Recurrences

We have seen many algorithms thus far. For each one we have tried to express how many basic operations are required as a function of some parameter n which is typically the size of the input e.g. the number of elements in a list.

For algorithms that involve `for` loops, we can often write the number of operations of the loop component as a power of n, which corresponds to the number of nested loops. For example, if we have two nested `for` loops, each of which run n times, then these loops take time proportional to n^2. The sorting algorithms from lecture 6 and the grade school multiplication algorithm are a few examples.

For recursive algorithms, it is less obvious how to express the number of operations as a function of the size of the input. We have given some examples of recursive algorithms in the last few lectures. For these examples and others, we would like to express in a more general way how the time it takes to solve the problem depends on n. That is what we will do next and next lectuer. In each case, we express a function $t(n)$ in terms of $t(...)$ where the argument depends on n but it is a value smaller than n. Such a recursive definition of $t(n)$ is called a *recurrence relation.*

Example 1: reversing a list

Let $t(n)$ be the time it takes to reverse a list with n elements. Recall how this is done. You remove the first element of the list. Then, you take the remaining $n-1$ element list and recursively reverse them. Then you add the element you removed to the end of the reversed list.

Each recursive call reduces the problem from size n to size $n-1$. This suggests a relationship:

$$t(n) = c + t(n-1)$$

where the constant c is the time it takes in total to remove the first element from a list plus the time it takes to add that same element to the end of a list. We are not saying what c is. All that matters is that it is constant: it doesn't depend on n. (By the way, I am making an assumption here that the first element can be removed in constant time. If we are using an array list, then this is not so.)

To obtain an expression for $t(n)$ that is not recursive, we repeatedly substitute on the right side, as follows:

$$\begin{aligned} t(n) &= c + t(n-1) \\ &= c + c + t(n-2) \\ &= c + c + c + t(n-3) \\ &= \dots \\ &= cn + t(0). \end{aligned}$$

This method is called *backwards substitution.* Note $t(0)$ is the base case of the recursion and done in constant time. It would probably make more sense to stop at $n = 1$ rather than $n = 0$, and we would end up with

$$t(n) = c\,(n-1) + t(1)$$

in that case.

[ASIDE: One often writes such a recurrence in a slightly simpler way ($c = 1$):

$$t(n) = 1 + t(n-1) \ .$$

The idea is that since the constant c has no "units" anyhow, its meaning is unspecified except for the fact that it is constant, so we just treat it as a unit (1) number of instructions.]

Example 2: sorting a list by finding the minimum element

Recall the recursive algorithm for sorting which found the smallest element in a list and removed it, then sorted the remaining $n-1$ elements, and finally added the removed element to the front of the list. We express the time taken, using the recurrence:

$$t(n) = c\ n + t(n-1) \ .$$

As I discussed in the lecture, we could write the recurrence more precisely as

$$t(n) = c_1 + c_2\ n + t(n-1)$$

since in each recursive call there is some constant c_1 amount of work, plus some amount of work $c_1 n$ that depends linearly on the size n of the list in that call. But let's keep it simple and consider just the first recurrence.

Solving by back substitution:

$$\begin{aligned}
t(n) &= c\ n + t(n-1) \\
&= c\ n + c \cdot (n-1) + t(n-2) \\
&= \ldots \\
&= c\ \{\ n + (n-1) + (n-2)\ + \cdots + (n-k)\} + t(n-k-1) \\
&= c\ \{\ n + (n-1) + (n-2)\ + \cdots + 2 + 1\} + t(0) \\
&= \frac{cn(n+1)}{2} + t(0)
\end{aligned}$$

Informally we will say that is $O(n^2)$ since the largest term here that depends on n is n^2.

Example 3: Tower of Hanoi

Recall the Tower of Hanoi problem. Let $t(n)$ be the number of disk moves. The recurrence relation is:

$$t(n) = c + 2\ t(n-1).$$

The c on the right side refers to the work that is done with the call to `tower`. There is the single disk move that is done in each call, and there is also some administrative work that is done for each

recursive call (making a stack frame, pushing it on the call stack, and then popping the call stack when the method exits). All this constant time work is bundled together as a single constant c. This work is added to a term $2\ t(n-1)$ which is the time needed for the *two* recursive calls on the problem of size $n-1$.

Proceeding by back substitution, we get

$$\begin{aligned}
t(n) &= c + 2\ t(n-1) \\
&= c + 2(c + 2\ t(n-2)) \\
&= c\ (1+2) + 4\ t(n-2) \\
&= c\ (1+2) + 4\ (c + 2\ t(n-3)) \\
&= c\ (1+2+4)\ +\ 8\ t(n-3) \\
&= \ ... \\
&= c\ (1+2+4+8+\cdots+2^{k-1}) + 2^k\ \ t(n-k) \\
&= c\ (1+2+4+8+\cdots+2^{n-1}) + 2^n\ t(0) \\
&= c\ (2^n - 1) + 2^n\ t(0)
\end{aligned}$$

where we have used the familiar geometric series (recall lecture 2)

$$\sum_{i=0}^{n-1} x^i = \frac{x^n - 1}{x - 1}$$

for the case that $x = 2$. Note that we would say that the algorithm takes time $O(2^n)$.

Example 4: binary search

Recall the recursive binary search algorithm. We assume we have an ordered list of elements, and we would like to find a particular element `e` in the list. The algorithm computes the mid index and compares the element `e` to the element at that mid index. The algorithm then recursively calls itself, searching for `e` either in the lower or upper half of the list. Since the recursive call is on a list that is only half the size, we can express the time using the recurrence:

$$t(n) = c + t(\frac{n}{2})\ .$$

We suppose that n is a power of 2. Then,

$$\begin{aligned}
t(n) &= c + t(\frac{n}{2}) \\
&= c + c + t(\frac{n}{4}) \\
&= c + c + \cdots + t(\frac{n}{2^k}) \\
&= c + c + \cdots + c + t(\frac{n}{n})\ \text{, where } 2^k = n\ \ \textit{i.e. } k = \log_2 n, \text{ and we have } \log_2\ c's \\
&= c \log_2 n + t(1)
\end{aligned}$$

So we say that binary search is $O(\log_2 n)$ since the largest term that depends on n is $\log_2 n$.

Here we examine the recurrences for mergesort and quicksort.

Mergesort

Recall the mergesort algorithm: we divide the list of things to be sorted into two approximately equal size sublists, sort each of them, and then merge the result. Merging two sorted lists of size $\frac{n}{2}$ takes time proportional to n, since the merging requires iterating through the elements of each list. If n is even, then there are $\frac{n}{2} + \frac{n}{2} = n$ elements in the two lists. If n is odd then one of the lists has size one greater than the other, but there are still n steps to the merge.

Let's assume that n is a power of 2. This keeps the math simpler since we don't have to deal with the case that the two sublists are not exactly the same length. In this case, the recurrence relation for mergesort is:

$$t(n) = cn + 2t(\frac{n}{2}).$$

[ASIDE: If we were to consider a general n, then the correct way to write the recurrence would be:

$$t(n) = t(\ \lfloor \frac{n}{2} \rfloor\ ) + t(\ \lceil \frac{n}{2} \rceil\ ) + cn$$

where the $\lfloor \frac{n}{2} \rfloor$ means $floor(n/2)$ and $\lceil \frac{n}{2} \rceil$ means $ceiling(n/2)$, or "round down" and "round up", respectively. That is, rather than treating $n/2$ as an integer division and ignoring the remainder (rounding down always), we would be treating it as $n/2.0$ and either rounding up or down. In the lecture, I gave the example of $n = 13$ so $\lfloor \frac{n}{2} \rfloor = 6$ and $\lceil \frac{n}{2} \rceil = 7$. In COMP 250, we don't concern ourselves with this level of detail since nothing particualrly interesting happens in the general case, and we would just be getting bogged down with notation. The interesting aspect of mergesort is most simply expressed by considering the case that n is a power of 2.]

Let's solve the mergesort recurrent using backwards substitution:

$$\begin{aligned}
t(n) &= c\ n + 2\ t(\frac{n}{2}) \\
&= c\ n + 2\ (\ c\frac{n}{2} + \ 2t(\frac{n}{4})) \\
&= c\ n + c\ n + 4\ t(\frac{n}{4}) \\
&= c\ n + c\ n + 4\ (c\frac{n}{4}\ +\ 2\ t(\ \frac{n}{8}\ )) \\
&= c\ n + \ c\ n + \ c\ n + 8\ t(\frac{n}{8}) \\
&= c\ n\ k\ +\ 2^k\ t(\frac{n}{2^k}) \\
&= c\ n \log_2 n + \ n\ t(1), \quad \text{when } n = 2^k
\end{aligned}$$

which is $O(n\ \log_2 n)$ since the dominant term is $n\ \log_2 n$.

What is the intuition for why mergesort is $O(n \log_2 n)$? Think of the merge phases. The list of size n is ultimately partitioned down into n lists of size 1. If n is a power of 2, then these n lists are merged into $\frac{n}{2}$ lists of size 2, which are merged into $\frac{n}{4}$ lists of size 4, etc. So there are $\log_2 n$ "levels" of merging, and each require $O(n)$ work.

Notice that the bottleneck of the algorithm is the merging part, which takes a total of cn operations at each "level" of the recursion. There are $\log n$ levels.

On the base case of mergesort

With mergesort, we have a base case of $n = 1$. What if we had stopped the recursion at a larger base case? For example, suppose that when the list size has been reduced to 4 or less, we switch to running bubble sort instead of mergesort. Since bubble sort is $O(n^2)$, one might ask whether this would cause the mergesort algorithm to increase from $O(n \log_2 n)$ to $O(n^2)$.

Let's solve the recurrence for mergesort by assuming $t(n) = 2t(\frac{n}{2}) + c_1 n$ when $n > 4$ but that some other $t(n)$ holds for $n \leq 4$. Assume n is a power of 2 (to simplify the argument). We want to stop the backsubstitution at $t(4)$ on the right side. So we let k be such that $\frac{n}{2^k} = 4$, that is, $2^k = \frac{n}{4}$.

$$\begin{aligned} t(n) &= c\ n + 2\ t(\frac{n}{2}) \\ &= \ldots \\ &= c\ n\ k\ +\ 2^k\ t(\frac{n}{2^k}), \text{ and letting } 2^k = \frac{n}{4} \text{ gives...} \\ &= c\ n(\log_2 n - 2) + \frac{n}{4}\ t(4) \\ &= c\ n \log_2 n\ -\ 2cn + \frac{n}{4}\ t(4) \end{aligned}$$

Note that this is still $O(n \log_2 n)$ since the dominant term is $n \log_2 n$. Also note that by switching to bubble sort when $n = 4$, we really should write the solution as:

$$t(n) = c\ n \log_2 n\ -\ 2cn + \frac{n}{4}\ t_{bubble}(4).$$

While this might seem like a toy problem, it makes an important point. Sometimes when we write recursive methods, we find that the base case can be tricky to encode. If there is a slower method available that can be used for small instances of the problem, and this slower method is easy to encode, then use it!

Quicksort

Let's now turn to the recurrence for quicksort. Recall the main idea of quicksort. We remove some element called the pivot, and then partition the remaining elements based on whether they are smaller than or greater than the pivot, recursively quicksort these two lists, and then concatentate the two, putting the pivot in between.

In the best case, the partition produces two roughly equal sized lists. This is the best case because then one only needs about $\log n$ levels of the recursion and approximately the same recurrence as mergesort can be written and solved.

What about the worst case performance? If the element chosen as the pivot happens to be smaller then all the elements in the list, or larger then all the elements in the list, then the two lists are of size 0 and $n-1$. If this poor splitting happens at every level of the recursion, then performance degenerates to that of the $O(n^2)$ sorting algorithms we saw earlier, namely the recurrence becomes

$$t(n) = cn + t(n-1)\ .$$

Solving this by backstitution (see last lecture) gives

$$t(n) = c\frac{n(n+1)}{2} + t(0)n$$

which is $O(n^2)$.

Why is quicksort called "quick" when its worst case is $O(n^2)$? In particular, it would seem that mergesort would be quicker since mergesort is $O(n \log n)$, regardless of best or worst case.

There are two basic reasons why quicksort is "quick". One reason is that the first case is easy to avoid in practice. For example, if one is a bit more clever about choosing the pivot, then one can make the worst case situation happen with very low probability One idea for choosing a good pivot is to examine three particular elements in the list: the first element, the middle element, and the last element (`list[0], list[mid], list[size-1]`. For the pivot, one sorts these three elements and takes the middle value (the median) as the pivot. The idea is that it is very unlikely for *all three* of these elements to be among the three smallest (or three largest). In particlar, if the list happens to be close to sorted (or sorted in the wrong direction) then the "median of three" will tend to be close the median of the entire list. *Note that the best split occurs if we take the pivot to be the median of the whole list.* In practice, such a simple idea works very well, and partitions have close to even size.

The second reason that quicksort is quick is that, if one uses an array list to represent the list, then it is possible to do the partition *in place*, that is, without using extra space. The "in place" property of quicksort is a big advantage, since it reduces the number of copies one needs to do. By contrast, the straightforward implementation of mergesort requires that we use a second array and merge the elements into it. This leads to lots of copying, which tends to be slow. There are clever ways to implement mergesort which make it run faster, but the details are way beyond the scope of the course. Besides, experimental results have shown that, in practice, quicksort tends to be faster than mergesort even if the 'in place' mergesort method is used. Quicksort truly is very quick (if done in place).

... one last example

The last example we considered is:

$$t(n) = t(\frac{n}{2}) + n$$

This is similar to binary search, but now we have to do n operations during each call. What do you predict? Is this $O(\log_2 n)$ or $O(n)$ or $O(n^2)$ or what? The solution is given on the next page. Note that there is tricky geometric series that needs to be solved, namely $\sum_{i=0}^{\log_2 n} 2^i$. I suggest you work through it and do it on your own. This geometric series came up with array lists when we wanted to add n items to an empty array list and we were concerned about re-sizing. (See Q6 of the arraylist exercises.) It also comes up with hash tables, when you want to add n entries and you have a maximum on the load factor to deal with, and so you need to rehash possibly many times.

$$
\begin{aligned}
t(n) &= n + t(\frac{n}{2}) \\
&= n + \frac{n}{2} + t(\frac{n}{4}) \\
&= n + \frac{n}{2} + \frac{n}{4} + t(\frac{n}{8}) \\
&= n + \frac{n}{2} + \frac{n}{4} + \cdots \frac{n}{2^{k-1}} + t(\frac{n}{2^k}) \\
&= n + \frac{n}{2} + \frac{n}{4} + \cdots + 4 + 2 + t(1), \quad \text{when } 2^k = n \\
&= n + \frac{n}{2} + \frac{n}{4} + \cdots + 4 + 2 + 1 - 1 + t(1) \\
&= n \sum_{i=0}^{\log_2 n} 2^i + t(1) - 1 \\
&= (2^{(\log_2 n)+1} - 1)/(2-1) + t(1) - 1 \\
&= 2n - 1 \quad + \; t(1) - 1
\end{aligned}
$$

