Parma University, Math Department

Algorithms and Data Structures
(Asymptotics and Recursions)

prof. Marco Pellegrini

Spring 2002, lecture 2

Asymptotics

Asymptotic notation aims at a classification of functions according to their behaviour at infinity. Let f,g:$\mathbb{N}\to\mathbb{R}$.

$$f(n) \prec g(n) \overset{def}{\Leftrightarrow} \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

If both f and g diverge to infinity, then *informally* g goes to infinity *faster* than f.

(a) transitivity: f(n)≺g(n) and g(n)≺h(n) implica f(n)≺h(n).

(b) For arbirary α e β, $n^{\alpha} \prec n^{\beta} \Leftrightarrow \alpha < \beta$.

(c) $1 \prec f(n) \prec g(n) \Rightarrow e^{|f(n)|} \prec e^{|g(n)|}$

A typical hierarchy of functions (for any 0<ε<1<c):

$$1 \prec \log\log n \prec \log n \prec n^{\varepsilon} \prec n^{c} \prec n^{\log n} \prec n^{n} \prec c^{c^{n}}$$

Exercise: prove that, for every ε>0:

$$\log n \prec e^{\sqrt{\log n}} \prec n^{\varepsilon}$$

We define the relation f(n)≈g(n) when:

$$\exists C, n_0 \forall n > n_0 \,|\, f(n)| \leq C\,|g(n)| \wedge |g(n)| \leq C\,|f(n)|$$

We define the relation f(n)~g(n) when:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$$

The second relation is stronger, the first easier to prove.

Logarithmic-exponential functions

G.H. Hardy defined the following class $\mathcal{L}$:

1) For any real α, the constant function f(n)= α is in $\mathcal{L}$.

2) The identity function f(n)=n is in $\mathcal{L}$.

3) If f(n) and g(n) are in $\mathcal{L}$, also f(n)-g(n) is in $\mathcal{L}$.

4) If f(n) is in $\mathcal{L}$, also exp(f(n)) is in $\mathcal{L}$.

5) If f(n) is in $\mathcal{L}$ and is eventually positive, also ln f(n) is in $\mathcal{L}$.

Property: Every function in $\mathcal{L}$ is either eventually positive, or eventually negative or identically null.

Asymptotic Hierarchy property:

For any two functions f(n) and g(n) in $\mathcal{L}$ always *one and only one* of these cases is true:

f(n) ≺g(n), g(n) ≺f(n), g(n) ≈f(n).

Moreover, if g(n) ≈f(n) there exits c such that f(n) ~cg(n).

Thus within the class $\mathcal{L}$ w can always compare two functions accorcing to the ≺ and ≈ relations.

Exercise: show that the sum, product and quotient of two functions in $\mathcal{L}$ is also in $\mathcal{L}$.

Big O, Big Ω, Big Θ

$$O(g(n)) = \{f(n) \mid \exists C, n_0 \forall n > n_0 : |f(n)| \leq C\,|g(n)|\}$$

$$\Omega(g(n)) = \{f(n) \mid \exists C, n_0 \forall n > n_0 : 0 < C\,|g(n)| \leq |f(n)|\}$$

$$\Theta(g(n)) = \{f(n) \mid \exists C_1, C_2, n_0 \forall n > n_0 : 0 < C_1\,|g(n)| \leq |f(n)| \leq C_2\,|g(n)|\}$$

Note that O, Ω, and Θ are sets of functions. When we find in an expression a notation O, Ω, or Θ this indicates a generic unspecified element of that set.

The advantage in using O, Ω, and Θ is

1) we often avoid specifying constants.

2) we can use pseudo-arithmetic to simulate set operations.

3) easy composition rule.

Disadvantage:

4) some expressions/manipualtions can be meaningless

For example: $3n^3 = O(n^3)$ is true, $O(n^3) = 3n^3$ is meaningless, since here = simulates the $\in$ relation.

O(.) is a *coarsification* of the information.

(a) $f(n) = O(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$.

(b) $f(n) = \Theta(g(n)) \Leftrightarrow f(n) = \Omega(g(n))$ and $f(n) = O(g(n))$.

(c) $f(n) = \Theta(g(n)) \Leftrightarrow f(n) \approx g(n)$.

(d) $f(n) < g(n) \Rightarrow f(n) = O(g(n))$.

(a) $n^h = O(n^k)$ for every $k > h$.

(b) $n^h \notin O(n^k)$ for every $k < h$.

(c) $O(f(n))+O(g(n)) = O(|f(n)|+|g(n)|)$.

(d) $O(O(f(n))) = O(f(n))$.

(e) $O(f(n))O(g(n)) = O(f(n)g(n))$.

(f) $O(f(n)g(n)) = f(n)O(g(n))$.

Asymptotics and Time functions

For algorithm A, $T_A^W(n) = \Theta(f(n))$ has the following meaning:

(a) there is a C_1 and n_0 so that, *for every* $n > n_0$ and *for every* $x \in I$ with $|x| = n$, $T(x) \leq C_1 f(n)$.

(b) there is a C_2 and n_0 so that, *for every* $n > n_0$ and *some* $x \in I$ with $|x| = n$, $T(x) \geq C_2 f(n)$.

So for every length class in I it is sufficient to find a bad input to pump up the lower bound.

(b) is called a *worst case lower bound for algorithm A*.

For algorithm A, $T_A^B(n) = \Theta(f(n))$ has the following meaning:

(a) there is a C_1 and n_0 so that, *for every* $n > n_0$ and *some* $x \in I$ with $|x| = n$, $T(x) \leq C_1 f(n)$.

(b) there is a C_2 and n_0 so that, *for every* $n > n_0$ and *for every* $x \in I$ with $|x| = n$, $T(x) \geq C_2 f(n)$.

Therefore the lower bound (b) holds on *all* inputs. This is called a *best case lower bound for Algorithm A*.

Problems and Algorithms

Given an algorithm A, this computes a function $g_A: I \to \Omega$, from an input state we obtain an output state. The function g_A, is called *computable*.

On the other hand we might want an algorithm that computes a specified (computable) function P (that it, solves a specified problem P). $P: \mathbb{I} \to \mathbb{O}$.

We say that algorithm A ***solves*** problem P if, for every element $X \in \mathbb{I}$

$$decode(g_A(encode(X)) = P(X).$$

Where *encode*: $\mathbb{I} \to I$, and *decode*: $\Omega \to \mathbb{O}$, are often only conceptual, but sometimes may require a computation.

Lower bound for problems.

We say that a *problem* P has worst case lower bound $\Omega(f(n))$ in a machine model M, if:

for every algorithm A in machine model M *solving* P:

$$T^{W}_{AM}(n) = \Omega(f(n)) .$$

Capturing all algorithms solving a given problem P is rather difficult and non trivial lower bound results on problems are difficult to get (however you should know the main results).

For a *w.c. upper bound* on problem P it is sufficient to find one algorithm A solving P and take its w. c. upper bound.

Recursive equations

Recursive programs are analyzied with the help of *recursive equations*. The anlysis is in two steps:

(1) Examine the algorithm to derive the equation.

(2) Solve the equation.

Here we suppose (1) has been done and discuss how to do (2).

Example: $$T(n) \le \begin{cases} \Theta(1) & n = 1 \\ 2T(n/2) + \Theta(n) & n > 1 \end{cases} \qquad \text{(A)}$$

$T:\mathbb{N}\to\mathbb{N}$ is an unknown function that is constrained to satisfy the recursive equation.

(a) If T is the worst case time function, we may write = or $\le$ to get an upper bound,

(b) If T is the worst case time function, we may write = or $\ge$ to get an lower bound,

(c) T might be re-defined only on a subset of $\mathbb{N}$, for example here the set of powers of two $\{2^k | k\in\mathbb{N}\}$ or the equation is re-written:

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

(d) Experience has shown that often solving (A) on a restricted set is easier than solving (B) on $\mathbb{N}$ and leads to the same result.

(e) If we use the asyptotic notation in a recursive equation we define a *set of solution functions* (which we express again in asymptotic notation)

General strategies for solving r.e.

(1) Substitution method (verification by induction).

(2) Change of Domains and Ranges.

(3) Iteration method (summation method).

(4) Recursion tree.

(5) Master theorem.

Substitution method

(1) Guess a solution.

(2) Test by induction that it satisfies the recursive inequality.

(3) Important: *symbolic constants hidden in asymptotic notation must be made explicit*.

Example: $$T(n) \le \begin{cases} 1 & n = 1 \\ 2T(\lfloor n/2 \rfloor) + n & n > 1 \end{cases}$$

Guess: T(n)=O(n *log* n)+O(1), that is $\exists$c,d such that

$$T(n) \le c\, n \log n + d.$$

Proof that the guess is right.

(a) Base case (n=1): T(1)=1 ≤ c1log1 +d=d. It is true for d≥1.

(b) Inductive hypothesis. T(m) ≤ cm *log* m +d for every m<n.

Now we show it is true for n>1.

$$T(n) \le 2T(\lfloor n/2 \rfloor) + n \le 2c\lfloor n/2 \rfloor \log\lfloor n/2 \rfloor + 2d + n$$

Upper bounding!

$$2c\lfloor n/2 \rfloor \log\lfloor n/2 \rfloor + 2d + n \le cn\log(n/2) + 2d + n$$

Because $\lfloor a \rfloor \le a$

$$cn\log(n/2) + 2d + n = cn\log n + n(1-c) + 2d$$

$$cn\log n + n(1-c) + 2d \le cn\log n + d$$

True if n(1-c)+d ≤0.

Take d=1 and c=2, so n(1-c)+d ≤0 for all n ≥ 1.

(a) At the end we get exactly the thesis (the guess).

(b) We have a chain of ≤ joining the first term T(n) to the last.

(c) Collect constraints on the constants c,d and stisfy them.

Important:

(a) *never* use asymptotics in the thesis (guess).

Strategies for guessing:

(a) look for solutions of similar equations.

(b) try lower bounds and upper bound guesses and try to make them meet.

(c) Use in your guess enough free constants.

Change of Domain and Range

Example: Consider this strange looking recurrence:

$$T(n) \le 2T(\sqrt{n}) + \log n$$

Set m = log n. The recurrence becomes:

$$T(2^m) \le 2T(2^{m/2}) + m$$

Set $T(2^m)=S(m)$. The recurrence becomes:

$$S(m) \le 2S(m/2) + m$$

This has solution S(m)=O(m log m), so T(n)=O(log n log log n)

Iteration method

The idea is to develpe the inequality into a sum. We need to find out:

(a) The generic term of the sum.

(b) A bound on the nunber of terms of the sum.

Example: $$T(n) = \begin{cases} \Theta(1) & n \le 3 \\ 3T(\lfloor n/4 \rfloor) + n & n \ge 4 \end{cases}$$

Developing:

$$\begin{aligned} T(n) &\le n + 3T(\lfloor n/4 \rfloor) \\ &\le n + 3(\lfloor n/4 \rfloor + 3T(\lfloor n/16 \rfloor)) \\ &\le n + 3\lfloor n/4 \rfloor + 9(\lfloor n/16 \rfloor + 3T(\lfloor n/64 \rfloor)) \end{aligned}$$

(a) The generic term of the is $3^i\lfloor n/4^i \rfloor$.

(b) The argument of T(.) is ≤1 after $k=\lceil \log_4 n \rceil$ iterations.

Therefore:

$$T(n) \le n\left(\sum_{i=0}^{k} \left(\tfrac{3}{4}\right)^i\right) + \Theta(1)3^k$$

Since $3^{\log_4 n} = n^{\log_4 3} \prec n$

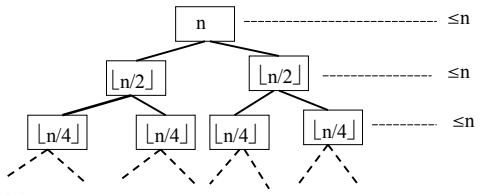
And $\sum_{i=0}^{k} \left(\tfrac{3}{4}\right)^i \le \sum_{i=0}^{\infty} \left(\tfrac{3}{4}\right)^i = 4$

We get T(n) = O(n)

Recursion Tree

Recursion trees develope a recursive equation in a tree and makes summations level by level.

Example: $T(n) = \begin{cases} \Theta(1) & n \le 1 \\ 2T(\lfloor n/2 \rfloor) + n & n \ge 2 \end{cases}$



(a) The sum of contributions at each level od at most n.

(b) The nubmer of levels is at most $\lceil \log_2 n \rceil$.

The total cost is T(n) = O(n log n).

The master method

This is a theorem that gives the answer for all recurrences of the form:

$$T(n) = \begin{cases} \Theta(1) & n \in \Theta(1) \\ aT(n/b) + f(n) & otherwise \end{cases}$$

Where $a \ge 1$,$b>1$, f(n) is eventually positive and n/b stands either for $\lceil n/b \rceil$ or $\lfloor n/b \rfloor$.

Set: $x = \log_b a$

Informally, we compare f(n) with n^x, and we take the largest.

Master theorem:

(a) If $f(n)=\Theta(n^x)$, then ***$T(n) = \Theta(n^x \log n)$.***

(b) If there exists a constant $\varepsilon>0$ such that $f(n)=O(n^{x-\varepsilon})$, then ***$T(n) = \Theta(n^x)$.***

(c) If there exists a constant $\varepsilon>0$ such that $f(n)=\Omega(n^{x+\varepsilon})$, then ***$T(n) = \Theta(f(n))$.*** Provided f(.) satifies this condition: there is a constant c such that, for n sufficiently large, *$af(n/b) \le cf(n)$.*

Conclusions

(a) *Asymptotic notation* (relations, expressions) allows to compare the order of growth of functions even if not completely specified.

(b) *Recursive equations* are a basic tool to derive the time bounds of recursive programs. We have seen a few basic solution techniques.