

Recurrences

We have seen many algorithms thus far, and for each one we have tried to express how many operations $t(n)$ are required as a function of some parameter n which is typically the “size” of the problem.

For algorithms that involve `for` loops, it is straightforward to write the number of operations of the loop component as a polynomial, whose degree is 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 .

For recursive algorithms, it is typically less obvious how to express $t(n)$. Today, we will look at a number of examples.

Example 1: Factorial

Let $t(n)$ be the time it takes to compute $n!$. You should have an intuition that $t(n)$ is $O(n)$. This is easy to see if you use an iterative algorithm to compute $n!$ since we have a `for` loop which we iterate n times. What about if we use a recursive algorithm?

Algorithm: factorial(n)

```

if  $n > 0$  then
    return  $n * \text{factorial}(n - 1)$ 
end if

```

The recursive algorithm for computing $n!$ involves a method call, and a multiplication. These operations can each be done in constant time. Moreover, each method call reduces the problem from size n to size $n - 1$. This suggests a relationship:

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

namely the time it takes to compute $n!$ is some constant plus the time it takes to compute $(n - 1)!$. Such a relationship, in which $t(n)$ is expressed in terms of $t(*)$ where the argument is value smaller than n is called a *recurrence relation*.

Repeatedly substituting on the right side yields:

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

This method is called *backwards substitution* because we substitute starting at $t(n)$ and work our way back to $t(0)$. Note $t(0)$ is the base case of the recursion and done in constant time c_0 . So, $t(n) = cn + c_0$.

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: Tower of Hanoi

Here we analyze the asymptotic running time of the Tower of Hanoi algorithm (recall lecture 10). Let $t(n)$ be the number of disk “moves”. The recurrence relation is:

$$t(n) = 1 + 2 t(n - 1)$$

and $t(0) = 0$ since there is nothing to do when there are no disks to move. The “1” on the right side refers to the single disk move in each call. The $2 t(n - 1)$ is the time needed for the *two* recursive calls *within* the algorithm.

Proceeding by back substitution, we get

$$\begin{aligned} t(n) &= 1 + 2t(n - 1) \\ &= 1 + 2(1 + 2t(n - 2)) \\ &= 1 + 2 + 4t(n - 2) \\ &= 1 + 2 + 4(1 + 2t(n - 3)) \\ &= 1 + 2 + 4 + 8t(n - 3) \\ &= 1 + 2 + 4 + 8 + \dots + 2^{n-1} + 2^n t(0) \\ &= 2^n - 1 + 2^n t(0), \quad (*) \\ &= 2^n - 1, \quad \text{since } t(0) = 0 \end{aligned}$$

where (*) used the geometric series

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

for the case that $a = 2$.

[ASIDE: Alternatively, you can prove by induction that the solution to the recurrence equation is

$$t(n) = 2^n - 1.$$

By inspection, it is true for $n = 1$ since $t(1) = 1$, i.e. the towers of Hanoi problem for 1 disk just requires one move. Now assume it is true for $n = k$ and prove for $n = k + 1$.

$$\begin{aligned} t(k + 1) &= 1 + 2t(k) \\ &= 1 + 2 \cdot (2^k - 1) \text{ (by the induction hypothesis),} \\ &= 2^{k+1} - 1 \end{aligned}$$

which proves the induction step, so we are done.]

You may be wondering whether or not you need the factor “2” in the recurrence equation. For example, on the previous page I used $c = 1$ instead of the more general c and suggested it didn’t matter. What happens if we drop the factor 2 in the recurrent of Towers of Hanoi, and instead consider

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

However, notice that this just the recurrence we saw with $n!$, which was $O(n)$ rather than $O(2^n)$. Quite a big difference! So clearly the factor 2 matters.

Example 3

To see what's going on here, compare

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

versus

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

The first recurrence says that it takes c extra steps to reduce the size of the problem by 1. This implies that the total number of steps behaves as cn which is $O(n)$. The second recurrence says that it takes c *times* as many steps as when we reduce the problem size by 1. The number of steps now behaves as c^n which is $O(c^n)$ which is much bigger than $O(n)$.

Example 4

Let's consider another recurrence:

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

For example, consider a sorting algorithm that finds the minimum element in a list of size n , removes that element, and then (recursively) sorts the remaining list of size $n - 1$.

How do you solve this recurrence? By backwards substitution, we get

$$\begin{aligned} t(n) &= n + t(n - 1) \\ &= n + n - 1 + t(n - 2) \\ &= \dots \\ &= n + n - 1 + n - 2 + \dots + 2 + 1, \text{ where we assume } t(1) = 1 \\ &= \frac{n(n + 1)}{2} \end{aligned}$$

which is $O(n^2)$ and $\Omega(n^2)$.

Example 5

Here is a similar one:

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

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

which again is $O(n^2)$ and $\Omega(n^2)$.

Example 6: binary search

Recall the binary search algorithm. We assume we have an ordered list of elements, and we would like to find a particular element in the list. The algorithm computes the mid index and compares the element to the element at that index. It then recursively calls either the lower or upper half of the list. So,

$$t(n) = c + t(n/2) .$$

To keep the analysis simple, let's suppose that n is a power of 2, namely $n = 2^k$, and proceed by back substitution.

$$\begin{aligned} t(n) &= c + t(n/2) \\ &= c + c + t(n/4) \\ &= c + c + \dots + t(n/n) \\ &= c \log n + t(1) \end{aligned}$$

Example 7: power

Recall the recursive algorithm for computing x^n . At each level of the recursion, one either does two multiplications or one multiplication depending on whether the exponent argument at that level is odd or even respectively. This means that you cannot write an occurrence as an equality and instead one needs to use an inequality, e.g.

$$t(n) \leq c_2 + t\left(\frac{n}{2}\right)$$

or

$$t(n) \geq c_1 + t\left(\frac{n}{2}\right).$$

Solving these two inequalities gives an upper or lower bound respectively and one can easily show $t(n)$ is $O(\log n)$ and $\Omega(\log n)$.

Notation: floor and ceiling (rounding)

At the end of this lecture, I took some time to mention notation that will be using, when we round numbers down (floor) or up (ceiling):

$\lfloor x \rfloor$ is the largest integer that is less than or equal to x . $\lfloor \cdot \rfloor$ is called the *floor* operator.

$\lceil x \rceil$ is the smallest integer that is greater than or equal to x . $\lceil \cdot \rceil$ is called the *ceiling* operator.

For example, for any positive integer n , there is a unique integer l such that

$$2^l \leq n < 2^{l+1}$$

or

$$l \leq \log n < l + 1,$$

and $l = \lfloor \log n \rfloor$. We will see this inequality later when study binary trees.