The Design and Analysis of Algorithms

Dexter C. Kozen

With 72 Illustrations

Springer-Verlag
New York Berlin Heidelberg London Paris
Tokyo Hong Kong Barcelona Budapest

Lecture 1 Algorithms and Their Complexity

This is a course on the design and analysis of algorithms intended for first-year graduate students in computer science. Its purposes are mixed: on the one hand, we wish to cover some fairly advanced topics in order to provide a glimpse of current research for the benefit of those who might wish to specialize in this area; on the other, we wish to introduce some core results and techniques which will undoubtedly prove useful to those planning to specialize in other areas.

We will assume that the student is familiar with the classical material normally taught in upper-level undergraduate courses in the design and analysis of algorithms. In particular, we will assume familiarity with:

- sequential machine models, including Turing machines and random access machines (RAMs)
- discrete mathematical structures, including graphs, trees, and dags, and their common representations (adjacency lists and matrices)
- fundamental data structures, including lists, stacks, queues, arrays, balanced trees
- fundamentals of asymptotic analysis, including $O(\cdot)$, $o(\cdot)$, and $\Omega(\cdot)$ notation, and techniques for the solution of recurrences
- fundamental programming techniques, such as recursion, divide-and-conquer, dynamic programming
- basic sorting and searching algorithms.

These notions are covered in the early chapters of [3, 39, 100].

Familiarity with elementary algebra, number theory, and discrete probability theory will be helpful. In particular, we will be making occasional use of the following concepts: linear independence, basis, determinant, eigenvalue, polynomial, prime, modulus, Euclidean algorithm, greatest common divisor, group, ring, field, random variable, expectation, conditional probability, conditional expectation. Some excellent classical references are [69, 49, 33].

The main emphasis will be on *asymptotic worst-case complexity.* This measures how the worst-case time or space complexity of a problem grows with the size of the input. We will also spend some time on probabilistic algorithms and analysis.

1.1 Asymptotic Complexity

Let f and g be functions $\mathcal{N} \to \mathcal{N}$, where $\mathcal{N}$ denotes the natural numbers $\{0, 1, \ldots\}$. Formally,

- f is $O(g)$ if

$$\exists c \in \mathcal{N} \ \overset{\infty}{\forall} n \ f(n) \leq c \cdot g(n) \ .$$

  The notation $\overset{\infty}{\forall}$ means "for almost all" or "for all but finitely many". Intuitively, f grows no faster asymptotically than g to within a constant multiple.

- f is $o(g)$ if

$$\forall c \in \mathcal{N} \ \overset{\infty}{\forall} n \ f(n) \leq \frac{1}{c} \cdot g(n) \ .$$

  This is a stronger statement. Intuitively, f grows strictly more slowly than any arbitrarily small positive constant multiple of g. For example, n^{347} is $o(2^{(\log n)^2})$.

- f is $\Omega(g)$ if g is $O(f)$. In other words, f is $\Omega(g)$ if

$$\exists c \in \mathcal{N} \ \overset{\infty}{\forall} n \ f(n) \geq \frac{1}{c} \cdot g(n) \ .$$

- f is $\Theta(g)$ if f is both $O(g)$ and $\Omega(g)$.

There is one cardinal rule:

Always use O and o for upper bounds and Ω for lower bounds. *Never* use O for lower bounds.

There is some disagreement about the definition of Ω. Some authors (such as [43]) prefer the definition as given above. Others (such as [108]) prefer: f is $\Omega(g)$ if g is not $o(f)$; in other words, f is $\Omega(g)$ if

$$\exists c \in \mathcal{N} \ \overset{\infty}{\exists} n \ f(n) > \frac{1}{c} \cdot g(n) \ .$$

(The notation $\overset{\infty}{\exists}$ means "there exist infinitely many".) The latter is weaker and presumably easier to establish, but the former gives sharper results. We won't get into the fray here, but just comment that neither definition precludes algorithms from taking less than the stated bound on certain inputs. For example, the assertion, "The running time of mergesort is $\Omega(n \log n)$" says that there is a c such that for all but finitely many n, there is some input sequence of length n on which mergesort makes at least $\frac{1}{c} n \log n$ comparisons. There is nothing to prevent mergesort from taking less time on some other input of length n.

The exact interpretation of statements involving O, o, and Ω depends on assumptions about the underlying model of computation, how the input is presented, how the size of the input is determined, and what constitutes a single step of the computation. In practice, authors often do not bother to write these down. For example, "The running time of mergesort is $O(n \log n)$" means that there is a fixed constant c such that for any n elements drawn from a totally ordered set, at most $cn \log n$ comparisons are needed to produce a sorted array. Here nothing is counted in the running time except the number of comparisons between individual elements, and each comparison is assumed to take one step; other operations are ignored. Similarly, nothing is counted in the input size except the number of elements; the size of each element (whatever that may mean) is ignored.

It is important to be aware of these unstated assumptions and understand how to make them explicit and formal when reading papers in the field. When making such statements yourself, always have your underlying assumptions in mind. Although many authors don't bother, it is a good habit to state any assumptions about the model of computation explicitly in any papers you write.

The question of what assumptions are reasonable is more often than not a matter of esthetics. You will become familiar with the standard models and assumptions from reading the literature; beyond that, you must depend on your own conscience.

1.2 Models of Computation

Our principal model of computation will be the unit-cost random access machine (RAM). Other models, such as uniform circuits and PRAMs, will be introduced when needed. The RAM model allows random access and the use

of arrays, as well as unit-cost arithmetic and bit-vector operations on arbitrarily large integers; see [3].

For graph algorithms, arithmetic is often unnecessary. Of the two main representations of graphs, namely *adjacency matrices* and *adjacency lists*, the former requires random access and $\Omega(n^2)$ array storage; the latter, only linear storage and no random access. (For graphs, *linear* means $O(n+m)$, where n is the number of vertices of the graph and m is the number of edges.) The most esthetically pure graph algorithms are those that use the adjacency list representation and only manipulate pointers. To express such algorithms one can formulate a very weak model of computation with primitive operators equivalent to **car**, **cdr**, **cons**, **eq**, and **nil** of pure LISP; see also [99].

1.3 A Grain of Salt

No mathematical model can reflect reality with perfect accuracy. Mathematical models are abstractions; as such, they are necessarily flawed.

For example, it is well known that it is possible to abuse the power of unit-cost RAMs by encoding horrendously complicated computations in large integers and solving intractible problems in polynomial time [50]. However, this violates the unwritten rules of good taste. One possible preventative measure is to use the log-cost model; but when used as intended, the unit-cost model reflects experimental observation more accurately for data of moderate size (since multiplication really does take one unit of time), besides making the mathematical analysis a lot simpler.

Some theoreticians consider asymptotically optimal results as a kind of Holy Grail, and pursue them with a relentless frenzy (present company not necessarily excluded). This often leads to contrived and arcane solutions that may be superior by the measure of asymptotic complexity, but whose constants are so large or whose implementation would be so cumbersome that no improvement in technology would ever make them feasible. What is the value of such results? Sometimes they give rise to new data structures or new techniques of analysis that are useful over a range of problems, but more often than not they are of strictly mathematical interest. Some practitioners take this activity as an indictment of asymptotic complexity itself and refuse to admit that asymptotics have anything at all to say of interest in practical software engineering.

Nowhere is the argument more vociferous than in the theory of parallel computation. There are those who argue that many of the models of computation in common use, such as uniform circuits and PRAMs, are so inaccurate as to render theoretical results useless. We will return to this controversy later on when we talk about parallel machine models.

Such extreme attitudes on either side are unfortunate and counterproductive. By now asymptotic complexity occupies an unshakable position in our computer science consciousness, and has probably done more to guide us in

improving technology in the design and analysis of algorithms than any other mathematical abstraction. On the other hand, one should be aware of its limitations and realize that an asymptotically optimal solution is not necessarily the best one.

A good rule of thumb in the design and analysis of algorithms, as in life, is to use common sense, exercise good taste, and always listen to your conscience.

1.4 Strassen's Matrix Multiplication Algorithm

Probably the single most important technique in the design of asymptotically fast algorithms is *divide-and-conquer*. Just to refresh our understanding of this technique and the use of recurrences in the analysis of algorithms, let's take a look at Strassen's classical algorithm for matrix multiplication and some of its progeny. Some of these examples will also illustrate the questionable lengths to which asymptotic analysis can sometimes be taken.

The usual method of matrix multiplication takes 8 multiplications and 4 additions to multiply two 2×2 matrices, or in general $O(n^3)$ arithmetic operations to multiply two $n \times n$ matrices. However, the number of multiplications can be reduced. Strassen [97] published one such algorithm for multiplying 2×2 matrices using only 7 multiplications and 18 additions:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} s_1 + s_2 - s_4 + s_6 & s_4 + s_5 \\ s_6 + s_7 & s_2 - s_3 + s_5 - s_7 \end{bmatrix}$$

where

$$\begin{aligned} s_1 &= (b-d)\cdot(g+h) \\ s_2 &= (a+d)\cdot(e+h) \\ s_3 &= (a-c)\cdot(e+f) \\ s_4 &= (a+b)\cdot h \\ s_5 &= a\cdot(f-h) \\ s_6 &= d\cdot(g-e) \\ s_7 &= (c+d)\cdot e \end{aligned}$$

Assume for simplicity that n is a power of 2. (This is not the last time you will hear that.) Apply the 2×2 algorithm recursively on a pair of $n \times n$ matrices by breaking each of them up into four square submatrices of size $\frac{n}{2} \times \frac{n}{2}$:

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \cdot \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} S_1 + S_2 - S_4 + S_6 & S_4 + S_5 \\ S_6 + S_7 & S_2 - S_3 + S_5 - S_7 \end{bmatrix}$$

where

$$S_1 = (B-D)\cdot(G+H)$$

$$\begin{aligned} S_2 &= (A+D)\cdot(E+H) \\ S_3 &= (A-C)\cdot(E+F) \\ S_4 &= [illegible]\cdot(A+B)\cdot H \\ S_5 &= A\cdot(F-H) \\ S_6 &= D\cdot(G-E) \\ S_7 &= [illegible](C+D)\cdot E \end{aligned}$$

Everything is the same as in the 2×2 case, except now we are manipulating $\frac{n}{2} \times \frac{n}{2}$ matrices instead of scalars. (We have to be slightly cautious, since matrix multiplication is not commutative.) Ultimately, how many scalar operations $(+, -, \cdot)$ does this recursive algorithm perform in multiplying two $n \times n$ matrices? We get the recurrence

$$T(n) = 7T(\frac{n}{2}) + dn^2$$

with solution

$$\begin{aligned} T(n) &= (1 + \frac{4}{3}d)n^{\log_2 7} + O(n^2) \\ &= O(n^{\log_2 7}) \\ &= O(n^{2.81\ldots}) \end{aligned}$$

which is $o(n^3)$. Here d is a fixed constant, and dn^2 represents the time for the matrix additions and subtractions.

This is already a significant asymptotic improvement over the naive algorithm, but can we do even better? In general, an algorithm that uses c multiplications to multiply two $d \times d$ matrices, used as the basis of such a recursive algorithm, will yield an $O(n^{\log_d c})$ algorithm. To beat Strassen's algorithm, we must have $c < d^{\log_2 7}$. For a 3×3 matrix, we need $c < 3^{\log_2 7} = 21.8\ldots$, but the best known algorithm uses 23 multiplications.

In 1978, Victor Pan [83, 84] showed how to multiply 70×70 matrices using 143640 multiplications. This gives an algorithm of approximately $O(n^{2.795\ldots})$. The asymptotically best algorithm known to date, which is achieved by entirely different methods, is $O(n^{2.376\ldots})$ [25]. Every algorithm must be $\Omega(n^2)$, since it has to look at all the entries of the matrices; no better lower bound is known.