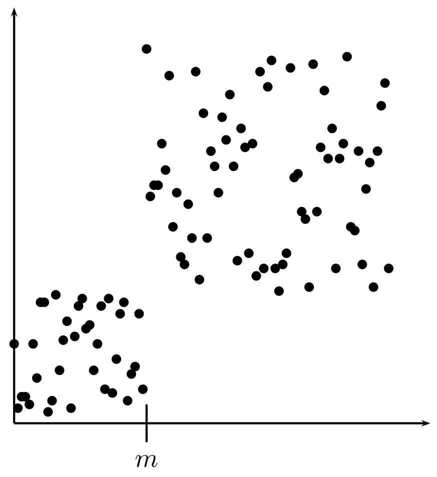

Figure 2: A pictorial representation of the state of Quick Sort after calling `partition`.

2.1 Running Time

To analyze the running time of Quick Sort, we use the same approach as we did for Merge Sort (and is common for many recursive algorithms, unless they are completely obvious). We let $T(n)$ represent the worst-case running time of the Quick Sort algorithm on an array of size n. To get a hold of $T(n)$, we look at the algorithm line by line. The call to `partition` takes time $\Theta(n)$, because it runs one linear scan through the array, plus some constant time. Then, we have two recursive calls to Quick Sort. We let $k = m - 1 - l$ denote the size of the left subarray. Then, the first recursive call takes time $T(k)$, because it is a call on an array of size k. The second recursive call will take time $T(n - 1 - k)$, because the size of the right subarray is $n - 1 - k$. Therefore, the total running time of Quick Sort satisfies the recurrence

$$\begin{aligned} T(n) &= \Theta(n) + T(k) + T(n-1-k), \\ T(1) &= \Theta(1). \end{aligned}$$

This is quite a bit messier-looking than the recurrence for Merge Sort, and if we know nothing about k, we cannot really solve this recurrence. But in order to get a feel for it, we can play with it a bit, and explore different possible values of k.

1. For $k = \frac{n}{2}$, the recurrence becomes much simpler: $T(n) = \Theta(n) + T(n/2) + T(n/2 - 1)$, which — as we discussed in the context of Merge Sort — we can simplify to $T(n) = \Theta(n) + 2T(n/2)$. That's exactly the recurrence we have already solved for Merge Sort, and thus the running time of Quick Sort would be $\Theta(n \log n)$.
2. At the other extreme is $k = 0$ (or, similarly, $k = n - 1$). Then, we get only that $T(n) = \Theta(n) + T(0) + T(n-1)$, and since $T(0) = \Theta(1)$, this recurrence becomes $T(n) = \Theta(n) + T(n-1)$. This recurrence unrolls as $T(n) = \Theta(n) + \Theta(n-1) + \Theta(n-2) + \ldots + \Theta(1)$, so $T(n) = \Theta(\sum_{i=1}^{n} i) = \Theta(n^2)$.

The running time for $k = 0$ or $k = n - 1$ is thus just as bad as for the simple algorithms, and in fact, for $k = 0$, Quick Sort is essentially the same as Selection Sort. Of course, this quadratic running time would not be a problem if only the cases $k = 0$ and $k = n - 1$ did not appear in practice. But in fact, they do: with the pivot choice we implemented, these cases will happen whenever the array is already sorted (increasingly or decreasingly), which should actually be an easy case. They will also happen if the array is nearly sorted. This is quite likely in practice, for instance, because the array may have been sorted, and then just messed up a little with some new insertions.

As we can see, the choice of pivot is extremely important for the performance of Quick Sort. Our favorite pivot would be the median, since it ensures that $k = \frac{n}{2}$, giving us the best solution $T(n) = \Theta(n \log_2 n)$. There are in fact linear algorithms for finding the median; they are not particularly difficult (in particular, the randomized one is quite easy; the deterministic algorithm is a bit more clever), but belong more in an algorithms class.

Of course, we don't have to find exactly the median. As long as there exists some *constant* $\alpha > 0$ with $\alpha n \leq k \leq (1-\alpha)n$ for all recursive calls, we will get that $T(n) = \Theta(n \log n)$. However, the constant hidden inside the Θ gets worse as $\alpha \to 0$, because the array sizes in Quick Sort become more and more imbalanced.

So we would be quite happy if somehow, we could always pick a pivot p such that at least 25% of the array's entries are less than p and at least 25% are more than p. Notice that half of all entries in the array would actually satisfy this, namely, the entries which (in the sorted version) sit in array positions $n/4 \ldots 3n/4$. Unfortunately, right now, we don't really know how to find such a pivot, either. (It can be done with basically the same idea as finding a median.)

But one thing we can do is pick a *random* element of the array as a pivot. That will not *always* guarantee that the pivot satisfies this nice 25% property, but it happens on average half the time. So, half the time, the subarray sizes shrink at least by 25%, which is enough to ensure that we only have $O(\log n)$ levels of recursive calls, and thus running time $O(n \log n)$. The analysis of *Randomized Quick Sort* is not particularly difficult, but would take us about a lecture, so we'll skip it here — you will likely see it in CSCI270. But one can show that the expected running time $E[T(n)] = \Theta(n \log n)$. Notice that when the algorithm makes random choices, like Randomized Quick Sort does, the running time becomes a random variable. One can show that not only in expectation is it $\Theta(n \log n)$, but in fact, this happens most of the time.

In practice, Randomized Quick Sort performs quite well. Another way to achieve the same result is to scramble the array truly randomly first, and then run the default Quick Sort algorithm from above on the scrambled array. That is mathematically equivalent.

As an aside, here is how you randomly scramble an array:

```
for (int i = n-1; i > 0; i --)
     a.swap (i, rand()%(i+1));
```

This ensures that each permutation of the array is equally likely in the end. Don't try to scramble it by repeatedly swapping random pairs of elements, like the following:

```
for (int i = 0; i < MAXSWAPS; i ++)
    a.swap (rand()%n, rand()%n);
```

This does eventually give you a truly random array, but it takes way more swaps than you would normally run. This is actually an interesting area of mathematics (Analysis of Mixing Times of Markov Chains and Random Walks, specifically for card shuffling), and the main result is that you need MAXSWAPS $= \Omega(n \log n)$ pairwise swaps before the array actually looks random.