Recurrences
-draft-

M. Pellegrini*

March 6, 2002

1 Thomas H. Cormen, Charles Eric Leiserson and Ronald L. Rivest

Thomas H. Cormen and Charles Eric Leiserson and Ronald L. Rivest [CLR90, chapter 4] discuss general technique and issues in solving recurrence equations. A recurrence relation is an equation in which the unknown quantity is a function of one variable (or more) subject to additional boundary constraints. Typically recurrence equations are derived diring the analysis of recursive programs (i.e. programs that call themselves as subroutines). A typical example is the following equation derived from the analysis of the *merge-sort* algorithm. $T(n)$ is the unknown function, that is the worst case running time of merge-sort.

$$T(n) \leq \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1. \end{cases} \quad (1)$$

Using inequalities is appropriate when known functions are described explicitly or unknown constants are spelled out during the derivation, however if we use asymptotic notation we can also write an $=$, the inequality being implicit in the asymptotic notation. The above equation is for an unknown function $T : N \to N$, however, since $n/2$ might not be an integer number, it is safely defined only on a subset of N, namely the set $P = \{2^k | k \in N\}$ of all powers of two. Thus also the set of inputs of the algorithm is restricted to be the set of subsets of N whose cardinality is in P. One might solve this equation and then claim the bound holds for any input by invoking a preliminary phase of padding with dummy input data, and a successive phase of stripping the dummy data from the output. The overall bound being the sum of the bounds of the three phases. This trick makes sure that there is an algorithm

*Istituto di Matematica Computazionale del CNR, pellegrini@imc.pi.cnr.it

that matches this analysis. In practice one would round $n/2$ the input to the closest integer, thus the corresponding equation would be:

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

Experience has shown that in most cases solving an equation like (2) involving the rounding up and rounding down operators is more laborious and yields the same result as solving the corresponding simpler equation (1). However sometimes it does make a difference. The boundary conditions usually involve giving an upper bound to the unknown function for small values of n. We are interested in the behaviour of the function for $n > n_0$ for a constant n_0 we are free to specify, this can be used to avoid specific small values of n for which auxiliary functions might not be defined. In general it is a safe assumption that $T(n) = \Theta(1)$ for $n = \Theta(1)$.

Recursive Equations and Dis-Equations. Note that the collection of (1) a recursive *equation*, (2) boundary conditions, and (3) specification of relevant ranges and domains with no asymptotic notation or symbolic unknown constants involved does have one and only one solution. In such condition the equation itself can be seen as the specification of an algorithm that specifies point-wise the function. In this case finding a closed form of such a function is the challenge. Recursions written in order to analyze algorithms usually are dis-equalities and involve unknown constants (either explicit or implicit in asymptotic notation). In general we have an infinite set of solutions and we are interesting in finding among those solutions the best one, (even if only in asymptotic terms). In particular, if both $T(n) = O(f(n))$ and $T(n) = O(g(n))$ are solutions of a dis-equality, and $f(n) = O(g(n))$, then the solution $T(n) = O(f(n))$ is a more precise characterization.

Bits of Advise. Although boundary conditions are a minor part in the solution of the recurrence it is well possible that writing the code handling the corresponding boundary inputs absorbs most of the time and effort of the programmer. Proper handling of boundary cases is as essential for correctness as any other part of the algorithm.

1.1 Substitution method

This method, which we might also call *verification by induction*, is indeed a method for verifying a guess of a candidate solution. Example:

$$T(n) \leq \begin{cases} 1 & \text{if } n = 1, \\ 2T(\lfloor n/2 \rfloor) + n & \text{if } n > 1. \end{cases} \tag{3}$$

Guess: $T(n) = O(n \log n) + O(1)$. From the definition this is equivalent to the thesis of the existence of a constant c and d such that $T(n) \leq cn \log n + d$, for $n \geq 1$. Verification by induction. Base case. Since from the equation $T(1) = 1$, the thesis is true for $n = 1$, provided $d \geq 1$. Inductive step. Assume $n > 1$ and that the thesis

true for any m in the range $1 \leq m < n$ and prove that this implies that it is true for n. Since $\lfloor n/2 \rfloor < n$ we can write:

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

where the passage is correct since we are upper bounding $2T(\lfloor n/2 \rfloor)$.

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

This passage is correct since for positive integers $\lfloor x \rfloor \leq x$.

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

By simple arithmetic and remembering that log is in base 2.

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

The last passage is true if $n(1 - c) + d \leq 0$, setting $d = 1$, this is true for every n when $c \geq 2$. By transitivity of the inequalities we see that:

$$T(n) \leq cn \log n + d$$

This is the same statement as the thesis, so the thesis is proved only using the inductive step and standard arithmetic considerations. ∎

Note that the technique works by finding a chain of upper bounds until the same expression as the thesis emerge as the last term. in the process we find constraints that the constants must satisfy to make the passages of the proof correct. Finally one has to check that there is an assignment of the constants that make all the collected constraints on the constants true simultaneously.

How to make good guesses. Guessing solutions takes experience. One can look at the tabulated solution of similar recursive equations. Or proceed iteratively from coarse to finer guesses. It is also wise to look for at upper bounds and lower bounds alternatively since we can limit the range of possible guess much more quickly. For example in the above case one might guess $T(n) = O(n^2)$ first, then $T(n) = O(n^{1+\epsilon})$. An obvious lower bound is $\Omega(n)$. Moreover sometimes it may be easier to prove a stronger thesis, rather than a weaker one, since we make the inductive hypothesis stronger too. This usually involves using additive terms (not only multiplicative ones) and having enough parameters to play with.

Exercise 1 *Try proving thesis (a) $T(n) \leq cn$ and (b) $T(n) \leq cn - d$ as a guessed solution for:*

$$T(n) \leq \begin{cases} 1 & \text{if } n = 1, \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + 1 & \text{if } n > 1. \end{cases} \tag{4}$$

Which one of the two thesis was easier to prove?

Very important. Never use asymptotic notation in the statement of the thesis, and do not make use of coarsification in the derivation of chain of inequalities. Always use symbolic constants. In inductive proofs constants are important and therefore must be manipulated explicitly.

1.2 Change of variable

By changing the domain and range of the known and unknown functions in a recursive equation we can make an odd looking equation look like a familiar one. Thus solve it and apply the inverse transformations to domains and ranges. [**Put an example**]

1.3 The iteration method

Also known as *telescoping*. This is applied by unwinding the recursion and expand it into the sum of an increasing number of terms, each depending only on n and the initial condition. This involves writing an arbitrarily long summation, which however we try to express in closed form. Finally we find an upper bound of the summation. Example:

$$T(n) \leq \begin{cases} \Theta(1) & \text{if } n \leq 3, \\ 3T(\lfloor n/4 \rfloor) + n & \text{if } n \geq 4. \end{cases} \tag{5}$$

Iterating we have:

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

where we have used the fact that for any integers n, $a \neq 0$ and $b \neq 0$, $\lfloor \lfloor n/a \rfloor / b \rfloor = \lfloor n/ab \rfloor$. Clearly we can proceed and at iteration i we add $3^i \lfloor n/4^i \rfloor$ to the summation. We will stop expanding for an i such that: $\lfloor n/4^i \rfloor \leq 1$, that is $i = \log_4 n$. At this point we can evaluate (upper bound) explicitly the reminder containing the occurrence of $T(.)$. All terms except the last one form a geometric series. This series is surely upper bounded by adding all elements up to infinity. The last term is $\Theta(1)(3^{\log_4 n})$. Eliminating the $\lfloor . \rfloor$ we just increase the summation. We get:

$$T(n) \leq n \left(\sum_{i=0}^{\infty} \left(\frac{3}{4} \right)^i \right) + \Theta(1)(3^{\log_4 n})$$

The first term is bounded by $4n$. The second term is equivalent to $\Theta(n^{\log_4 3}) \prec n$. Thus finally $T(n) = O(n)$.
This technique required familiarity with summations and algebraic manipulations. The two main ingredient are the generic term of the summation, and the maximum length of the summation.

1.4 The recursion tree

The above method relied on the fact that the summation was unfolding a term at a time and there was an obvious linear order to follow. Sometimes we can use to our

advantage a different ordering of the terms, for example by placing them on a tree that has the same structure as the tree of the recursion calls. Then, typically, we first sum terms level by level in the tree and we get a single term per level, then we sum the contributions of the levels. See [CLR90, pages 59-60] for a few examples.

1.5 The master method

The master theorem gives a "cookbook" solution to many recursions found in simple divide and conquer algorithms. Suppose that we solve our problem by dividing it into a equivalent sub-problems each of size n/b. Finding the subdivision and merging the results takes overall time $f(n)$. In this case the recursion will be of the form:

$$T(n) \leq \begin{cases} \Theta(1) & \text{if } n \in \Theta(1), \\ aT(n/b) + f(n) & \text{otherwise.} \end{cases} \tag{6}$$

where $a \geq 1$, $b > 1$ are non necessarily integer constants and $f(.)$ is eventually positive. Technically speaking n/b might not be an integer so we might have to place floors and ceiling in the above recursion, which however do not alter the following result:

Theorem 1 (Master) *Let $x = \log_b a$,*

1. *If there exists a constant $\epsilon > 0$ such that $f(n) = O(n^{x-\epsilon})$ then $T(n) = \Theta(n^x)$.*
2. *If $f(n) = \Theta(n^x)$ then $T(n) = \Theta(n^x \log n)$.*
3. *If there exists a constant $\epsilon > 0$ such that $f(n) = \Omega(n^{x+\epsilon})$, then $T(n) = \Theta(f(n))$, provided that f satisfies the following regularity condition: there exists a constant c, so that for every n sufficiently large: $af(n/b) \leq cf(n)$.*

In order to use this theorem we determine $x = \log_b a$ and then we compare asymptotically the functions n^x and $f(n)$. If the comparison is possible (which is always true in the $\mathcal{L}$ family of functions) there are three cases, two gaps, and a regularity condition to consider. The first case is when f is not only upper bounded by $O(n^x)$ but there is a polynomial separation, then n^x wins. The third case is when f is not only lower bounded by $\Omega(n^x)$ but there is a polynomial separation, and here we have th check the regularity of f. Finally if the two have the same rate of growth, there is an extra logarithmic factor in the solution.

Proof of the master theorem. It is in [CLR90, pages 64-72]. Briefly, the result is proved for the set of exact powers of b: $P_b = \{b^k | k \in N\}$. Then this solution is extended to all integers by finding an upper bound to the recursion:

$$T(n) \leq \begin{cases} \Theta(1) & \text{if } n \in \Theta(1), \\ aT(\lfloor n/b \rfloor) + f(n) & \text{otherwise.} \end{cases} \tag{7}$$

and a lower bound to

$$T(n) \leq \begin{cases} \Theta(1) & \text{if } n \in \Theta(1), \\ aT(\lceil n/b \rceil) + f(n) & \text{otherwise.} \end{cases} \tag{8}$$

The intuition is that in the recursion tree unfolding the cost is concentrated at the leaves, or at the root, or distributed evenly on the levels.

2 Other books

Recurrence equations are ubiquitous in the analysis of algorithms and usually their solution might need the use of advanced techniques like the *generating functions* technique [SF96],[Knu69]

References

[CLR90] Thomas H. Cormen, Charles Eric Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, Cambridge, Massachusetts, 1990.

[Knu69] D. E. Knuth. *The art of computer programming*, volume 1, 2, 3. Addison Wesley, 1969.

[SF96] Robert Sedgewick and Philippe Flajolet. *An Introduction to the Analysis of Algorithms*. Addison-Wesley, Reading MA, 1996.