CIS 5511. Programming Techniques

Growth of Functions

A central topic in algorithm analysis is to compare the time complexity (or efficiency) of algorithms (for the same problem).

As the running time of an algorithm is represented as *T(n)*, a function of instance size, the problem is transformed into the categorization of these functions.

1. Asymptotic notations

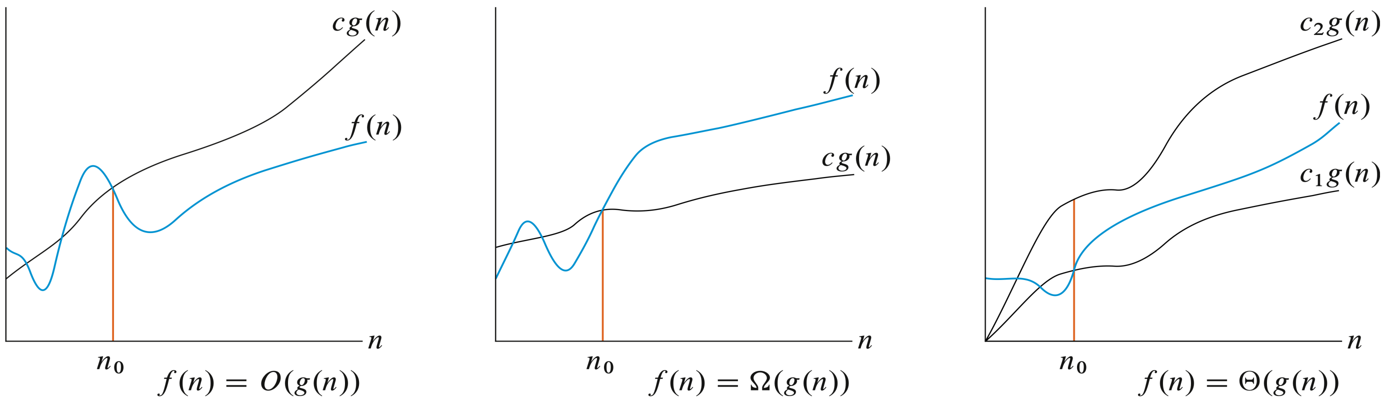
To simplify the analysis of the time expense of algorithms, the following assumptions are usually made:

1. A *size* measurement can be established among the (infinite number of) instances of a problem to indicate their *difficulty*, so that the *time* spent by an algorithm on an instance is a *nondecreasing function* of its size (at least after a certain point);
2. To compare the efficiency of algorithms on all problem instances, what matters is the instances larger than a certain size, as there are only finite smaller ones, but infinite larger ones since *size* has no upper bound;
3. No matter where the threshold is set, in general the algorithm that grows faster will cost more than another one that grows slower, so what really matters is not T(n), but its derivative T'(n);
4. Since it is impossible to decide or compare the growth rates of all functions, it is often enough to categorize them by *order of growth*, with simple functions identifying the categories.

The *asymptotic* efficiency of an algorithm is studied, that is, how its running time increases with the size of input *in the limit*, roughly speaking.

- $\Theta(g(n))$ = {f(n) : There exist positive constants c_1, c_2, and n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0$}. We say that g(n) is an *asymptotically tight bound* of f(n), and write $f(n) = \Theta(g(n))$. [It actually means $f(n) \in \Theta(g(n))$. The same for the following.]
- $O(g(n))$ = {f(n) : There exist positive constants c and n_0 such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0$}. We say that g(n) is an *asymptotically upper bound* of f(n), and write $f(n) = O(g(n))$.
- $\Omega(g(n))$ = {f(n) : There exist positive constants c and n_0 such that $0 \le cg(n) \le f(n)$ for all $n \ge n_0$}. We say that g(n) is an *asymptotically lower bound* of f(n), and write $f(n) = \Omega(g(n))$.
- $o(g(n))$ = {f(n) : For any positive constant $c > 0$, there exist a constant $n_0 > 0$ such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0$}. We say that g(n) is an *upper bound* of f(n) but *not asymptotically tight*, and write $f(n) = o(g(n))$.
- $\omega(g(n))$ = {f(n) : For any positive constant $c > 0$, there exist a constant $n_0 > 0$ such that 0

≤ cg(n) ≤ f(n) for all n ≥ n_0}. We say that g(n) is a *lower bound* of f(n) but *not asymptotically tight*, and write f(n) = ω(g(n)).



(a) (b) (c)

Examples:

- $4n^3 - 10.5n^2 + 7n + 103 = \Theta(n^3)$
- $4n^3 - 10.5n^2 + 7n + 103 = O(n^3)$
- $4n^3 - 10.5n^2 + 7n + 103 = \Omega(n^3)$
- $4n^3 - 10.5n^2 + 7n + 103 = O(n^4)$
- $4n^3 - 10.5n^2 + 7n + 103 = \Omega(n^2)$
- $4n^3 - 10.5n^2 + 7n + 103 = o(n^4)$
- $4n^3 - 10.5n^2 + 7n + 103 = \omega(n^2)$

Intuitively, the five notations correspond to relations =, ≤, ≥, <, and >, respectively, when applied to compare growth order of two functions (though not all functions are asymptotically comparable). All of the five are *transitive*, the first three are *reflexive*, and only the first one is *symmetric*.

Relations among the five:

- f(n) = Θ(g(n)) if and only if f(n) = O(g(n)) and f(n) = Ω(g(n)).
- f(n) = O(g(n)) if and only if f(n) = Θ(g(n)) or f(n) = o(g(n)).
- f(n) = Ω(g(n)) if and only if f(n) = Θ(g(n)) or f(n) = ω(g(n)).
- f(n) = O(g(n)) if and only if g(n) = Ω(f(n)).
- f(n) = o(g(n)) if and only if g(n) = ω(f(n)).

One way to distinguish them:

- f(n) = Θ(g(n)) if and only if Lim[f(n)/g(n)] = c (a positive constant).
- f(n) = o(g(n)) if and only if Lim[f(n)/g(n)] = 0.
- f(n) = ω(g(n)) if and only if Lim[f(n)/g(n)] = ∞.

The common growth orders, from low to high, are:

- constant: Θ(*1*)
- logarithmic: Θ($\log_a n$), *a* > 1
- polynomial: Θ(n^a), *a* > 0

- exponential: $\Theta(a^n)$, $a > 1$
- factorial: $\Theta(n!)$

Polynomial functions can be further divided into *linear* ($\Theta(n)$), *quadratic* ($\Theta(n^2)$), *cubic* ($\Theta(n^3)$), and so on.

Asymptotic notions can be used in equations and inequalities, as well as in certain calculations. For example,
$2n^2 + 3n + 1 = \Theta(n^2) + \Theta(n) + \Theta(1) = \Theta(n^2)$.

In algorithm analysis, the most common conclusions are worst-case expenses expressed as O(g(n)), which can be obtained by focusing on the most expensive step in the algorithm, as in the analysis of Insertion Sort algorithm.

2. Analysis of recurrences

When an algorithm contains a recursive call to itself, its running time can often be described by a recurrence, i.e., an equation or inequality that describes a function in terms of its value on smaller inputs.

A "divide-and-conquer" approach often uses *D(n)* time to divide a problem into a number *a* of smaller subproblems of the same type, each with *1/b* of the original size. After the subproblems are solved, their solutions are combined in *C(n)* time to get the solution for the original problem. This process is repeated on the subproblems, until the input size becomes so small (less than a constant *c*) that the problem can be solved directly (usually in constant time) without recursion. This approach gives us the following recurrence:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c \text{ ,} \\ aT(n/b) + D(n) + C(n) & \text{otherwise .} \end{cases}$$

For example, for the Merge Sort algorithm, we have

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \text{ ,} \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{if } n > 1 \text{ .} \end{cases}$$

This is the case, because each time an array of size *n* is divided into two halves, and to merge the results together takes linear time.

Often we need to solve the recurrence, so as to get a running time function without recursive call.

One way to solve a recurrence is to use the *substitution method*, which uses mathematical induction to prove a function that was guessed previously.

For example, if the function is

$$T(n) = 2T(\lfloor n/2 \rfloor) + n$$

one reasonable guess is T(n) = O(*n* lg *n*). To show that this is indeed the case, we can prove that T(n) $\le$ *c n* lg *n* for an appropriate choice of the constant *c* > 0. We start by assuming that this bound holds for halves of the array, then, substituting into the recurrence yields,

$$\begin{aligned} T(n) &\le 2(c\lfloor n/2\rfloor \lg(\lfloor n/2\rfloor)) + n \\ &\le cn\lg(n/2) + n \\ &= cn\lg n - cn\lg 2 + n \\ &= cn\lg n - cn + n \\ &\le cn\lg n\ , \end{aligned}$$

where the last step holds as long as c ≥ 1.

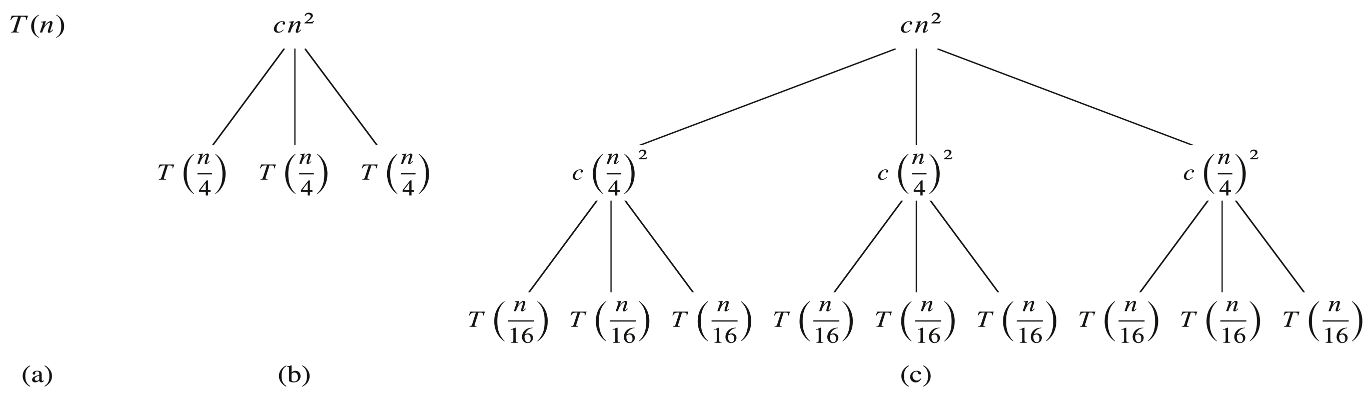
Furthermore, mathematical induction requires us to show that the solution holds for the boundary cases, that is, we can choose the constant *c* large enough so that T(n) ≤ *c n* lg *n* holds there. For this example, if the boundary condition is T(1) = 1, then c = 1 does not work there because *c n* lg(n) = 1 1 lg(1) = 0. To resolve this issue we only need to show that for n = 2, we do have T(n) ≤ *c* 2 lg 2 as long as c ≥ 2.

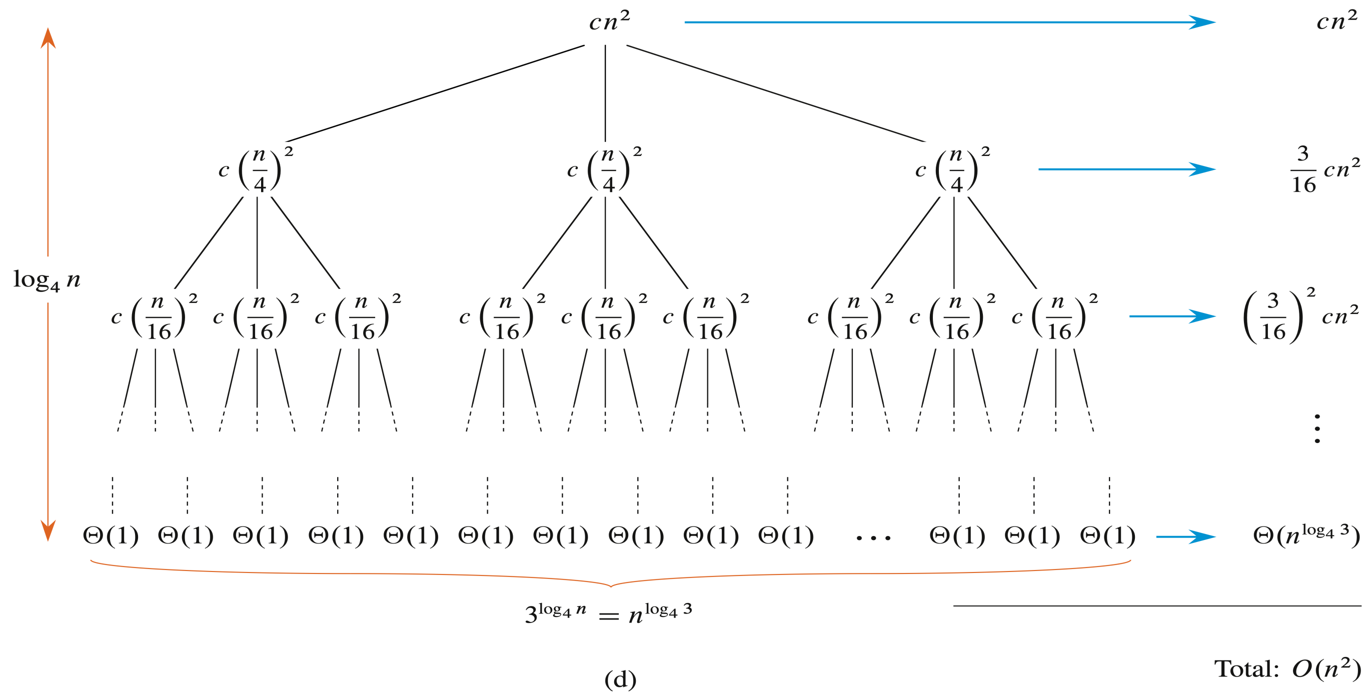
In summary, we can use *c ≥ 2*, and we have shown that T(n) ≤ *c n* lg *n* for all *n* ≥ 2.

To get a good guess for recurrence function, one way is to draw a *recursion tree*. For example, if the function is

$$T(n) = 3T(\lfloor n/4\rfloor) + \Theta(n^2).$$

we can create the following recursion tree:


$T(n)$
cn^2
$T\left(\frac{n}{4}\right)$
$c\left(\frac{n}{4}\right)^2$
$T\left(\frac{n}{16}\right)$
(a)
(b)
(c)


cn^2
$c\left(\frac{n}{4}\right)^2$
$c\left(\frac{n}{16}\right)^2$
$\log_4 n$
$\Theta(1)$
$\frac{3}{16}cn^2$
$\left(\frac{3}{16}\right)^2 cn^2$
$\Theta(n^{\log_4 3})$
$3^{\log_4 n} = n^{\log_4 3}$
Total: $O(n^2)$
(d)

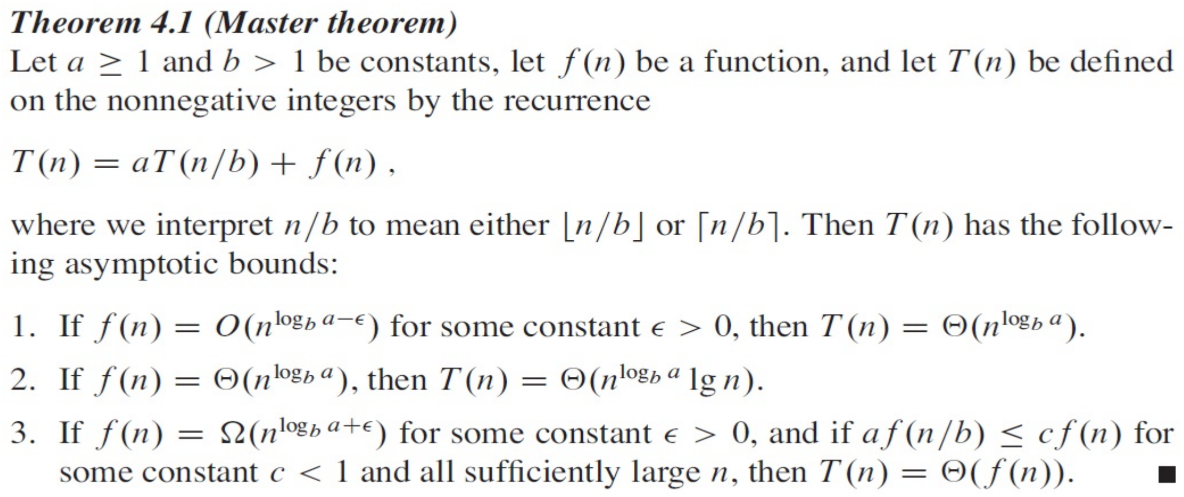
To determine the height of the tree i, we have $n / 4^i = 1$, that is, $i = \log_4 n$. So the tree has $\log_4 n + 1$ levels.

At level i, there are 3^i nodes, each with a cost $c(n / 4^i)^2$, so the total is $(3/16)^i cn^2$.

At the leaves level, the number of nodes is $3^{\log_4 n}$, which is equal to $n^{\log_4 3}$ (because $a^{\log_b c} = c^{\log_b a}$). At that level, each node costs T(1), so the total is $n^{\log_4 3}T(1)$, which is $\Theta(n^{\log_4 3})$.

After some calculation (see Section 4.4 of the textbook for details), the guess $T(n) = O(n^2)$ can be obtained.

Finally, there is a "master method" that provides a general solution for divide-and-conquer recurrence. Intuitively, the theorem says that the solution to the recurrence is determined by comparing the costs at the top and bottom levels of the recursion tree:



Theorem 4.1 (Master theorem)

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n) ,$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

The above example falls into Case 3.

The proof of the master theorem is given in the textbook. It is important to realize that the three cases of the master theorem do not cover all the possibilities.

As a special case, when $a = b$ we have $\log_b a = 1$. The above results are simplified to (1) Θ(n), (2) Θ(n lg n), and (3) Θ(f(n)), respectively, depending on whether f(n) is linear, or faster/ slower than linear. We can see that Merge Sort belongs to Case 2.