Recurrence Equations as a Tool for Asymptotic Analysis*

As illustrated through examples discussed in class, it can be tedious to calculate the precise number of operations performed by a particular algorithm implementation, as a function of the size of the input. Here we introduce some techniques for simplifying and approximating these calculations. (Since we are generally only interested in the asymptotic order of growth of the running time, approximations are often justified.)

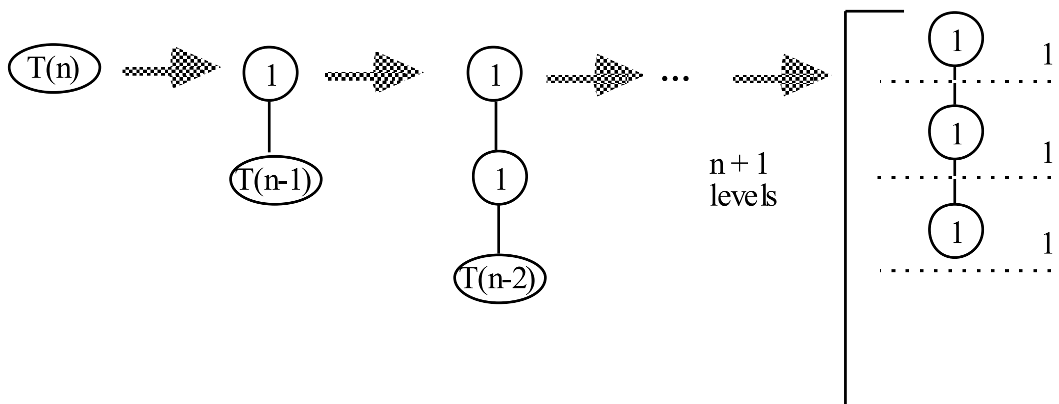
Linear Functions

Consider the first implementation of a method `buildVector()` for building a vector of integers, where each new integer is inserted at the back of the vector. Suppose that the function T(n) measures the number of references to locations in the vector that are required, as a function of the number n in `buildVector(n)`. Then we can define T(n) recursively as follows:

$$T(n) = 1 + T(n - 1),\ n > 0$$
$$T(0) = 0$$

Such recursive equations for defining a numerical function are called ***recurrence equations***. The solution to a set of recurrence equations is a function that satisfies the equations. Typically, we prefer to see such a function written in a non-recursive form. For example, the solution to the above recurrence equations is $T(n) = n$.

We can often view recurrence equations as expanding to a tree of nodes labeled by cost. For example, expanding the above equations gives:



The goal is to find a simple formula for summing up the costs in all the nodes of the fully expanded tree. In this case, there are n nodes, each of which costs 1 unit, so the total cost is $T(n) = n$. Note that $T(n) \in O(n)$. It turns out that $T(n) \in O(n)$ even if we generalize this particular set of recurrence equations:

$$T(n) = a + T(n - b),\ n > 0$$
$$T(n) = c,\ n \leq 0$$

---

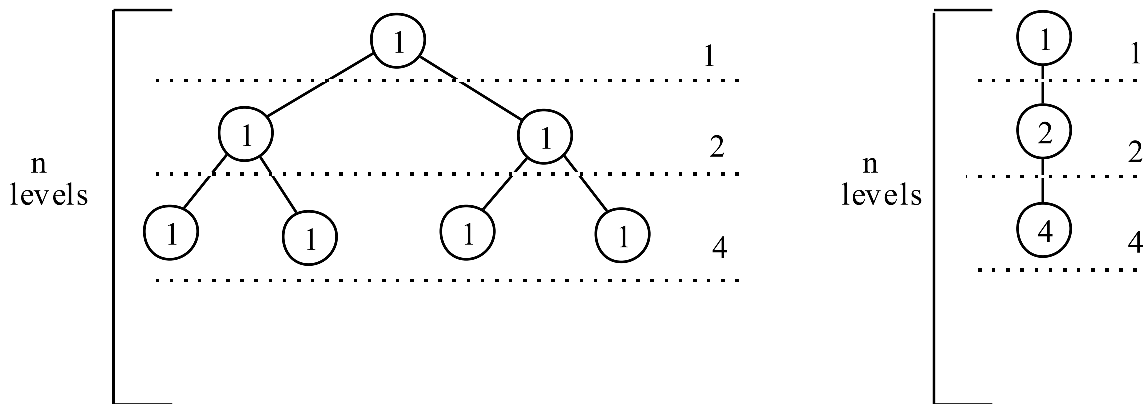
* Adapted from a handout by Lyn Turbak and Ellen Hildreth at Wellesley College

Exponential Functions

More generally, the tree constructed from a recurrence equation may have branching. Consider the following equations, which model the time taken to solve the Tower of Hanoi problem for n disks:

$T(n) = 1 + 2 \cdot T(n - 1), n > 0$
$T(0) = 0, n \leq 0$

This corresponds to either of the following two trees:



Adding up the cost of each level of the trees gives $T(n) = 1 + 2 + 4 + 8 + ... + 2^{n-1}$. Such a sequence, in which each element is the result of multiplying the previous element by a constant c, is called a ***geometric series***. How do we sum such a sequence? Consider the summation of a geometric series with m + 1 terms:

$$S(m) = a + ac + ac^2 + ac^3 + ... + ac^m$$

Notice the following:

$$\begin{aligned} cS(m) &= \quad\; ac + ac^2 + ac^3 + ... + ac^m + ac^{m+1} \\ -\,S(m) &= a + ac + ac^2 + ac^3 + ... + ac^m \end{aligned}$$

$$(c - 1)\, S(m) = ac^{m+1} - a = a(c^{m+1} - 1)$$

$$S(m) = \frac{a(c^{m+1} - 1)}{(c - 1)}$$

In this case of T(n) for trees, a = 1, c = 2, and m = n-1, so $T(n) = 2^n - 1$. Such a function is said to be ***exponential*** because the n appears in the exponent position. Exponential functions grow much more quickly than polynomial functions (functions in which n only appears as the base of exponents). Exponential functions typically arise in the context of recursive algorithms in which a problem is divided into multiple subproblems, each of whose size differs from the size of the original problem by a constant.

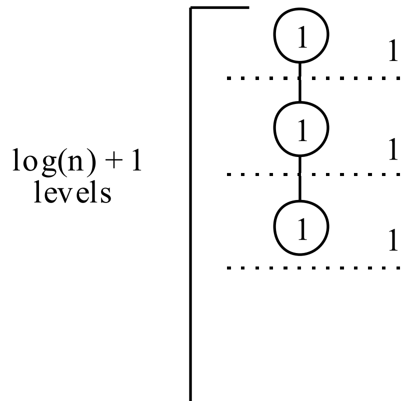
Logarithmic Functions

The "inverse" of an exponential function is a logarithmic function. Logarithmic functions are specified by recurrence equations like the following, which describes processes like binary search:

$T(n) = 1 + T(n/2), n \geq 1$
$T(n) = 0, n < 1$

Logarithmic functions typically arise in the context of algorithms in which a problem reduces to a single subproblem whose size is a fraction of the size of the original.

Such a set of equations generates a simple tree with $\log_2(n) + 1$ levels, each level of which contributes 1 to the total cost:



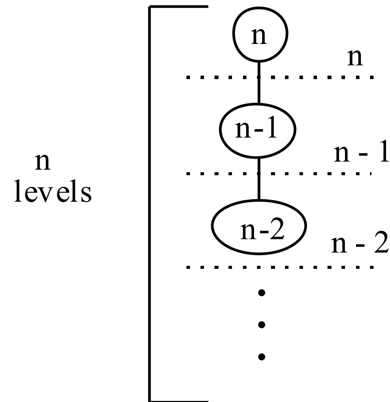
The total cost represented by such a tree is $T(n) = \log_2(n) + 1 \in O(\log_2(n))$. It turns out that the particular base of the logarithm does not matter: $O(\log_a(n)) = O(\log_b(n))$ for any a and b. (In contrast, the base of an exponent does matter: the functions in $O(2^n)$ are strictly smaller than those in $O(3^n)$, so $O(2^n) \neq O(3^n)$.)

Quadratic Functions

Quadratic processes result from recurrence equations like the following:

$T(n) = n + T(n - 1), \; n > 0,$
$T(n) = 0, \; n \leq 0$

These equations give rise to the following recurrence tree:



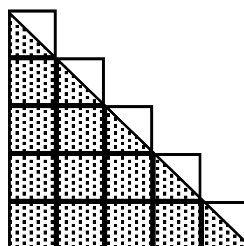
Determining the cost of such a tree requires calculating the sum:

$$1 + 2 + 3 + \ldots (n - 2) + (n - 1) + n$$

This is an arithmetic series: one in which each element is an additive constant larger than the previous. A little algebraic trick helps to solve such summations. Note that the first and last elements sum to (n + 1), the second and next-to-last elements also sum to (n + 1), and so on. Thus, there are n/2 pairs of elements that sum to n + 1, so the total sum must be

$$T(n) = \frac{n(n + 1)}{2}$$

Below is a picture that may help you remember this result. Each square represents 1 unit of the summation; for an n x n figure, the area of the figure is the desired summation. The shaded area is $\frac{n^2}{2}$, while the unshaded area is $\frac{n}{2}$. The sum of these two areas is the desired result.



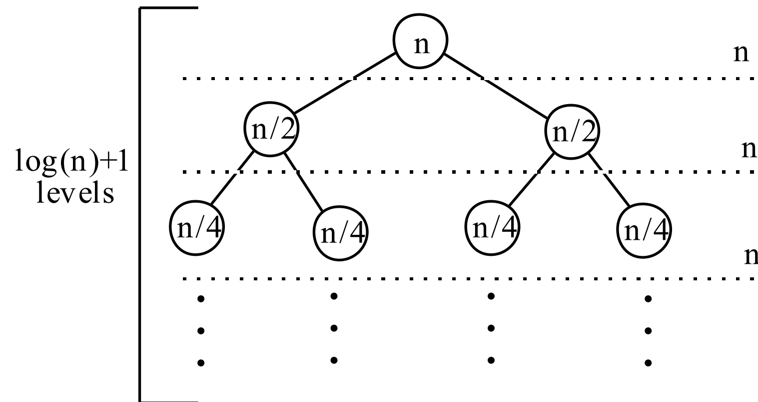
Note that this T(n) is a member of $O(n^2)$. This running time is typical of an algorithm that involves n instances of a linear process, e.g. naive approaches to sorting a list or array.

n log(n) Functions

There are many other kinds of recurrence equations that show up in practice. As a final example, we will consider a particularly important one that gives rise to a class of functions situated between linear and quadratic:

$$T(n) = n + 2T(n/2), n \geq 1$$
$$T(n) = 0, n < 1$$

These generate the following recurrence tree:



There are log(n) + 1 levels in the tree, each of which contributes n to the total sum. So the total sum is $n \cdot (\log(n) + 1) = O(n \log n)$. This is the running time of many algorithms that improve upon naive quadratic algorithms. As indicated by the form of the recurrence equations, the trick is to divide the given problem into two problems, each half the size, such that the cost of dividing the problem and combining the results is linear in the size of the problem. We will study several important algorithms that have this form.

Summary

The following table summarizes the above information. Equivalence classes of functions are arranged in the table from "biggest" to "smallest". That is, if function f appears above function g in the table, then g is O(f) but f is not O(g). It is worth noting that there are many more equivalence classes than are listed in the table, but the ones in the table are the ones most commonly encountered in this course.

Equivalence class of functions	Name	Typical Recurrence Equations
$O(3^n)$	exponential (base = 3)	$T(n) = a + 3 \cdot T(n - b), n > 0, a > 0, b > 0$ $T(0) = 0, n \leq 0$
$O(2^n)$	exponential (base = 2)	$T(n) = a + 2 \cdot T(n - b), n > 0, a > 0, b > 0$ $T(0) = 0, n \leq 0$
$O(n^2)$	quadratic	$T(n) = a \cdot n + T(n - b), n > 0, a > 0, b > 0$ $T(n) = 0, n \leq 0$
$O(n \log(n))$	n log(n)	$T(n) = n + k \cdot T(n/k), n \geq 1, b > 1, k > 1$ $T(n) = 0, n < 1$
$O(n)$	linear	$T(n) = a + T(n - b), n > 0, a > 0, b > 0$ $T(n) = 0, n \leq 0$
$O(\log(n))$	logarithmic (base is irrelevant)	$T(n) = 1 + T(n/k), n \geq 1, k > 1$ $T(n) = 0, n < 1$
$O(1)$	constant	$T(n) = c$