A General Method for Solving Divide-and-Conquer Recurrences[1]

Jon Louis Bentley[2]
Dorothea Haken
James B. Saxe
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

Abstract

The approximate complexity of divide-and-conquer algorithms is often described by recurrence relations of the form

$$T(n) = kT(n/c) + f(n) .$$

The only well-defined method currently used for solving such recurrences consists of solution tables for fixed functions f and varying k and c. In this note we describe a unifying method for solving these recurrences that is both general in applicability and easy to apply. This method is appropriate both as a classroom technique and as a tool for practicing algorithm designers.

1. Introduction

The mathematical analysis of algorithms has proven to be an important subject of both practical and theoretical interest. One school within this field, championed by Professor D. E. Knuth of Stanford University, has concentrated on *exact* analyses. In addition to providing practical tools (i.e., the algorithms analyzed), this research has also produced much deep and beautiful mathematics. On the other hand, *approximate* analyses, although less substantial mathematically, are still of practical importance. In such analyses, only the leading term of the desired function is given, and that often only to within a multiplicative factor. It is this second type of analysis that we will study in this note.

The approximate running times of recursive algorithms can often be expressed by simple recurrence relations. Divide-and-conquer algorithms are an important subclass of recursive algorithms (see Aho, Hopcroft and Ullman [1974, Section 2.6] for a discussion of this class). The recurrence relations describing the approximate complexities of these algorithms as a function of the problem size, n, frequently

[1]This research was supported in part by the Office of Naval Research under Contract N00014-76-C-0370.

[2]Also with the Department of Mathematics.

have the form[3]

$T(1)$ given,

$T(n) = kT(n/c) + f(n)$.

Much work has been done on systematic ways of solving recurrences of this form; see, for example, Aho, Hopcroft and Ullman [1974, pp. 64-65], Borodin and Munro [1975, p. 80], Keller [1980], and Stanat and McAllister [1977, pp. 249, 255]. The only well-defined methods described by the above authors, however, consist of solution tables for fixed functions f and varying k and c. Although this method is quite satisfactory when applied to recurrences with common f, there are many times when a function f arises that has not been previously tabulated.

In this note we describe a general method for solving recurrences of the above form. Our method is based on rewriting the recurrence into a standard *template*, which can then be solved easily with the help of a small table. This template is illustrated for one example in Section 2, and then discussed in Section 3 for a more general case (though still confined to $c = 2$). In Section 4 we apply the template to solve a number of particular recurrences. The application of the template to the case that $c \neq 2$ is the subject of Section 5. Extending this work to solve a broader class of recurrences is the subject of Section 6, and conclusions are then offered in Section 7. The primary contribution of this paper is not in solving any particular new recurrences, but rather in providing a general and succinct method by which approximate solutions may be found for a large class of recurrences. It has already been the experience of the authors that this method is an excellent didactic tool in the classroom context.

2. A Simple Example

In this section we will examine the recurrence

$T(1) = 1$,

$T(n) = 2T(n/2) + n$.

Recurrences similar to this arise in the (approximate) analysis of the MergeSort and Fast Fourier Transform algorithms; both of these algorithms (and their recurrences) are described by Aho, Hopcroft and Ullman [1974].

To solve this recurrence we rearrange the recursive part as

$T(n) = n + 2T(n/2)$.

Expanding the recursive definition of $T(n/2)$ shows that

$T(n) = n + 2[n/2 + 2T(n/4)]$,

and multiplying and cancelling yields

$T(n) = n + n + 4T(n/4)$.

(We will see in the next section that this ability to cancel is critical.) Applying the definition of $T(n/4)$ to this expression shows that

$T(n) = n + n + n + 8T(n/8)$.

[3]Note that $T(n)$ is defined only when n is a power of c; the implications of this restriction are discussed by Aho, Hopcroft and Ullman [1974, p. 65].

This process of expansion and cancellation can be iterated lg n times to yield[4]

$$\begin{aligned} T(n) &= [\sum_{1\le i\le \lg n} n] + n\cdot T(1) \\ &= n\,[\sum_{1\le i\le \lg n} 1 + T(1)] \\ &= n\,[\lg n + 1]\,. \end{aligned}$$

Mathematical induction can now be used to show formally that this is indeed the unique solution to the above recurrence.

In the next section we will see how the technique used here to solve this recurrence can be used to solve many divide-and-conquer recurrences. The crucial "trick" in that section is to rewrite the recurrence so that successive terms cancel nicely as the recurrence is "iterated" to a solution.

3. The Template

In this section we will investigate recurrences of the form

$$\begin{aligned} &T(1) \text{ given}, \\ &T(n) = kT(n/2) + f(n). \end{aligned} \tag{1}$$

As mentioned before, this recurrence is defined only for n a power of two. To solve the recurrence we will rewrite it into the *template*

$$\begin{aligned} &T(1) \text{ given}, \\ &T(n) = 2^pT(n/2) + n^pg(n), \end{aligned} \tag{2}$$

where $p = \lg k$ and $g(n) = f(n)/n^p$. This new recurrence is easily solved using the "iteration and cancellation" technique of Section 2; it has the unique solution

$$T(n) = n^p[g(n) + g(n/2) + g(n/4) + \ldots + g(4) + g(2) + T(1)].$$

(Note that this solution is easy to express in terms of g because $n^p = 2^p\cdot(n/2)^p$; this is the generalization of the cancellation property of the previous section.) We abbreviate the above solution as

$$T(n) = n^p[T(1) + \tilde{g}(n)], \tag{3}$$

where $\tilde{g}$ is defined as

$$\tilde{g}(n) = \sum_{1\le i\le \lg n} g(2^i).$$

Mathematical induction on the powers of two can be used to show that Equation 3 is indeed the unique solution to the recurrence of Equation 2.

The above facts provide us with a method for solving any recurrence in the form of Equation 1. We cast it in the template of Equation 2 by dividing and taking a logarithm, and then the solution to the recurrence is given by Equation 3. The only complicated part of this process is determining the sum implicit in the function $\tilde{g}$ in

[4]Throughout this paper we use lg n as an abbreviation for $\log_2 n$ and $\lg^j n$ as an abbreviation for $(\lg n)^j$.

Equation 3, and this can usually be done with the aid of the following table in which we describe $\tilde{g}$ in relation to g.

Table 1.

$g(n)$		$\tilde{g}(n)$
$O(n^q)$	$q<0$	$\theta(1)$
$\lg^j n$	$j\geq 0$	$(\lg^{j+1} n)/(j+1) + \theta(\lg^j n)$
$\Omega(n^q)$	$q>0$	$\theta(g(n))$

Each of these entries can be verified easily by expanding the sum defining $\tilde{g}$. For the third entry we use Ω in the following restricted sense: we write $g(n) = \Omega(n^q)$ if there exists an n_0 and an $a>0$ such that $g(cn) \geq ac^q g(n)$ for all $c>1$ and all $n>n_0$.

4. Examples

In this section we will study a few common recurrences that have appeared in the literature and show that they can be solved by the method of Section 3. In these examples we will assume that T is defined only at powers of two and that some initial value T(1) is given.

Example 1. $T(n) = T(n/2) + 1$

This recurrence can be used to describe the worst-case cost of performing a binary search in an ordered table; see Knuth [1973, Section 6.2.1] for a description of that algorithm. The recurrence can be cast in the template of Equation 2 with $p = 0$ and $g(n) = 1$. By the second entry in Table 1 (setting $j = 0$) we have

$$\begin{aligned} T(n) &= 2^0 T(n/2) + n^0(\lg^0 n) \\ &= n^0[T(1) + \lg n + \theta(1)] \\ &= \lg n + \theta(1). \end{aligned}$$

Example 2. $T(n) = 2T(n/2) + n \lg n$

This recurrence arises in a multitude of applications. Kung [1973] describes an algorithm for computing the n normalized derivatives $P^{(i)}(t)/i!$ for $i = 1,\dots,n$, where P is an n-th degree polynomial; the running time of his algorithm is described by this recurrence. Given a set of n points in 3-space, the "All Nearest Neighbors" problem calls for finding the nearest neighbor for each point in the set among the rest of the points; Bentley and Shamos's [1976] algorithm for this problem has running time described by the recurrence. Kung, Luccio and Preparata [1975] describe the problem of computing all the *maximal* elements in a set of vectors. (Vector A *dominates* vector B if A is greater than B in all components; a vector is *maximal* in a set if it is not dominated by any other vector in the set.) They give algorithms for finding the maxima of k-dimensional vector sets; their algorithm for the case k=4 has running time described by this recurrence. This recurrence also describes the number of comparators needed to implement Batcher's nonadaptive odd-even merge sort (which is described by Liu [1977, pp. 200-203]).

To cast this recurrence in the template of Equation 2 we let $p = 1$ and $g(n) = \lg n$. We then use the second entry in Table 1 (with $j = 1$) which yields

$$\begin{aligned} T(n) &= 2^1 T(n/2) + n^1 \lg n \\ &= n^1[T(1) + (\lg^2 n)/2 + \theta(\lg n)] \\ &= (n \lg^2 n)/2 + \theta(n \lg n). \end{aligned}$$

Example 3. $T(n) = 7T(n/2) + \theta(n^2)$

This recurrence describes the running time of Strassen's [1969] matrix multiplication algorithm (the algorithm is also discussed by Aho, Hopcroft and Ullman [1974, pp. 230-232]). We cast this recurrence into the template of Equation 2 by letting $p = \lg 7$ and $g(n) = n^{2 - \lg 7}$. Since $2 - \lg 7 < 0$ we use the first entry in Table 1 to conclude that

$$\begin{aligned} T(n) &= 2^{\lg 7} T(n/2) + n^{\lg 7}\theta(n^{2 - \lg 7}) \\ &= n^{\lg 7}[T(1) + \theta(1)] \\ &= \theta(n^{\lg 7}). \end{aligned}$$

Example 4. $T(n) = T(n/2) + n \lg n$

This recurrence arises in a number of algorithms based on "extrapolative recursion". This term is described in more detail by Borodin and Munro [1975, p. 80]; many examples of such algorithms can be found later in their book. The same recurrence also arises in the analysis of the Ford-Johnson sorting algorithm (see Ford and Johnson [1959] or Knuth [1973]) which was the best-known sorting algorithm (in terms of using minimal comparisons) for almost twenty years--an algorithm faster for some values of n was recently obtained by Manacher [1977].

To cast this recurrence in the template of Equation 2 we let $p = 0$ and $g(n) = n \lg n$. Since $n \lg n = \Omega(n^1)$ we can use the third entry in Table 1 to deduce that

$$\begin{aligned} T(n) &= 2^0 T(n/2) + n^0(n \lg n) \\ &= n^0[T(1) + \theta(n \lg n)] \\ &= \theta(n \lg n). \end{aligned}$$

5. A More General Template

The template developed in Section 3 can easily be generalized to solve recurrences of the form

$$\begin{aligned} &T(1) \text{ given,} \\ &T(n) = kT(n/c) + f(n). \end{aligned}$$

(Note that this generalizes Equation 1 of Section 3 from division by two to division by any constant.) To solve the recurrence we cast it into the template

$$T(n) = c^pT(n/c) + n^pg(n),$$

where $p = \log_c k$ and $g(n) = f(n)/n^p$. The solution to this modified recurrence is

$$T(n) = n^p[T(1) + \sum_{1 \le i \le \log_c n} g(c^i)].$$

As in Section 3, the proof that this is the unique solution to the recurrence is by mathematical induction on the powers of c. For monotone functions g it can be proved that the above sum differs from $\tilde{g}(n)$ by at most a factor of lg c, so Table 1 can be used to solve the recurrence to within a constant factor.

This template can be used, for example, to analyze the running time of Pan's [1978] matrix multiplication algorithm, which satisfies the recurrence

$$T(n) = 143640\ T(n/70) + \theta(n^2).$$

This recurrence is rewritten into the template as

$$T(n) = 70^pT(n/70) + n^pg(n),$$

where p is defined as $\log_{70} 143640$, or approximately 2.79, and $g(n) = \theta(n^{2-p})$. The first entry of Table 1 tells us that $\tilde{g}(n) = \theta(1)$, so the solution of the recurrence is

$$\begin{aligned} T(n) &= n^p[T(1) + \theta(\tilde{g}(n))] \\ &= n^p[T(1) + \theta(1)] \\ &= \theta(n^p), \end{aligned}$$

or approximately $\theta(n^{2.79})$. This method can, of course, also be employed to analyze other matrix multiplication algorithms based on similar approaches, in which 143640 and 70 are replaced by other constants.

6. Extensions

The special template that we have seen in this note can be extended in several ways. Table 1 can be expanded to include many functions besides the three described in Section 3. For instance, if $g(n) = 1/(\lg n)$, then $\tilde{g}(n)$ is $H_{\lg n}$, the ($\lg n$)-th harmonic number, which is $O(\lg \lg n)$. Similarly, if $g(n) = 2^{(\lg^{1/2} n)}$, then $\tilde{g}(n)$ is $\theta(\lg^{1/2} n\ 2^{\lg^{1/2} n})$. Additionally, there exists a method for "interpolating" to find values of the ~ operator for functions not in the table: if $g_2(n)$ is between $g_1(n)$ and $g_3(n)$, then $\tilde{g}_2(n)/g_2(n)$ is between $\tilde{g}_3(n)/g_3(n)$ and $\tilde{g}_1(n)/g_1(n)$. (One must use "between" in a formal sense and assume "smoothness" properties of the various functions to show this result.) The linearity of the ~ operator (which follows from the linearity of $\sum$) can be used to find the second and lower order terms of the solution to the recurrence.

Templates can also be defined to solve other classes of recurrences of the form

$$T(n_0) \text{ given},$$
$$T(n) = a(n)\ T(b(n)) + f(n).$$

The critical observation in defining such a template is to rewrite the recurrence as

$$T(n) = a(n)\ T(b(n)) + h(n)\cdot g(n) \tag{4}$$

where the function h satisfies the property

$$h(n) = a(n)\, h(b(n)) .$$

In other words, h is a homogeneous solution of the original recurrence--it is precisely this property that facilitates cancellation as the recurrence is iterated. The solution of the original recurrence for T is then

$$T(n) = h(n)\cdot[g(n) + g(b(n)) + g(b(b(n))) + \dots + g(b^{-1}(n_0)) + T(n_0)/h(n_0)],$$

where b^{-1} represents the functional inverse of b.

As an example of this method, consider solving the recurrence

$$T(2) = 2,$$
$$T(n) = n^{1/2}\, T(n^{1/2}) + n \lg n,$$

which is defined only at integers of the form 2^{2^i}. Since a solution of the homogeneous equation $h(n) = n^{1/2}h(n^{1/2})$ is $h(n) = n$, we "rewrite" the recurrence as

$$T(n) = n^{1/2}T(n^{1/2}) + n \cdot (\lg n),$$

which has solution

$$T(n) = h(n) \cdot [\lg n + \lg n^{1/2} + \lg n^{1/4} + \dots + \lg 4 + T(2)/h(2)]$$

or

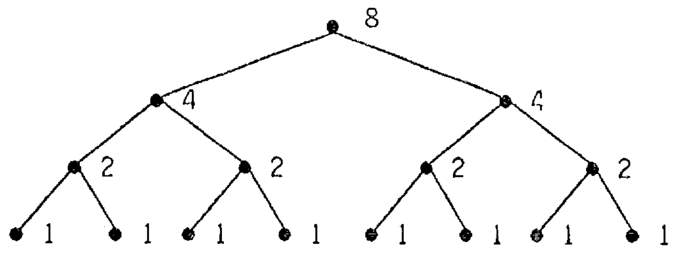
$$T(n) = n \cdot (2 \lg n - 1).$$

(The reader is encouraged to repeat this exercise with "lg n" replaced by "lg lg n" in the statement of the recurrence.)

This general method can be understood more intuitively if we recast Recurrence 4 as a *recursion tree*. A recursion tree is perhaps best defined by an example; the evaluation of the recurrence equation

$$T(1) = 1,$$
$$T(n) = 2T(n/2) + n$$

at the value n=8 is given by the recursion tree



Note that the sum of the values of all nodes in the tree gives 32, which is exactly T(8). In general, the recursion tree for the evaluation of Recurrence 4 at the value n is

- a single node with the value $T(n_0)$, if $n=n_0$;
- otherwise, a node with the value f(n) and a(n) sons, each of which is a recursion tree for the evaluation of the recurrence at b(n).

Note that the sum of the values of all nodes in the tree gives precisely T(n).

We can easily explain Recurrence 4 in the terminology of recursion trees if we define two of their properties. The first is the *depth* of the recursion tree, which we write as RD(n), for recursion depth (the depth of the above tree is three). The second is the number of leaves in the recursion tree, which is defined by the recurrence

$L(n_0) = 1,$
$L(n) = a(n)\ L(b(n)).$

(Notice that L(n) is just one of the homogeneous solutions of Recurrence 4.) We can now describe the value of T(n) by a case analysis of f(n)/L(n), which is the ratio of the work at the root to the number of leaves in the tree.

- If $f(n)/L(n) = O(q^{-RD(n)})$ for some $q>0$, then $T(n) = \theta(L(n))$. (That is, the work at the leaves dominates the cost.)
- If $f(n)/L(n) = \theta(RD(n)^j)$ for some nonnegative j, then $T(n) = \theta(f(n) \cdot RD(n))$. (That is, the work is approximately equal at all RD(n) levels of the tree.)
- If $f(n)/L(n) = \Omega(q^{RD(n)})$ for some $q>0$, then $T(n) = \theta(f(n))$. (That is, the work at the root dominates the cost.)

Notice that the above case analysis is isomorphic to Table 1 when a(n) is k and b(n) is n/2; as in that table, not all possible cases of f(n)/L(n) are enumerated above.

7. Conclusions

To conclude this paper we will briefly review its contents. The primary contribution is the *method* that we presented to solve the various recurrences--rewriting the recurrence by decomposing the additive term into some function times a solution of the homogeneous system. We described this method both in the very general terms of a recursion tree, and in the specific framework of divide-and-conquer recurrences. This method can be applied by memorizing a simple template (or the recursion tree formulation) and a table of three entries. While the novice can use this framework to solve (approximately) divide-and-conquer recurrences, the more experienced algorithm designer can use it in a more advanced fashion: to prune his search for an efficient algorithm. For example, if he were trying to find an $O(n \lg^3 n)$ algorithm by marrying together the solution to two problems of size n/2 each, his marriage step must require $O(n \lg^2 n)$ time.

Much work remains to be done in developing general methods for rapidly solving equations that describe the approximate complexity of algorithms; we mentioned some areas for further work in Section 6. Such methods will never be sufficient for the detailed "Knuthian" analysis of algorithms, but they can free algorithm designers from mundane analyses to let them work on more interesting problems.

Acknowledgements

The authors would like to thank Professor D. E. Knuth for (properly) chiding us about the "cook book" flavor of an earlier draft, and Professor V. Pratt for suggesting the recursion tree formulation of Section 6.

References

Aho, A. V., J. E. Hopcroft and J. D. Ullman [1974]. *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass.

Bentley, J. L. and M. I. Shamos [1976]. "Divide and conquer in multidimensional space", *Proceedings of the Eighth Symposium on the Theory of Computing*, ACM, May 1976, pp. 220-230.

Borodin, A. and I. Munro [1975]. *The Computational Complexity of Algebraic and Numeric Problems*, American Elsevier, New York, N.Y.

Ford, L. and S. Johnson [1959]. "A tournament problem", *American Mathematical Monthly* 66, pp. 391-395.

Keller, K. [1980]. "Computation cost functions for divide-and-conquer algorithms", to appear in *Journal of Undergraduate Mathematics*.

Knuth, D. E. [1973]. *The Art of Computer Programming, volume 3: Sorting and Searching*, Addison-Wesley, Reading, Mass.

Kung, H. T. [1973]. *A New Upper Bound on the Complexity of Derivative Evaluation*, Carnegie-Mellon University Computer Science Report, September 1973.

Kung, H. T., F. Luccio and F. P. Preparata [1975]. "On finding the maxima of a set of vectors", *JACM 22*, 4, October 1975, pp. 469-476.

Liu, C. L. [1977]. *Elements of Discrete Mathematics*, McGraw-Hill, New York, N. Y.

Manacher, G. K. [1977]. "The Ford-Johnson sorting algorithm is not optimal", *Proceedings of the Fifteenth Annual Allerton Conference on Communications, Control and Computing*, Sepember 1977, pp. 390-397.

Pan, V. Ya. [1978]. "An introduction to the trilinear technique of aggregating, uniting and canceling and applications of the technique for constructing fast algorithms for matrix operations", *Proceedings of the Nineteenth Annual Symposium on the Foundations of Computer Science*, IEEE, 1978.

Stanat, D. F. and D. F. McAllister [1977]. *Discrete Mathematics in Computer Science*, Prentice-Hall, Englewood Cliffs, N. J.

Strassen, V. [1969]. "Gaussian elimination is not optimal", *Numerische Mathematik 13*, pp. 354-356.