13

Numerical Problems

If most problems you encounter are numerical in nature, there is an excellent chance that you are reading the wrong book. *Numerical Recipes* [PFTV07] gives a terrific overview to the fundamental problems in numerical computing, including linear algebra, numerical integration, statistics, and differential equations. Different flavors of the book include source code for all the algorithms in C++, Fortran, and even Pascal. Their coverage is somewhat skimpier on the combinatorial/numerical problems we consider in this section, but you should be aware of this book. Check it out at *http://www.nr.com*.

Numerical algorithms tend to be different beasts than combinatorial algorithms for at least two reasons:

- *Issues of Precision and Error* – Numerical algorithms typically perform repeated floating-point computations, which accumulate error at each operation until, eventually, the results are meaningless. My favorite example [MV99] concerns the Vancouver Stock Exchange, which over a twenty-two month period accumulated enough round-off error to reduce its index to 574.081 from the correct value of 1098.982.

  A simple and dependable way to test for round-off errors in numerical programs is to run them both at single and double precision, and then think hard whenever there is a disagreement.

- *Extensive Libraries of Codes* – Large, high-quality libraries of numerical routines have existed since the 1960s, which is still not yet the case for combinatorial algorithms. There are several reasons for this, including (1) the early emergence of Fortran as a standard for numerical computation, (2) the nature of numerical computations to be recognizably independent rather than

S.S. Skiena, *The Algorithm Design Manual*, 2nd ed., DOI: 10.1007/978-1-84800-070-4_13,

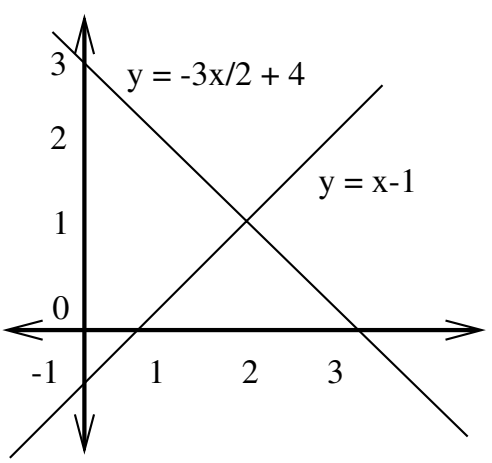
embedded within large applications, and (3) the existence of large scientific communities needing general numerical libraries.

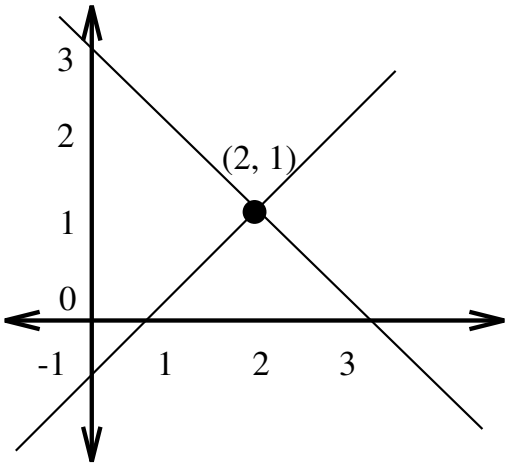
Regardless of why, you should exploit this software base. There is probably no reason to implement algorithms for any of the problems in this section as opposed to using existing codes. Searching Netlib (see Section 19.1.5) is an excellent place to start.

Many scientist's and engineer's ideas about algorithms culturally derive from Fortran programming and numerical methods. Computer scientists grow up programming with pointers and recursion, and so are comfortable with the more sophisticated data structures required for combinatorial algorithms. Both sides can and should learn from each other, since problems such as pattern recognition can be modeled either numerically or combinatorially.

There is a vast literature on numerical algorithms. In addition to *Numerical Recipes*, recommended books include:

- *Chapara and Canale* [CC05] – The contemporary market leader in numerical analysis texts.
- *Mak* [Mak02] – This enjoyable text introduces Java to the world of numerical computation, and visa versa. Source code is provided.
- *Hamming* [Ham87] – This oldie but goodie provides a clear and lucid treatment of fundamental methods in numerical computation. It is available in a low-priced Dover edition.
- *Skeel and Keiper* [SK00] – A readable and interesting treatment of basic numerical methods, avoiding overly detailed algorithm descriptions through its use of the computer algebra system Mathematica. I like it.
- *Cheney and Kincaid* [CK07] – A traditional Fortran-based numerical analysis text, with discussions of optimization and Monte Carlo methods in addition to such standard topics as root-finding, numerical integration, linear systems, splines, and differential equations.
- *Buchanan and Turner* [BT92] – Thorough language-independent treatment of all standard topics, including parallel algorithms. It is the most comprehensive of the texts described here.





INPUT OUTPUT

13.1 Solving Linear Equations

Input description: An $m \times n$ matrix A and an $m \times 1$ vector b, together representing m linear equations on n variables.

Problem description: What is the vector x such that $A \cdot x = b$?

Discussion: The need to solve linear systems arises in an estimated 75% of all scientific computing problems [DB74]. For example, applying Kirchhoff's laws to analyze electrical circuits generates a system of equations—the solution of which puts currents through each branch of the circuit. Analysis of the forces acting on a mechanical truss generates a similar set of equations. Even finding the point of intersection between two or more lines reduces to solving a small linear system.

Not all systems of equations have solutions. Consider the equations $2x + 3y = 5$ and $2x + 3y = 6$. Some systems of equations have multiple solutions, such as $2x + 3y = 5$ and $4x + 6y = 10$. Such *degenerate* systems of equations are called *singular*, and they can be recognized by testing whether the determinant of the coefficient matrix is zero.

Solving linear systems is a problem of such scientific and commercial importance that excellent codes are readily available. There is no good reason to implement your own solver, even though the basic algorithm (Gaussian elimination) is one you learned in high school. This is especially true when working with large systems.

Gaussian elimination is based on the observation that the solution to a system of linear equations is invariant under scaling (if $x = y$, then $2x = 2y$) and adding

equations (the solution to $x = y$ and $w = z$ is the same as the solution to $x = y$ and $x + w = y + z$). Gaussian elimination scales and adds equations to eliminate each variable from all but one equation, leaving the system in such a state that the solution can be read off from the equations.

The time complexity of Gaussian elimination on an $n \times n$ system of equations is $O(n^3)$, since to clear the ith variable we add a scaled copy of the n-term ith row to each of the $n-1$ other equations. On this problem, however, constants matter. Algorithms that only partially reduce the coefficient matrix and then backsubstitute to get the answer use 50% fewer floating-point operations than the naive algorithm.

Issues to worry about include:

- *Are roundoff errors and numerical stability affecting my solution?* – Gaussian elimination would be quite straightforward to implement except for roundoff errors. These accumulate with each row operation and can quickly wreak havoc on the solution, particularly with matrices that are *almost* singular.

  To eliminate the danger of numerical errors, it pays to substitute the solution back into each of the original equations and test how close they are to the desired value. *Iterative methods* for solving linear systems refine initial solutions to obtain more accurate answers. Good linear systems packages will include such routines.

  The key to minimizing roundoff errors in Gaussian elimination is selecting the right equations and variables to pivot on, and to scale the equations to eliminate large coefficients. This is an art as much as a science, which is why you should use a well-crafted library routine as described next.

- *Which routine in the library should I use?* – Selecting the right code is also somewhat of an art. If you are taking your advice from this book, start with the general linear system solvers. Hopefully they will suffice for your needs. But search through the manual for more efficient procedures solving special types of linear systems. If your matrix happens to be one of these special types, the solution time can reduce from cubic to quadratic or even linear.

- *Is my system sparse?* – The key to recognizing that you have a special-case linear system is establishing how many matrix elements you really need to describe A. If there are only a few non-zero elements, your matrix is *sparse* and you are in luck. If these few non-zero elements are clustered near the diagonal, your matrix is *banded* and you are in even more luck. Algorithms for reducing the bandwidth of a matrix are discussed in Section 13.2. Many other regular patterns of sparse matrices can also be exploited, so consult the manual of your solver or a better book on numerical analysis for details.

- *Will I be solving many systems using the same coefficient matrix?* – In applications such as least-squares curve fitting, we must solve $A \cdot x = b$ repeatedly with different b vectors. We can preprocess A to make this easier. The lower-upper or *LU-decomposition* of A creates lower- and upper-triangular matrices

L and U such that $L \cdot U = A$. We can use this decomposition to solve $A \cdot x = b$, since

$$A \cdot x = (L \cdot U) \cdot x = L \cdot (U \cdot x) = b$$

This is efficient since backsubstitution solves a triangular system of equations in quadratic time. Solving $L \cdot y = b$ and then $U \cdot x = y$ gives the solution x using two $O(n^2)$ steps instead of one $O(n^3)$ step, after the LU-decomposition has been found in $O(n^3)$ time.

The problem of solving linear systems is equivalent to that of matrix inversion, since $Ax = B \leftrightarrow A^{-1}Ax = A^{-1}B$, where $I = A^{-1}A$ is the identity matrix. Avoid it however since matrix inversion proves to be three times slower than Gaussian elimination. LU-decomposition proves useful in inverting matrices as well as computing determinants (see Section 13.4 (page 404)).

Implementations: The library of choice for solving linear systems is apparently LAPACK—a descendant of LINPACK [DMBS79]. Both of these Fortran codes, as well as many others, are available from Netlib. See Section 19.1.5 (page 659).

Variants of LAPACK exist for other languages, like CLAPACK (C) and LAPACK++ (C++). The *Template Numerical Toolkit* is an interface to such routines in C++, and is available at *http://math.nist.gov/tnt/*.

JScience provides an extensive linear algebra package (including determinants) as part of its comprehensive scientific computing library. *JAMA* is another matrix package written in Java. Links to both and many related libraries are available at *http://math.nist.gov/javanumerics/*.

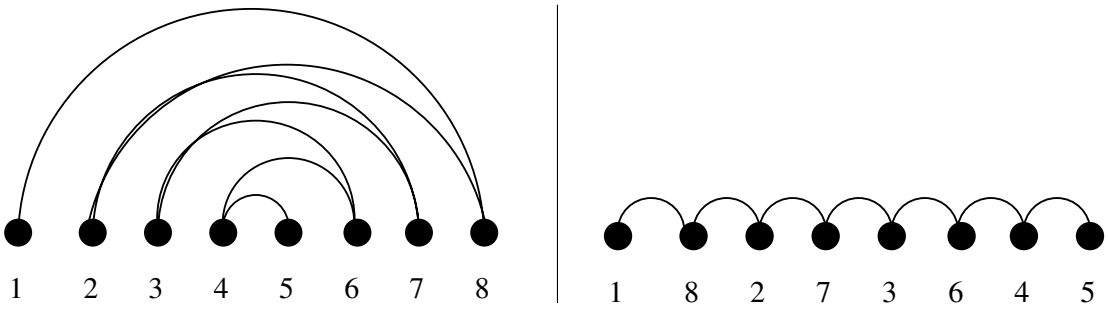
Numerical Recipes [PFTV07] (*www.nr.com*) provides guidance and routines for solving linear systems. Lack of confidence in dealing with numerical procedures is the most compelling reason to use these ahead of the free codes.

Notes: Golub and van Loan [GL96] is the standard reference on algorithms for linear systems. Good expositions on algorithms for Gaussian elimination and LU-decomposition include [CLRS01] and a host of numerical analysis texts [BT92, CK07, SK00]. Data structures for linear systems are surveyed in [PT05].

Parallel algorithms for linear systems are discussed in [Gal90, KSV97, Ort88]. Solving linear systems is one of most important applications where parallel architectures are used widely in practice.

Matrix inversion and (hence) linear systems solving can be done in matrix multiplication time using Strassen's algorithm plus a reduction. Good expositions on the equivalence of these problems include [AHU74, CLRS01].

Related Problems: Matrix multiplication (see page 401), determinant/permanent (see page 404).



13.2 Bandwidth Reduction

Input description: A graph $G = (V, E)$, representing an $n \times n$ matrix M of zero and non-zero elements.

Problem description: Which permutation p of the vertices minimizes the length of the longest edge when the vertices are ordered on a line—i.e. , minimizes $\max_{(i,j)\in E} |p(i) - p(j)|$?

Discussion: Bandwidth reduction lurks as a hidden but important problem for both graphs and matrices. Applied to matrices, bandwidth reduction permutes the rows and columns of a sparse matrix to minimize the distance b of any non-zero entry from the center diagonal. This is important in solving linear systems, because Gaussian elimination (see Section 13.1 (page 395)) can be performed in $O(nb^2)$ on matrices of bandwidth b. This is a big win over the general $O(n^3)$ algorithm if $b << n$.

Bandwidth minimization on graphs arises in more subtle ways. Arranging n circuit components in a line to minimize the length of the longest wire (and hence time delay) is a bandwidth problem, where each vertex of our graph corresponds to a circuit component and there is an edge for every wire linking two components. Alternatively, consider a hypertext application where we must store large objects (say images) on a magnetic tape. Each image has a set of possible images to visit next (i.e. , the hyperlinks). We seek to place linked images near each other on the tape to minimize the search time. This is exactly the bandwidth problem. More general formulations, such as rectangular circuit layouts and magnetic disks, inherit the same hardness and classes of heuristics from the linear versions.

The *bandwidth* problem seeks a linear ordering of the vertices, which minimizes the length of the longest edge, but there are several variations of the problem. In *linear arrangement*, we seek to minimize the sum of the lengths of the edges. This

has application to circuit layout, where we seek to position the chips to minimize the total wire length. In *profile minimization*, we seek to minimize the sum of one-way distances (i.e., for each vertex v) the length of the longest edge whose other vertex is to the left of of v.

Unfortunately, bandwidth minimization and all these variants is NP-complete. It stays NP-complete even if the input graph is a tree whose maximum vertex degree is 3, which is about as strong a condition as I have seen on any problem. Thus our only options are a brute-force search and heuristics.

Fortunately, ad hoc heuristics have been well studied and production-quality implementations of the best heuristics are available. These are based on performing a breadth-first search from a given vertex v, where v is placed at the leftmost point of the ordering. All of the vertices that are distance 1 from v are placed to its immediate right, followed by all the vertices at distance 2, and so forth until all vertices in G are accounted for. The popular heuristics differ according to how many different start vertices are considered and how equidistant vertices are ordered among themselves. Breaking ties with low-degree vertices over to the left however seems to be a good idea.

Implementations of the most popular heuristics—the Cuthill-McKee and Gibbs-Poole-Stockmeyer algorithms—are discussed in the implementation section. The worst case of the Gibbs-Poole-Stockmeyer algorithm is $O(n^3)$, which would wash out any possible savings in solving linear systems, but its performance in practice is close to linear.

Brute-force search programs can find the exact minimum bandwidth by backtracking through the set of $n!$ possible permutations of vertices. Considerable pruning can be achieved to reduce the search space by starting with a good heuristic bandwidth solution and alternately adding vertices to the left- and rightmost open slots in the partial permutation.

Implementations: Del Corso and Manzini's [CM99] code for exact solutions to bandwidth problems is available at *http://www.mfn.unipmn.it/∼manzini/bandmin*. Caprara and Salazar-González [CSG05] developed improved methods based on integer programming. Their branch-and-bound implementation in C is available at *http://joc.pubs.informs.org/Supplements/Caprara-2/*.

Fortran language implementations of both the Cuthill-McKee algorithm [CGPS76, Gib76, CM69] and the Gibbs-Poole-Stockmeyer algorithm [Lew82, GPS76] are available from Netlib. See Section 19.1.5 (page 659). Empirical evaluations of these and other algorithms on a test suite of 30 matrices are discussed in [Eve79b], showing Gibbs-Poole-Stockmeyer to be the consistent winner.

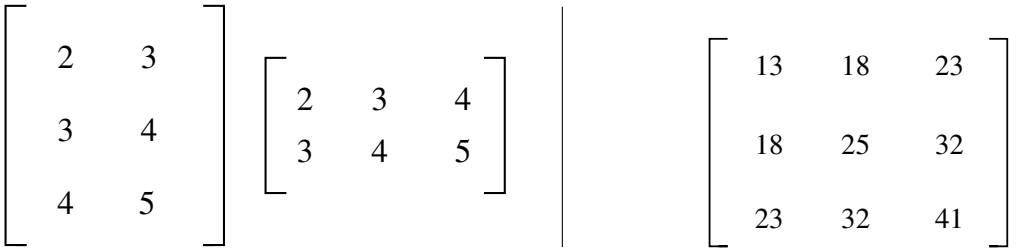
Petit [Pet03] performed an extensive experimental study on heuristics for the minimum linear arrangement problem. His codes and data are available at *http://www.lsi.upc.edu/∼jpetit/MinLA/Experiments/*.

Notes: Diaz et al. [DPS02] provide an excellent up-to-date survey on algorithms for bandwidth and related graph layout problems. See [CCDG82] for graph-theoretic and algorithmic results on bandwidth up to 1981.

Ad hoc heuristics have been widely studied—a tribute to its importance in numerical computation. Everstine [Eve79b] cites no less than 49 different bandwidth reduction algorithms! Del Corso and Romani [CR01] investigate a new class of spectral heuristics for bandwidth minimization.

The hardness of the bandwidth problem was first established by Papadimitriou [Pap76b], and its hardness on trees of maximum degree 3 in [GGJK78]. There are algorithms that run in polynomial time for fixed bandwidth k [Sax80]. Approximation algorithms offering a polylogarithmic guarantee exist for the general problem [BKRV00].

Related Problems: Solving linear equations (see page 395), topological sorting (see page 481).



13.3 Matrix Multiplication

Input description: An $x \times y$ matrix A and a $y \times z$ matrix B.

Problem description: Compute the $x \times z$ matrix $A \times B$.

Discussion: Matrix multiplication is a fundamental problem in linear algebra. Its main significance for combinatorial algorithms is its equivalence to many other problems, including transitive closure/reduction, parsing, solving linear systems, and matrix inversion. Thus, a faster algorithm for matrix multiplication implies faster algorithms for all of these problems. Matrix multiplication arises in its own right in computing the results of such coordinate transformations as scaling, rotation, and translation for robotics and computer graphics.

The following tight algorithm computes the product of $x \times y$ matrix A and $y \times z$ matrix B runs in $O(xyz)$. Remember to first initialize $M[i, j]$ to 0 for all $1 \leq i \leq x$ and $i \leq j \leq z$:

```
for i = 1 to x do
    for j = 1 to z
        for k = 1 to y
            M[i,j] = M[i,j] + A[i,k] · A[k,j]
```

An implementation in C appears in Section 2.5.4 (page 45). This straightforward algorithm would *seem* to be tough to beat in practice. That said, observe that the three loops can be arbitrarily permuted without changing the resulting answer. Such a permutation will change the memory access patterns and thus how effectively the cache memory is used. One can expect a 10-20% variation in run time among the six possible implementations, but could not confidently predict the winner (typically ikj) without running it on your machine with your given matrices.

When multiplying bandwidth-b matrices, where all non-zero elements of A and B lie within b elements of the main diagonals, a speedup to $O(xbz)$ is possible, since zero elements cannot contribute to the product.

Asymptotically faster algorithms for matrix multiplication exist, using clever divide-and-conquer recurrences. However, these prove difficult to program, require very large matrices to beat the trivial algorithm, and are less numerically stable to boot. The most famous of these is Strassen's $O(n^{2.81})$ algorithm. Empirical results (discussed next) disagree on the exact crossover point where Strassen's algorithm beats the simple cubic algorithm, but it is in the ballpark of $n \approx 100$.

There is a better way to save computation when you are multiplying a chain of more than two matrices together. Recall that multiplying an $x \times y$ matrix by a $y \times z$ matrix creates an $x \times z$ matrix. Thus, multiplying a chain of matrices from left to right might create large intermediate matrices, each taking a lot of time to compute. Matrix multiplication is not commutative, but it is associative, so we can parenthesize the chain in whatever manner we deem best without changing the final product. A standard dynamic programming algorithm can be used to construct the optimal parenthesization. Whether it pays to do this optimization will depend upon whether your matrix dimensions are sufficiently irregular and your chain multiplied often enough to justify it. Note that we are optimizing over the sizes of the dimensions in the chain, not the actual matrices themselves. No such optimization is possible if all your matrices are the same dimensions.

Matrix multiplication has a particularly interesting interpretation in counting the number of paths between two vertices in a graph. Let A be the adjacency matrix of a graph G, meaning $A[i,j] = 1$ if there is an edge between i and j. Otherwise, $A[i,j] = 0$. Now consider the square of this matrix, $A^2 = A \times A$. If $A^2[i,j] \geq 1$. This means that there must be a vertex k such that $A[i,k] = A[k,j] = 1$, so i to k to j is a path of length 2 in G. More generally, $A^k[i,j]$ counts the number of paths of length exactly k between i and j. This count includes nonsimple paths, where vertices are repeated, such as i to k to i to j.

Implementations: D'Alberto and Nicolau [DN07] have engineered a very efficient matrix multiplication code, which switches from Strassen's to the cubic algorithm at the optimal point. It is available at *http://www.ics.uci.edu/~fastmm/*. Earlier experiments put the crossover point where Strassen's algorithm beats the cubic algorithm at about $n = 128$ [BLS91, CR76].

Thus, an $O(n^3)$ algorithm will likely be your best bet unless your matrices are very large. The linear algebra library of choice is LAPACK, a descendant of LINPACK [DMBS79], which includes several routines for matrix multiplication. These Fortran codes are available from Netlib as discussed in Section 19.1.5 (page 659).

Algorithm 601 [McN83] of the Collected Algorithms of the ACM is a sparse matrix package written in Fortran that includes routines to multiply any combination of sparse and dense matrices. See Section 19.1.5 (page 659) for details.

Notes: Winograd's algorithm for fast matrix multiplication reduces the number of multiplications by a factor of two over the straightforward algorithm. It is implementable, although the additional bookkeeping required makes it doubtful whether it is a win. Expositions on Winograd's algorithm [Win68] include [CLRS01, Man89, Win80].

In my opinion, the history of theoretical algorithm design began when Strassen [Str69] published his $O(n^{2.81})$-time matrix multiplication algorithm. For the first time, improving an algorithm in the asymptotic sense became a respected goal in its own right. Progressive improvements to Strassen's algorithm have gotten progressively less practical. The current best result for matrix multiplication is Coppersmith and Winograd's [CW90] $O(n^{2.376})$ algorithm, while the conjecture is that $\Theta(n^2)$ suffices. See [CKSU05] for an alternate approach that recently yielded an $O(n^{2.41})$ algorithm.

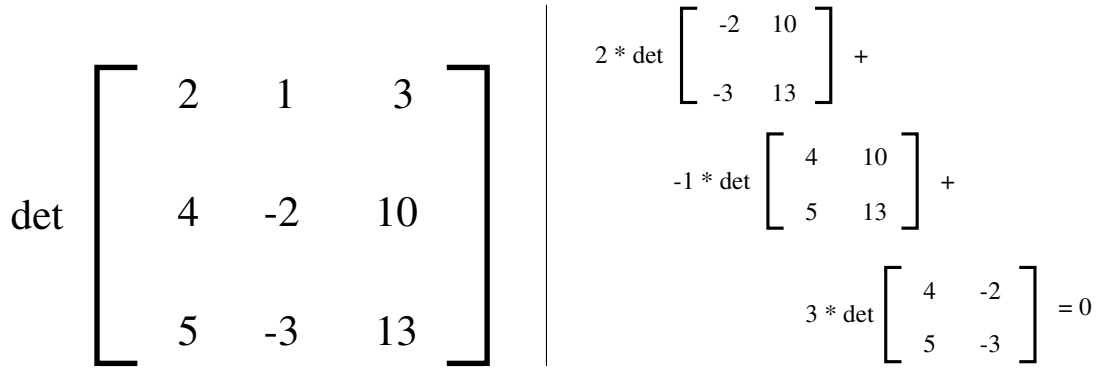
Engineering efficient matrix multiplication algorithms requires careful management of cache memory. See [BDN01, HUW02] for studies on these issues.

The interest in the squares of graphs goes beyond counting paths. Fleischner [Fle74] proved that the square of any biconnected graph has a Hamiltonian cycle. See [LS95] for results on finding the square roots of graphs—i.e. , finding A given A^2.

The problem of Boolean matrix multiplication can be reduced to that of general matrix multiplication [CLRS01]. The four-Russians algorithm for Boolean matrix multiplication [ADKF70] uses preprocessing to construct all subsets of $\lg n$ rows for fast retrieval in performing the actual multiplication, yielding a complexity of $O(n^3/\lg n)$. Additional preprocessing can improve this to $O(n^3/\lg^2 n)$ [Ryt85]. An exposition on the four-Russians algorithm, including this speedup, appears in [Man89].

Good expositions of the matrix-chain algorithm include [BvG99, CLRS01], where it is given as a standard textbook example of dynamic programming.

Related Problems: Solving linear equations (see page 395), shortest path (see page 489).



INPUT OUTPUT

13.4 Determinants and Permanents

Input description: An $n \times n$ matrix M.

Problem description: What is the determinant $|M|$ or permanent $perm(M)$ of the matrix m?

Discussion: Determinants of matrices provide a clean and useful abstraction in linear algebra that can be used to solve a variety of problems:

- Testing whether a matrix is *singular*, meaning that the matrix does not have an inverse. A matrix M is singular iff $|M| = 0$.
- Testing whether a set of d points lies on a plane in fewer than d dimensions. If so, the system of equations they define is singular, so $|M| = 0$.
- Testing whether a point lies to the left or right of a line or plane. This problem reduces to testing whether the sign of a determinant is positive or negative, as discussed in Section 17.1 (page 564).
- Computing the area or volume of a triangle, tetrahedron, or other simplicial complex. These quantities are a function of the magnitude of the determinant, as discussed in Section 17.1 (page 564).

The determinant of a matrix M is defined as a sum over all $n!$ possible permutations π_i of the n columns of M:

$$|M| = \sum_{i=1}^{n!} (-1)^{sign(\pi_i)} \prod_{j=1}^{n} M[j, \pi_j]$$

where $sign(\pi_i)$ denotes the number of pairs of elements out of order (called *inversions*) in permutation π_i.

A direct implementation of this definition yields an $O(n!)$ algorithm, as does the cofactor expansion method I learned in high school. Better algorithms to evaluate determinants are based on LU-decomposition, discussed in Section 13.1 (page 395). The determinant of M is simply the product of the diagonal elements of the LU-decomposition of M, which can be found in $O(n^3)$ time.

A closely related function called the *permanent* arises in combinatorial problems. For example, the permanent of the adjacency matrix of a graph G counts the number of perfect matchings in G. The permanent of a matrix M is defined by

$$perm(M) = \sum_{i=1}^{n!} \prod_{j=1}^{n} M[j, \pi_j]$$

differing from the determinant only in that all products are positive.

Surprisingly, it is NP-hard to compute the permanent, even though the determinant can easily be computed in $O(n^3)$ time. The fundamental difference is that $det(AB) = det(A) \times det(B)$, while $perm(AB) \neq perm(A) \times perm(B)$. There are permanent algorithms running in $O(n^2 2^n)$ time that prove to be considerably faster than the $O(n!)$ definition. Thus, finding the permanent of a 20×20 matrix is not out of the realm of possibility.

Implementations: The linear algebra package LINPACK contains a variety of Fortran routines for computing determinants, optimized for different data types and matrix structures. It can be obtained from Netlib, as discussed in Section 19.1.5 (page 659).

JScience provides an extensive linear algebra package (including determinants) as part of its comprehensive scientific computing library. *JAMA* is another matrix package written in Java. Links to both and many related libraries are available from *http://math.nist.gov/javanumerics/*.

Nijenhuis and Wilf [NW78] provide an efficient Fortran routine to compute the permanent of a matrix. See Section 19.1.10 (page 661). Cash [Cas95] provides a C routine to compute the permanent, motivated by the Kekulé structure count of computational chemistry.

Two different codes for approximating the permanent are provided by Barvinok. The first, based on [BS07], provides codes for approximating the permanent and a Hafnian of a matrix, as well as the number of spanning forests in a graph. See *http://www.math.lsa.umich.edu/∼barvinok/manual.html*. The second, based on [SB01], can provide estimates of the permanent of 200×200 matrices in seconds. See *http://www.math.lsa.umich.edu/∼barvinok/code.html*.

Notes: Cramer's rule reduces the problems of matrix inversion and solving linear systems to that of computing determinants. However, algorithms based on LU-determination are faster. See [BM53] for an exposition on Cramer's rule.

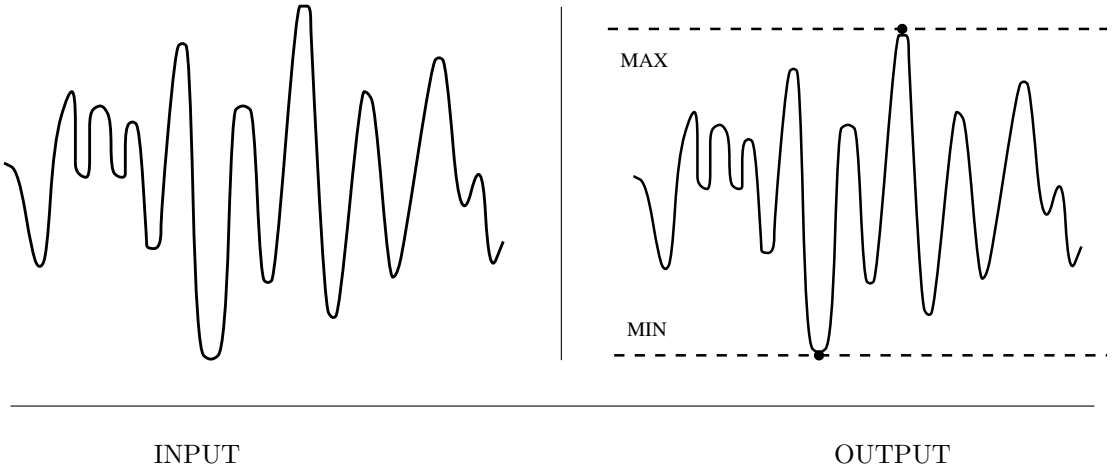
Determinants can be computed in $o(n^3)$ time using fast matrix multiplication, as shown in [AHU83]. Section 13.3 (page 401) discusses such algorithms. A fast algorithm for computing the sign of the determinant—an important problem for performing robust geometric computations—is due to Clarkson [Cla92].

The problem of computing the permanent was shown to be #P-complete by Valiant [Val79], where #P is the class of problems solvable on a "counting" machine in polynomial time. A counting machine returns the number of distinct solutions to a problem. Counting the number of Hamiltonian cycles in a graph is a #P-complete problem that is trivially NP-hard (and presumably harder), since any count greater than zero proves that the graph is Hamiltonian. Counting problems can be #P-complete even if the corresponding decision problem can be solved in polynomial time, as shown by the permanent and perfect matchings.

Minc [Min78] is the primary reference on permanents. A variant of an $O(n^2 2^n)$-time algorithm due to Ryser for computing the permanent is presented in [NW78].

Recently, probabilistic algorithms have been developed for estimating the permanent, culminating in a fully-polynomial randomized approximation scheme that provides an arbitrary close approximation in time that depends polynomially upon the input matrix and the desired error [JSV01].

Related Problems: Solving linear systems (see page 395), matching (see page 498), geometric primitives (see page 564).



13.5 Constrained and Unconstrained Optimization

Input description: A function $f(x_1, \ldots, x_n)$.

Problem description: What point $p = (p_1, \ldots, p_n)$ maximizes (or minimizes) the function f?

Discussion: Most of this book concerns algorithms that optimize one thing or another. This section considers the general problem of optimizing functions where, due to lack of structure or knowledge, we are unable to exploit the problem-specific algorithms seen elsewhere in this book.

Optimization arises whenever there is an objective function that must be tuned for optimal performance. Suppose we are building a program to identify good stocks to invest in. We have certain financial data available to analyze—such as the price-earnings ratio, the interest rate, and the stock price—all as a function of time t. The key question is how much weight we should give to each of these factors, where these weights correspond to coefficients of a formula:

$$\text{stock-goodness}(t) = c_1 \times \text{price}(t) + c_2 \times \text{interest}(t) + c_3 \times \text{PE-ratio}(t)$$

We seek the numerical values c_1, c_2, c_3 whose stock-goodness function does the best job of evaluating stocks. Similar issues arise in tuning evaluation functions for any pattern recognition task.

Unconstrained optimization problems also arise in scientific computation. Physical systems from protein structures to galaxies naturally seek to minimize their "energy" or "potential function." Programs that attempt to simulate nature thus often define potential functions assigning a score to each possible object configuration, and then select the configuration that minimizes this potential.

Global optimization problems tend to be hard, and there are lots of ways to go about them. Ask the following questions to steer yourself in the right direction:

- *Am I doing constrained or unconstrained optimization?* – In unconstrained optimization, there are no limitations on the values of the parameters other than that they maximize the value of f. However, many applications demand constraints on these parameters that make certain points illegal, points that might otherwise be the global optimum. For example, companies cannot employ less than zero employees, no matter how much money they think they might save doing so. Constrained optimization problems typically require mathematical programming approaches, like linear programming, discussed in Section 13.6 (page 411).

- *Is the function I am trying to optimize described by a formula?* – Sometimes the function that you seek to optimize is presented as an algebraic formula, such as finding the minimum of $f(n) = n^2 - 6n + 2^{n+1}$. If so, the solution is to analytically take its derivative $f'(n)$ and see for which points p' we have $f'(p') = 0$. These points are either local maxima or minima, which can be distinguished by taking a second derivative or just plugging p' back into f and seeing what happens. Symbolic computation systems such as Mathematica and Maple are fairly effective at computing such derivatives, although using computer algebra systems effectively is somewhat of a black art. They are definitely worth a try, however, and you can always use them to plot a picture of your function to get a better idea of what you are dealing with.

- *Is it expensive to compute the function at a given point?* – Instead of a formula, we are often given a program or subroutine that evaluates f at a given point. Since we can request the value of any given point on demand by calling this function, we can poke around and try to guess the maxima.

  Our freedom to search in such a situation depends upon how efficiently we can evaluate f. Suppose that $f\ (x_1, \ldots, x_n)$ is the board evaluation function in a computer chess program, such that x_1 is how much a pawn is worth, x_2 is how much a bishop is worth, and so forth. To evaluate a set of coefficients as a board evaluator, we must play a bunch of games with it or test it on a library of known positions. Clearly, this is time-consuming, so we would have to be frugal in the number of evaluations of f we use to optimize the coefficients.

- *How many dimensions do we have? How many do we need?* – The difficulty in finding a global maximum increases rapidly with the number of dimensions (or parameters). For this reason, it often pays to reduce the dimension by ignoring some of the parameters. This runs counter to intuition, for the naive programmer is likely to incorporate as many variables as possible into their evaluation function. It is just too hard, however, to optimize such complicated

functions. Much better is to start with the three to five most important variables and do a good job optimizing the coefficients for these.

- *How smooth is my function?* The main difficulty of global optimization is getting trapped in local optima. Consider the problem of finding the highest point in a mountain range. If there is only one mountain and it is nicely shaped, we can find the top by just walking in whatever direction is up. However, if there are many false summits, or other mountains in the area, it is difficult to convince ourselves whether we really are at the highest point. *Smoothness* is the property that enables us to quickly find the local optimum from a given point. We assume smoothness in seeking the peak of the mountain by walking up. If the height at any given point was a completely random function, there would be no way we could find the optimum height short of sampling every single point.

The most efficient algorithms for unconstrained global optimization use derivatives and partial derivatives to find local optima, to point out the direction in which moving from the current point does the most to increase or decrease the function. Such derivatives can sometimes be computed analytically, or they can be estimated numerically by taking the difference between the values of nearby points. A variety of *steepest descent* and *conjugate gradient* methods to find local optima have been developed—similar in many ways to numerical root-finding algorithms.

It is a good idea to try several different methods on any given optimization problem. For this reason, we recommend experimenting with the implementations below before attempting to implement your own method. Clear descriptions of these algorithms are provided in several numerical algorithms books, in particular *Numerical Recipes* [PFTV07].

For constrained optimization, finding a point that satisfies all the constraints is often the difficult part of the problem. One approach is to use a method for unconstrained optimization, but add a penalty according to how many constraints are violated. Determining the right penalty function is problem-specific, but it often makes sense to vary the penalties as optimization proceeds. At the end, the penalties should be very high to ensure that all constraints are satisfied.

Simulated annealing is a fairly robust and simple approach to constrained optimization, particularly when we are optimizing over combinatorial structures (permutations, graphs, subsets). Techniques for simulated annealing are described in Section 7.5.3 (page 254).

Implementations: The world of constrained/unconstrained optimization is sufficiently confusing that several guides have been created to point people to the right codes. Particularly nice is Hans Mittlemann's *Decision Tree for Optimization Software* at *http://plato.asu.edu/guide.html*. Also check out the selection at GAMS, the NIST *Guide to Available Mathematical Software*, at http://gams.nist.gov.

NEOS (Network-Enabled Optimization System) provides a unique service—the opportunity to solve your problem remotely on computers and software at Argonne

National Laboratory. Linear programming and unconstrained optimization are both supported. Check out *http://www-neos.mcs.anl.gov/* when you need a solution instead of a program.

Several of the *Collected Algorithms of the ACM* are Fortran codes for unconstrained optimization, most notably Algorithm 566 [MGH81] , Algorithm 702 [SF92] , and Algorithm 734 [Buc94] . Algorithm 744 [Rab95] does unconstrained optimization in Lisp. They are all available from Netlib (see Section 19.1.5 (page 659)).

General purpose simulated annealing implementations are available, and probably are the best place to start experimenting with this technique for constrained optimization. Feel free to try my code from Section 7.5.3 (page 254). Particularly popular is *Adaptive Simulated Annealing (ASA)*, written in C by Lester Ingber and available at *http://asa-caltech.sourceforge.net/*.

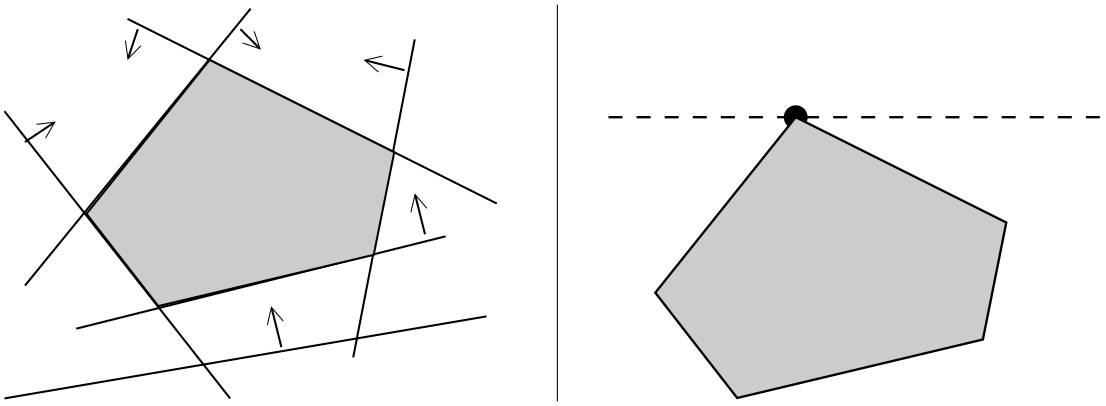
Both the Java Genetic Algorithms Package (JGAP) (*http://jgap.sourceforge.net/*) and the C Genetic Algorithm Utility Library (GAUL) (*http://gaul.sourceforge.net/*) are designed to aid in the development of applications that use genetic/evolutionary algorithms. I am highly skeptical about genetic algorithms (see Section 7.8 (page 266)), but other people seem to find them irresistible.

Notes: Steepest-descent methods for unconstrained optimization are discussed in most books on numerical methods, including [BT92, PFTV07]. Unconstrained optimization is the topic of several books, including [Bre73, Fle80].

Simulated annealing was devised by Kirkpatrick et al. [KGV83] as a modern variation of the Metropolis algorithm [MRRT53]. Both use Monte Carlo techniques to compute the minimum energy state of a system. Good expositions on all local search variations, including simulated annealing, appear in [AL97].

Genetic algorithms were developed and popularized by Holland [Hol75, Hol92]. More sympathetic expositions on genetic algorithms include [LP02, MF00]. Tabu search [Glo90] is yet another heuristic search procedure with a devoted following.

Related Problems: Linear programming (see page 411), satisfiability (see page 472).



INPUT OUTPUT

13.6 Linear Programming

Input description: A set S of n linear inequalities on m variables

$$S_i := \sum_{j=1}^{m} c_{ij} \cdot x_j \geq b_i,\ 1 \leq i \leq n$$

and a linear optimization function $f(X) = \sum_{j=1}^{m} c_j \cdot x_j$.

Problem description: Which variable assignment X' maximizes the objective function f while satisfying all inequalities S?

Discussion: Linear programming is the most important problem in mathematical optimization and operations research. Applications include:

- *Resource allocation* – We seek to invest a given amount of money to maximize our return. Often our possible options, payoffs, and expenses can be expressed as a system of linear inequalities such that we seek to maximize our possible profit given the constraints. Very large linear programming problems are routinely solved by airlines and other corporations.
- *Approximating the solution of inconsistent equations* – A set of m linear equations on n variables x_i, $1 \leq i \leq n$ is overdetermined if $m > n$. Such overdetermined systems are often *inconsistent*, meaning that there does not exist an assignment to the variables that simultaneously solves all the equations. To find the assignment that best fits the equations, we can replace each variable x_i by $x'_i + \epsilon_i$ and solve the new system as a linear program, minimizing the sum of the error terms.

- *Graph algorithms* – Many of the graph problems described in this book, including shortest path, bipartite matching, and network flow, can be solved as special cases of linear programming. Most of the rest, including traveling salesman, set cover, and knapsack, can be solved using *integer linear programming*.

The *simplex method* is the standard algorithm for linear programming. Each constraint in a linear programming problem acts like a knife that carves away a region from the space of possible solutions. We seek the point within the remaining region that maximizes (or minimizes) $f(X)$. By appropriately rotating the solution space, the optimal point can always be made to be the highest point in the region. The region (simplex) formed by the intersection of a set of linear constraints is convex, so unless we are at the top there is always a higher vertex neighboring any starting point. When we cannot find a higher neighbor to walk to, we have reached the optimal solution.

While the simplex algorithm is not too complex, there is considerable art to producing an efficient implementation capable of solving large linear programs. Large programs tend to be sparse (meaning that most inequalities use few variables), so sophisticated data structures must be used. There are issues of numerical stability and robustness, as well as choosing which neighbor we should walk to next (so-called *pivoting rules*). There also exist sophisticated *interior-point* methods, which cut through the interior of the simplex instead of walking along the outside, and beat simplex in many applications.

The bottom line on linear programming is this: you are much better off using an existing LP code than writing your own. Further, you are probably better off paying money than surfing the Web. Linear programming is an algorithmic problem of such economic importance that commercial implementations are far superior to free versions.

Issues that arise in linear programming include:

- *Do any variables have integrality constraints?* – It is impossible to send 6.54 airplanes from New York to Washington each business day, even if that value maximizes profit according to your model. Such variables often have natural integrality constraints. A linear program is called an *integer program* if all its variables have integrality constraints, or a *mixed integer program* if some of them do.

  Unfortunately, it is NP-complete to solve integer or mixed programs to optimality. However, there are integer programming techniques that work reasonably well in practice. *Cutting plane techniques* solve the problem first as a linear program, and then add extra constraints to enforce integrality around the optimal solution point before solving it again. After sufficiently many iterations, the optimum point of the resulting linear program matches that of the original integer program. As with most exponential-time algorithms,

run times for integer programming depend upon the difficulty of the problem instance and are unpredictable.

- *Do I have more variables or constraints?* – Any linear program with m variables and n inequalities can be written as an equivalent *dual* linear program with n variables and m inequalities. This is important to know, because the running time of a solver might be quite different on the two formulations. In general, linear programs (LPs) with much more variables than constraints should be solved directly. If there are many more constraints than variables, it is usually better to solve the dual LP or (equivalently) apply the dual simplex method to the primal LP.

- *What if my optimization function or constraints are not linear?* – In least-squares curve fitting, we seek the line that best approximates a set of points by minimizing the sum of squares of the distance between each point and the line. In formulating this as a mathematical program, the natural objective function is no longer linear, but quadratic. Although fast algorithms exist for least squares fitting, general *quadratic programming* is NP-complete.

  There are three possible courses of action when you must solve a nonlinear program. The best is if you can model it in some other way, as is the case with least-squares fitting. The second is to try to track down special codes for quadratic programming. Finally, you can model your problem as a constrained or unconstrained optimization problem and try to solve it with the codes discussed in Section 13.5 (page 407).

- *What if my model does not match the input format of my LP solver?* – Many linear programming implementations accept models only in so-called *standard form*, where all variables are constrained to be nonnegative, the object function must be minimized, and all constraints must be equalities (instead of inequalities).

  Do not fear. There exist standard transformations to map arbitrary LP models into standard form. To convert a maximization problem to a minimization one, simply multiply each coefficient of the objective function by -1. The remaining problems can be solved by adding *slack variables* to the model. See any textbook on linear programming for details. Modeling languages such as AMPC can provide a nice interface to your solver and deal with these issues for you.

Implementations: The USENET Frequently Asked Question (FAQ) list is a very useful resource on solving linear programs. In particular, it provides a list of available codes with descriptions of experiences. Check it out at *http://www-unix.mcs.anl.gov/otc/Guide/faq/*.

There are at least three reasonable choices in free LP-solvers. `Lp_solve`, written in ANSI C by Michel Berkelaar, can also handle integer and mixed-integer

problems. It is available at *http://lpsolve.sourceforge.net/5.5/*, and a substantial user community exists. The simplex solver `CLP` produced under the Computational Infrastructure for Operations Research is available (with other optimization software) at *http://www.coin-or.org/*. Finally, the GNU Linear Programming Kit (GLPK) is intended for solving large-scale linear programming, mixed integer programming (MIP), and other related problems. It is available at *http://www.gnu.org/software/glpk/*. In recent benchmarks among free codes (see *http://plato.asu.edu/bench.html*), `CLP` appeared to be fastest on on linear programming problems and `lp_solve` on mixed integer problems.

NEOS (Network-Enabled Optimization System) provides an opportunity to solve your problem on computers and software at Argonne National Laboratory. Linear programming and unconstrained optimization are both supported. This is worth checking out at *http://www.mcs.anl.gov/home/otc/Server/* if you need an answer instead of a program.

Algorithm 551 [Abd80] and Algorithm 552 [BR80] of the *Collected Algorithms of the ACM* are simplex-based codes for solving overdetermined systems of linear equations in Fortran. See Section 19.1.5 (page 659) for details.

Notes: The need for optimization via linear programming arose in logistics problems in World War II. The simplex algorithm was invented by George Danzig in 1947 [Dan63]. Klee and Minty [KM72] proved that the simplex algorithm is exponential in worst case, but it is very efficient in practice.

Smoothed analysis measures the complexity of algorithms assuming that their inputs are subject to small amounts of random noise. Carefully constructed worst-case instances for many problems break down under such perturbations. Spielman and Teng [ST04] used smoothed analysis to explain the efficiency of simplex in practice. Recently, Kelner and Spielman developed a randomized simplex algorithm running in polynomial time [KS05b].

Khachian's ellipsoid algorithm [Kha79] first proved that linear programming was polynomial in 1979. Karmarkar's algorithm [Kar84] is an interior-point method that has proven to be both a theoretical and practical improvement of the ellipsoid algorithm, as well as a challenge for the simplex method. Good expositions on the simplex and ellipsoid algorithms for linear programming include [Chv83, Gas03, MG06].

Semidefinite programming deals with optimization problems over symmetric positive semidefinite matrix variables, with linear cost function and linear constraints. Important special cases include linear programming and convex quadratic programming with convex quadratic constraints. Semidefinite programming and its applications to combinatorial optimization problems are surveyed in [Goe97, VB96].

Linear programming is P-complete under log-space reductions [DLR79]. This makes it unlikely to have an NC parallel algorithm, where a problem is in NC iff it can be solved on a PRAM in polylogarithmic time using a polynomial number of processors. Any problem that is P-complete under log-space reduction cannot be in NC unless P=NC. See [GHR95] for a thorough exposition of the theory of P-completeness, including an extensive list of P-complete problems.

Related Problems: Constrained and unconstrained optimization (see page 407), network flow (see page 509).

13.7 Random Number Generation

Input description: Nothing, or perhaps a seed.

Problem description: Generate a sequence of random integers.

Discussion: Random numbers have an enormous variety of interesting and important applications. They form the foundation of simulated annealing and related heuristic optimization techniques. Discrete event simulations run on streams of random numbers, and are used to model everything from transportation systems to casino poker. Passwords and cryptographic keys are typically generated randomly. Randomized algorithms for graph and geometric problems are revolutionizing these fields and establishing randomization as one of the fundamental ideas of computer science.

Unfortunately, generating random numbers looks a lot easier than it really is. Indeed, it is fundamentally impossible to produce truly random numbers on any deterministic device. Von Neumann [Neu63] said it best: "Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin." The best we can hope for are *pseudorandom* numbers, a stream of numbers that appear as if they were generated randomly.

There can be serious consequences to using a bad random-number generator. In one famous case, a Web browser's encryption scheme was broken with the discovery that the seeds of its random-number generator employed too few random bits [GW96]. Simulation accuracy is regularly compromised or invalidated by poor random number generation. This is an area where people shouldn't mess around, but they do. Issues to think about include:

- *Should my program use the same "random" numbers each time it runs?* – A poker game that deals you the exact same hand each time you play quickly loses interest. One common solution is to use the lower-order bits of the machine clock as the *seed* or starting point for a stream of random numbers, so that each time the program runs it does something different.

Such methods are adequate for games, but not for serious simulations. There are liable to be periodicities in the distribution of random numbers whenever calls are made in a loop. Also, debugging is seriously complicated when program results are not repeatable. Should your program crash, you cannot go back and discover why. A possible compromise is to use a deterministic pseudorandom-number generator, but write the current seed to a file between runs. During debugging, this file can be overwritten with a fixed initial value of the seed.

- *How good is my compiler's built-in random number generator?* – If you need uniformly-generated random numbers, and won't be betting the farm on the accuracy of your simulation, my recommendation is simply to use what your compiler provides. Your best opportunity to mess things up is with a bad choice of starting seed, so read the manual for its recommendations.

  If you *are* going to bet the farm on the results of your simulation, you had better test your random number generator. Be aware that it is very difficult to eyeball the results and decide whether the output is really random. This is because people have very skewed ideas of how random sources should behave and often see patterns that don't really exist. Several different tests should be used to evaluate a random number generator, and the statistical significance of the results established. The National Institute of Standards and Technology (NIST) has developed a test suite for evaluating random number generators, discussed below.

- *What if I must implement my own random-number generator?* – The standard algorithm of choice is the *linear congruential generator*. It is fast, simple, and (if instantiated with the right constants) gives reasonable pseudorandom numbers. The nth random number R_n is a function of the $(n-1)$st random number:

  $$R_n = (aR_{n-1} + c) \bmod m$$

  In theory, linear congruential generators work the same way roulette wheels do. The long path of the ball around and around the wheel (captured by $aR_{n-1} + c$) ends in one of a relatively small number of bins, the choice of which is extremely sensitive to the length of the path (captured by the mod m-truncation).

  A substantial theory has been developed to select the constants a, c, m, and R_0. The period length is largely a function of the modulus m, which is typically constrained by the word length of the machine.

  Note that the stream of numbers produced by a linear congruential generator repeats the instant the first number repeats. Further, computers are fast enough to make 2^{32} calls to a random-number generator in a few minutes. Thus, any 32-bit linear congruential generator is in danger of cycling, motivating generators with significantly longer periods.

- *What if I don't want such large random numbers?* – The linear congruential generator produces a uniformly-distributed sequence of large integers that can be easily scaled to produce other uniform distributions. For uniformly distributed real numbers between 0 and 1, use R_i/m. Note that 1 cannot be realized this way, although 0 can. If you want uniformly distributed integers between l and h, use $\lfloor l + (h - l + 1)R_i/m \rfloor$.

- *What if I need nonuniformly distributed random numbers?* – Generating random numbers according to a given nonuniform distribution can be a tricky business. The most reliable way to do this correctly is the acceptance-rejection method. We can bound the desired geometric region to sample from by a box and then select a random point p from the box. This point can be generated by selecting the x and y coordinates independently at random. If it lies within the region of interest, we can return p as being selected at random. Otherwise we throw it away and repeat with another random point. Essentially, we throw darts at random and report those that hit the target.

  This method is correct, but it can be slow. If the volume of the region of interest is small relative to that of the box, most of our darts will miss the target. Efficient generators for Gaussian and other special distributions are described in the references and implementations below.

  Be cautious about inventing your own technique, however, since it can be tricky to obtain the right probability distribution. For example, an *incorrect* way to select points uniformly from a circle of radius r would be to generate polar coordinates by selecting an angle from 0 to 2π and a displacement between 0 and r—both uniformly at random. In such a scheme, half the generated points will lie within $r/2$ of the center, when only one-fourth of them should be! This is different enough to seriously skew the results, while being sufficiently subtle that it can easily escape detection.

- *How long should I run my Monte Carlo simulation to get the best results?* – The longer you run a simulation, the more accurately the results should approximate the limiting distribution, thus increasing accuracy. However, this is true only until you exceed the *period*, or cycle length, of your random-number generator. At that point, your sequence of random numbers repeats itself, and further runs generate no additional information.

  Instead of jacking up the length of a simulation run to the max, it is usually more informative to do many shorter runs (say 10 to 100) with different seeds and then consider the range of results you see. The variance provides a healthy measure of the degree to which your results are repeatable. This exercise corrects the natural tendency to see a simulation as giving "the" correct answer.

Implementations: See *http://random.mat.sbg.ac.at* for an excellent website on random-number generation and stochastic simulation. It includes pointers to papers and literally dozens of implementations of random-number generators.

Parallel simulations make special demands on random-number generators. How can we ensure that random streams are independent on each machine? L'Ecuyer et.al. [LSCK02] provide object-oriented generators with a period length of approximately 2^{191}. Implementations in C, C++, and Java are available at *http://www.iro.umontreal.ca/~lecuyer/myftp/streams00/*. Independent streams of random numbers are supported for parallel applications. Another possibility is the *Scalable Parallel Random Number Generators Library (SPRNG)* [MS00], available at *http://sprng.cs.fsu.edu/*.

Algorithms 488 [Bre74], 599 [AKD83], and 712 [Lev92] of the *Collected Algorithms of the ACM* are Fortran codes for generating non-uniform random numbers according to several probability distributions, including the normal, exponential, and Poisson distributions. They are available from Netlib (see Section 19.1.5 (page 659)).

The National Institute of Standards [RSN+01] has prepared an extensive statistical test suite to validate random number generators. Both the software and the report describing it are available at *http://csrc.nist.gov/rng/*.

True random-number generators extract random bits by observing physical processes. The website *http://www.random.org* makes available random numbers derived from atmospheric noise that passes the NIST statistical tests. This is an amusing solution if you need a small quantity of random numbers (say, to run a lottery) instead a random-number generator.

Notes: Knuth [Knu97b] provides a thorough treatment of random-number generation, which I heartily recommend. He presents the theory behind several methods, including the middle-square and shift-register methods we have not described here, as well as a detailed discussion of statistical tests for validating random-number generators.

That said, see [Gen04] for more recent developments in random number generation. The Mersenne twister [MN98] is a fast random number generator of period $2^{19937} - 1$. Other modern methods include [Den05, PLM06]. Methods for generating nonuniform random variates are surveyed in [HLD04]. Comparisons of different random-number generators in practice include [PM88].

Tables of random numbers appear in most mathematical handbooks as relics from the days before there was ready access to computers. Most notable is [RC55], which provides one million random digits.

The deep relationship between randomness, information, and compressibility is explored within the theory of Kolmogorov complexity, which measures the complexity of a string by its compressibility. Truly random strings are incompressible. The string of seemingly random digits of π cannot be random under this definition, since the entire sequence is defined by any program implementing a series expansion for π. Li and Vitáni [LV97] provide a thorough introduction to the theory of Kolmogorov complexity.

Related Problems: Constrained and unconstrained optimization (see page 407), generating permutations (see page 448), generating subsets (see page 452), generating partitions (see page 456).

INPUT	OUTPUT
833816926455584605284210207 1	179424673 2038074743 * 22801763489 833816926455584605284210207 1

13.8 Factoring and Primality Testing

Input description: An integer n.

Problem description: Is n a prime number, and if not what are its factors?

Discussion: The dual problems of integer factorization and primality testing have surprisingly many applications for a problem long suspected of being only of mathematical interest. The security of the RSA public-key cryptography system (see Section 18.6 (page 641)) is based on the computational intractability of factoring large integers. As a more modest application, hash table performance typically improves when the table size is a prime number. To get this benefit, an initialization routine must identify a prime near the desired table size. Finally, prime numbers are just interesting to play with. It is no coincidence that programs to generate large primes often reside in the games directory of UNIX systems.

Factoring and primality testing are clearly related problems, although they are quite different algorithmically. There exist algorithms that can demonstrate that an integer is *composite* (i.e. , not prime) without actually giving the factors. To convince yourself of the plausibility of this, note that you can demonstrate the compositeness of any nontrivial integer whose last digit is 0, 2, 4, 5, 6, or 8 without doing the actual division.

The simplest algorithm for both of these problems is brute-force trial division. To factor n, compute the remainder of n/i for all $1 < i \leq \sqrt{n}$. The prime factorization of n will contain at least one instance of every i such that $n/i = \lfloor n/i \rfloor$, unless n is prime. Make sure you handle the multiplicities correctly, and account for any primes larger than $\sqrt{n}$.

Such algorithms can be sped up by using a precomputed table of small primes to avoid testing all possible i. Surprisingly large numbers of primes can be represented in surprisingly little space by using bit vectors (see Section 12.5 (page 385)). A bit vector of all odd numbers less than 1,000,000 fits in under 64 kilobytes. Even tighter encodings become possible by eliminating all multiples of 3 and other small primes.

Although trial division runs in $O(\sqrt{n})$ time, it is *not* a polynomial-time algorithm. The reason is that it only takes $\lg_2 n$ bits to represent n, so trial division takes time exponential in the input size. Considerably faster (but still exponential time) factoring algorithms exist, whose correctness depends upon more substantial number theory. The fastest known algorithm, the *number field sieve*, uses randomness to construct a system of congruences—the solution of which usually gives a factor of the integer. Integers with as many as 200 digits (663 bits) have been factored using this method, although such feats require enormous amounts of computation.

Randomized algorithms make it much easier to test whether an integer is prime. Fermat's little theorem states that $a^{n-1} = 1 (\text{mod } n)$ for all a not divisible by n, provided n is prime. Suppose we pick a random value $1 \leq a < n$ and compute the residue of $a^{n-1} (\text{mod } n)$. If this residue is not 1, we have just proven that n cannot be prime. Such randomized primality tests are very efficient. PGP (see Section 18.6 (page 641)) finds 300+ digit primes using hundreds of these tests in minutes, for use as cryptographic keys.

Although the primes are scattered in a seemingly random way throughout the integers, there is some regularity to their distribution. The *prime number theorem* states that the number of primes less than n (commonly denoted by $\pi(n)$) is approximately $n/\ln n$. Further, there never are large gaps between primes, so in general, one would expect to examine about $\ln n$ integers if one wanted to find the first prime larger than n. This distribution and the fast randomized primality test explain how PGP can find such large primes so quickly.

Implementations: Several general systems for computational number theory are available. PARI is capable of handling complex number-theoretic problems on arbitrary-precision integers (to be precise, limited to 80,807,123 digits on 32-bit machines), as well as reals, rationals, complex numbers, polynomials, and matrices. It is written mainly in C, with assembly code for inner loops on major architectures, and includes more than 200 special predefined mathematical functions. PARI can be used as a library, but it also possesses a calculator mode that gives instant access to all the types and functions. PARI is available at *http://pari.math.u-bordeaux.fr/*

LiDIA (*http://www.cdc.informatik.tu-darmstadt.de/TI/LiDIA/*) is the acronym for the C++ *Library for Computational Number Theory*. It implements several of the modern integer factorization methods.

A Library for doing Number Theory (NTL) is a high-performance, portable C++ library providing data structures and algorithms for manipulating signed, arbitrary length integers, and for vectors, matrices, and polynomials over the integers and over finite fields. It is available at *http://www.shoup.net/ntl/*.

Finally, MIRACL (Multiprecision Integer and Rational Arithmetic C/C++ Library) implements six different integer factorization algorithms, including the quadratic sieve. It is available at *http://www.shamus.ie/*.

Notes: Expositions on modern algorithms for factoring and primality testing include Crandall and Pomerance [CP05] and Yan [Yan03]. More general surveys of computational number theory include Bach and Shallit [BS96] and Shoup [Sho05].

In 2002, Agrawal, Kayal, and Saxena [AKS04] solved a long-standing open problem by giving the first polynomial-time deterministic algorithm to test whether an integer is composite. Their algorithm, which is surprisingly elementary for such an important result, involves a careful analysis of techniques from earlier randomized algorithms. Its existence serves as somewhat of a rebuke to researchers (like me) who shy away from classical open problems due to fear. Dietzfelbinger [Die04] provides a self-contained treatment of this result.

The Miller-Rabin [Mil76, Rab80] randomized primality testing algorithm eliminates problems with Carmichael numbers, which are composite integers that always satisfy Fermat's theorem. The best algorithms for integer factorization include the quadratic-sieve [Pom84] and the elliptic-curve methods [Len87b].

Mechanical sieving devices provided the fastest way to factor integers surprisingly far into the computing era. See [SWM95] for a fascinating account of one such device, built during World War I. Hand-cranked, it proved the primality of $2^{31} - 1$ in 15 minutes of sieving time.

An important problem in computational complexity theory is whether P = NP $\cap$ co-NP. The decision problem "is n a composite number?" used to be the best candidate for a counterexample. By exhibiting the factors of n, it is trivially in NP. It can be shown to be in co-NP, since every prime has a short proof of its primality [Pra75]. The recent proof that composite numbers testing is in P shot down this line of reasoning. For more information on complexity classes, see [GJ79, Joh90].

The integer RSA-129 was factored in eight months using over 1,600 computers. This was particularly noteworthy because in the original RSA paper [RSA78] they had originally predicted such a factorization would take 40 quadrillion years using 1970s technology. Bahr, Boehm, Franke, and Kleinjung hold the current integer factorization record, with their successful attack on the 200-digit integer RSA-200 in May 2005. This required the equivalent of 55 years of computations on a single 2.2 GHz Opteron CPU.

Related Problems: Cryptography (see page 641), high precision arithmetic (see page 423).

INPUT	OUTPUT
$\frac{4957829128749149515150890542586957 8}{7436743693123724272726335813880436 7}$	$\frac{2}{3}$

13.9 Arbitrary-Precision Arithmetic

Input description: Two very large integers, x and y.

Problem description: What is $x + y$, $x - y$, $x \times y$, and x/y?

Discussion: Any programming language rising above basic assembler supports single- and perhaps double-precision integer/real addition, subtraction, multiplication, and division. But what if we wanted to represent the national debt of the United States in pennies? One trillion dollars worth of pennies requires 15 decimal digits, which is far more than can fit into a 32-bit integer.

Other applications require *much* larger integers. The RSA algorithm for public-key cryptography recommends integer keys of at least 1000 digits to achieve adequate security. Experimenting with number-theoretic conjectures for fun or research requires playing with large numbers. I once solved a minor open problem [GKP89] by performing an exact computation on the integer $\binom{5906}{2953} \approx 9.93285 \times 10^{1775}$.

What should you do when you need large integers?

- *Am I solving a problem instance requiring large integers, or do I have an embedded application?* – If you just need the answer to a specific problem with large integers, such as in the number theory application above, you should consider using a computer algebra system like Maple or Mathematica. These provide arbitrary-precision arithmetic as a default and use nice Lisp-like programming languages as a front end—together often reducing your problem to a 5- to 10- line program.

  If you have an embedded application requiring high-precision arithmetic instead, you should use an existing arbitrary precision math library. You are likely to get additional functions beyond the four basic operations for computing things like greatest common divisor in the bargain. See the Implementations section for details.

- *Do I need high- or arbitrary-precision arithmetic?* – Is there an upper bound on how big your integers can get, or do you really need *arbitrary*-precision—i.e. , unbounded. This determines whether you can use a fixed-length array to represent your integers as opposed to a linked-list of digits. The array is likely to be simpler and will not prove a constraint in most applications.

- *What base should I do arithmetic in?* – It is perhaps simplest to implement your own high-precision arithmetic package in decimal, and thus represent each integer as a string of base-10 digits. However, it is far more efficient to use a higher base, ideally equal to the square root of the largest integer supported fully by hardware arithmetic.

  Why? The higher the base, the fewer digits we need to represent a number. Compare 64 decimal with 1000000 binary. Since hardware addition usually takes one clock cycle independent of value of the actual numbers, best performance is achieved using the highest supported base. The factor limiting us to base $b = \sqrt{maxint}$ is the desire to avoid overflow when multiplying two of these "digits" together.

  The primary complication of using a larger base is that integers must be converted to and from base-10 for input and output. The conversion is easily performed once all four high-precision arithmetical operations are supported.

- *How low-level are you willing to get for fast computation?* – Hardware addition is much faster than a subroutine call, so you take a significant hit on speed using high-precision arithmetic when low-precision arithmetic suffices. High-precision arithmetic is one of few problems in this book where inner loops in assembly language prove the right idea to speed things up. Similarly, using bit-level masking and shift operations instead of arithmetical operations can be a win if you really understand the machine integer representation.

The algorithm of choice for each of the five basic arithmetic operations is as follows:

- *Addition* – The basic schoolhouse method of lining up the decimal points and then adding the digits from right to left with "carries" runs to time linear in the number of digits. More sophisticated carry-look-ahead parallel algorithms are available for low-level hardware implementation. They are presumably used on your microprocessor for low-precision addition.

- *Subtraction* – By fooling with the sign bits of the numbers, subtraction can be a special considered case of addition: $(A - (-B)) = (A + B)$. The tricky part of subtraction is performing the "borrow." This can be simplified by always subtracting from the number with the larger absolute value and adjusting the signs afterwards, so we can be certain there will always be something to borrow from.

- *Multiplication* – Repeated addition will take exponential time on large integers, so stay away from it. The digit-by-digit schoolhouse method is reasonable to program and will work much better, presumably well enough for your application. On very large integers, Karatsuba's $O(n^{1.59})$ divide-and-conquer algorithm wins. Dan Grayson, author of Mathematica's arbitrary-precision arithmetic, found that the switch-over happened at well under 100 digits.

Even faster for very large integers is an algorithm based on Fourier transforms. Such algorithms are discussed in Section 13.11 (page 431).

- *Division* – Repeated subtraction will take exponential time, so the easiest reasonable algorithm to use is the long-division method you hated in school. This is fairly complicated, requiring arbitrary-precision multiplication and subtraction as subroutines, as well as trial-and-error, to determine the correct digit at each position of the quotient.

  In fact, integer division can be reduced to integer multiplication, although in a nontrivial way, so if you are implementing asymptotically fast multiplication, you can reuse that effort in long division. See the references below for details.

- *Exponentiation* – We can compute a^n using $n-1$ multiplications, by computing $a \times a \times \ldots \times a$. However, a much better divide-and-conquer algorithm is based on the observation that $n = \lfloor n/2 \rfloor + \lceil n/2 \rceil$. If n is even, then $a^n = (a^{n/2})^2$. If n is odd, then $a^n = a(a^{\lfloor n/2 \rfloor})^2$. In either case, we have halved the size of our exponent at the cost of at most two multiplications, so $O(\lg n)$ multiplications suffice to compute the final value:

```
function power(a, n)
    if (n = 0) return(1)
    x = power(a, ⌊n/2⌋)
    if (n is even) then return(x^2)
        else return(a × x^2)
```

High- but not arbitrary-precision arithmetic can be conveniently performed using the Chinese remainder theorem and modular arithmetic. The *Chinese remainder theorem* states that an integer between 1 and $P = \prod_{i=1}^{k} p_i$ is uniquely determined by its set of residues mod p_i, where each p_i, p_j are relatively prime integers. Addition, subtraction, and multiplication (but not division) can be supported using such residue systems, with the advantage that large integers can be manipulated without complicated data structures.

Many of these algorithms for computations on long integers can be directly applied to computations on polynomials. See the references for more details. A particularly useful algorithm is Horner's rule for fast polynomial evaluation. When $P(x) = \sum_{i=0}^{n} c_i \cdot x^i$ is blindly evaluated term by term, $O(n^2)$ multiplications will be performed. Much better is observing that $P(x) = c_0 + x(c_1 + x(c_2 + x(c_3 + \ldots)))$, the evaluation of which uses only a linear number of operations.

Implementations: All major commercial computer algebra systems incorporate high-precision arithmetic, including Maple, Mathematica, Axiom, and Macsyma. If you have access to one of these, this is your best option for a quick, nonembedded

application. The rest of this section focuses on source code available for embedded applications.

The premier C/C++ library for fast, arbitrary-precision is the GNU Multiple Precision Arithmetic Library (GMP), which operates on signed integers, rational numbers, and floating point numbers. It is widely used and well-supported, and available at *http://gmplib.org/*.

The `java.math BigInteger` class provides arbitrary-precision analogues to all of Java's primitive integer operators. `BigInteger` provides additional operations for modular arithmetic, GCD calculation, primality testing, prime generation, bit manipulation, and a few other miscellaneous operations.

A lower-performance, less-tested, but more personal implementation of high-precision arithmetic appears in the library from my book *Programming Challenges* [SR03]. See Section 19.1.10 (page 661) for details.

Several general systems for computational number theory are available. Each of these supports operations of arbitrary-precision integers. Information about the PARI, LiDIA, NTL and MIRACL number-theoretic libraries can be found in Section 13.8 (page 420).

ARPREC is a C++/Fortran-90 arbitrary precision package with an associated interactive calculator. MPFUN90 is a similar package written exclusively in Fortran-90. Both are available at *http://crd.lbl.gov/∼dhbailey/mpdist/*. Algorithm 693 [Smi91] of the *Collected Algorithms of the ACM* is a Fortran implementation of floating-point, multiple-precision arithmetic. See Section 19.1.5 (page 659).

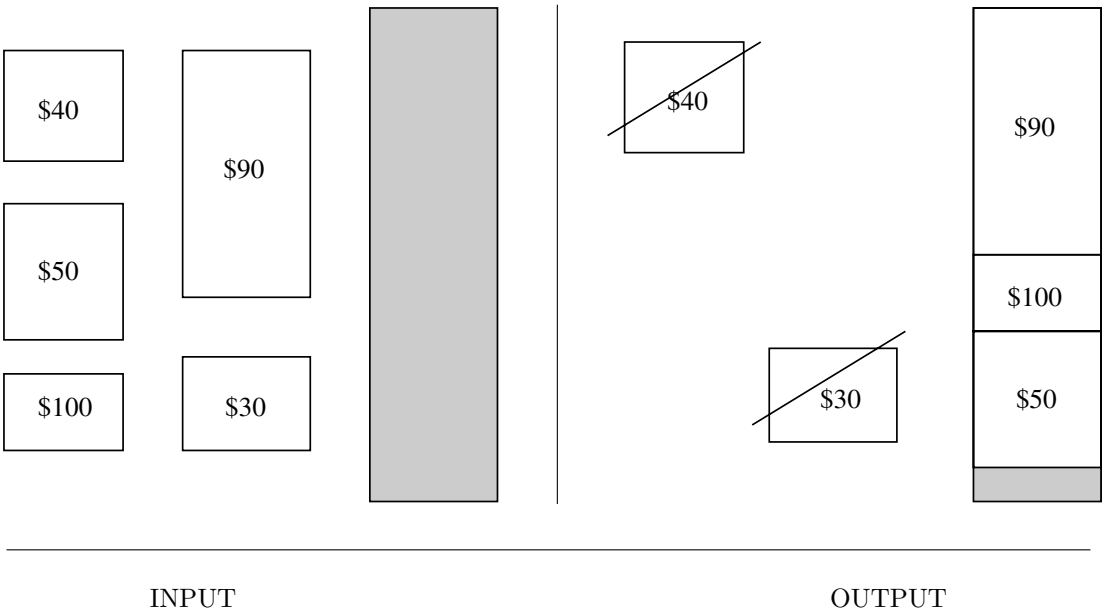
Notes: Knuth [Knu97b] is the primary reference on algorithms for all basic arithmetic operations, including implementations of them in the MIX assembly language. Bach and Shallit [BS96] and Shoup [Sho05] provide more recent treatments of computational number theory.

Expositions on the $O(n^{1.59})$-time divide-and-conquer algorithm for multiplication [KO63] include [AHU74, Man89]. An FFT-based algorithm multiplies two n-bit numbers in $O(n \lg n \lg \lg n)$ time and is due to Schönhage and Strassen [SS71]. Expositions include [AHU74, Knu97b]. The reduction between integer division and multiplication is presented in [AHU74, Knu97b]. Applications of fast multiplication to other arithmetic operations are presented by Bernstein [Ber04b]

Good expositions of algorithms for modular arithmetic and the Chinese remainder theorem include [AHU74, CLRS01]. A good exposition of circuit-level algorithms for elementary arithmetic algorithms is [CLRS01].

Euclid's algorithm for computing the greatest common divisor of two numbers is perhaps the oldest interesting algorithm. Expositions include [CLRS01, Man89].

Related Problems: Factoring integers (see page 420), cryptography (see page 641).



13.10 Knapsack Problem

Input description: A set of items $S = \{1, \ldots, n\}$, where item i has size s_i and value v_i. A knapsack capacity is C.

Problem description: Find the subset $S' \subset S$ that maximizes the value of $\sum_{i \in S'} v_i$, given that $\sum_{i \in S'} s_i \leq C$; i.e. , all the items fit in a knapsack of size C.

Discussion: The knapsack problem arises in resource allocation with financial constraints. How do you select what things to buy given a fixed budget? Everything has a cost and value, so we seek the most value for a given cost. The name *knapsack problem* invokes the image of the backpacker who is constrained by a fixed-size knapsack, and so must fill it only with the most useful and portable items.

The most common formulation is the *0/1 knapsack problem*, where each item must be put entirely in the knapsack or not included at all. Objects cannot be broken up arbitrarily, so it is not fair taking one can of Coke from a six-pack or opening a can to take just a sip. It is this 0/1 property that makes the knapsack problem hard, for a simple greedy algorithm finds the optimal selection when we are allowed to subdivide objects. We compute the "price per pound" for each item, and take the most expensive item or the biggest part thereof until the knapsack is full. Repeat with the next most expensive item. Unfortunately, the 0/1 constraint is usually inherent in most applications.

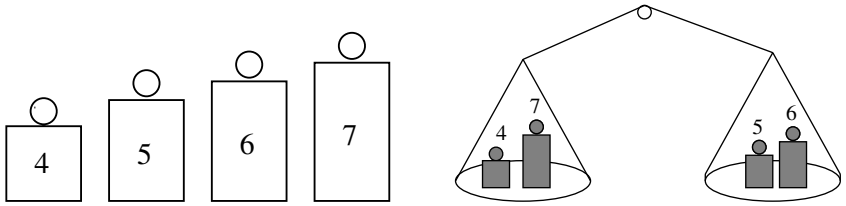


Figure 13.1: Integer partition is a special case of the knapsack problem

Issues that arise in selecting the best algorithm include:

- *Does every item have the same cost/value or the same size?* – When all items are worth exactly the same amount, we maximize our value by taking the greatest number of items. Therefore, the optimal solution is to sort the items in order of increasing size and insert them into the knapsack in this order until no more fit. The problem is similarly solved when each object has the same size but the costs are different. Sort by cost, and take the cheapest elements first. These are the easy cases of knapsack.

- *Does each item have the same "price per pound"?* – In this case, our problem is equivalent to ignoring the price and just trying to minimize the amount of empty space left in the knapsack. Unfortunately, even this restricted version is NP-complete, so we cannot expect an efficient algorithm that always solves the problem. Don't lose hope, however, because knapsack proves to be an "easy" hard problem, and one that can usually be handled with the algorithms described below.

  An important special case of a constant "price-per-pound" knapsack is the *integer partition* problem, presented in cartoon form in Figure 13.1. Here, we seek to partition the elements of S into two sets A and B such that $\sum_{a \in A} a = \sum_{b \in B}$, or alternately make the difference as small as possible. Integer partition can be thought of as bin packing into two equal-sized bins or knapsack with a capacity of half the total weight, so all three problems are closely related and NP-complete.

  The constant "price-per-pound" knapsack problem is often called the *subset sum* problem, because we seek a subset of items that adds up to a specific target number C; i.e. , the capacity of our knapsack.

- *Are all the sizes relatively small integers?* – When the sizes of the items and the knapsack capacity C are all integers, there exists an efficient dynamic programming algorithm to find the optimal solution in time $O(nC)$ and $O(C)$ space. Whether this works for you depends upon how big C is. It is great for $C \leq 1{,}000$, but not so great for $C \geq 10{,}000{,}000$.

The algorithm works as follows: Let S' be a set of items, and let $C[i, S']$ be true if and only if there is a subset of S' whose size adds up exactly to i. Thus, $C[i, \emptyset]$ is false for all $1 \leq i \leq C$. One by one we add a new item s_j to S' and update the affected values of $C[i, S']$. Observe that $C[i, S' \cup s_j] =$ true iff $C[i, S']$ or $C[i - s_j, S']$ is true, since we either use s_j in realizing the sum or we don't. We identify all sums that can be realized by performing n sweeps through all C elements—one for each s_j, $1 \leq j \leq n$—and so updating the array. The knapsack solution is given by the largest index of a true element of the largest realizable size. To reconstruct the winning subset, we must also store the name of the item number that turned $C[i]$ from false to true for each $1 \leq i \leq C$ and then scan backwards through the array.

This dynamic programming formulation ignores the values of the items. To generalize the algorithm, use each element of the array to store the value of the best subset to date summing up to i. We now update when the sum of the cost of $C[i - s_j, S']$ plus the cost of s_j is better than the previous cost of $C[i]$.

- *What if I have multiple knapsacks?* – When there are multiple knapsacks, your problem might be better thought of as a bin-packing problem. Check out Section 17.9 (page 595) for bin-packing/cutting-stock algorithms. That said, algorithms for optimizing over multiple knapsacks are provided in the Implementations section below.

Exact solutions for large capacity knapsacks can be found using integer programming or backtracking. A 0/1 integer variable x_i is used to denote whether item i is present in the optimal subset. We maximize $\sum_{i=1}^{n} x_i \cdot v_i$ given the constraint that $\sum_{i=1}^{n} x_i \cdot s_i \leq C$. Integer programming codes are discussed in Section 13.6 (page 411).

Heuristics must be used when exact solutions prove too costly to compute. The simple greedy heuristic inserts items according to the maximum "price per pound" rule described previously. Often this heuristic solution is close to optimal, but it can be arbitrarily bad depending upon the problem instance. The "price per pound" rule can also be used to reduce the problem size in exhaustive search-based algorithms by eliminating "cheap but heavy" objects from future consideration.

Another heuristic is based on *scaling*. Dynamic programming works well if the knapsack capacity is a reasonably small integer, say $\leq C_s$. But what if we have a problem with capacity $C > C_s$? We scale down the sizes of all items by a factor of C/C_s, round the size down to the nearest integer, and then use dynamic programming on the scaled items. Scaling works well in practice, especially when the range of sizes of items is not too large.

Implementations: Martello and Toth's collection of Fortran implementations of algorithms for a variety of knapsack problem variants are available at *http://www.or.deis.unibo.it/kp.html*. An electronic copy of the associated book [MT90a] has also been generously made available.

David Pisinger maintains a well-organized collection of C-language codes for knapsack problems and related variants like bin packing and container loading. These are available at *http://www.diku.dk/∼pisinger/codes.html*. The strongest code is based on the dynamic programming algorithm of [MPT99].

Algorithm 632 [MT85] of the *Collected Algorithms of the ACM* is a Fortran code for the 0/1 knapsack problem, with the twist that it supports multiple knapsacks. See Section 19.1.5 (page 659).

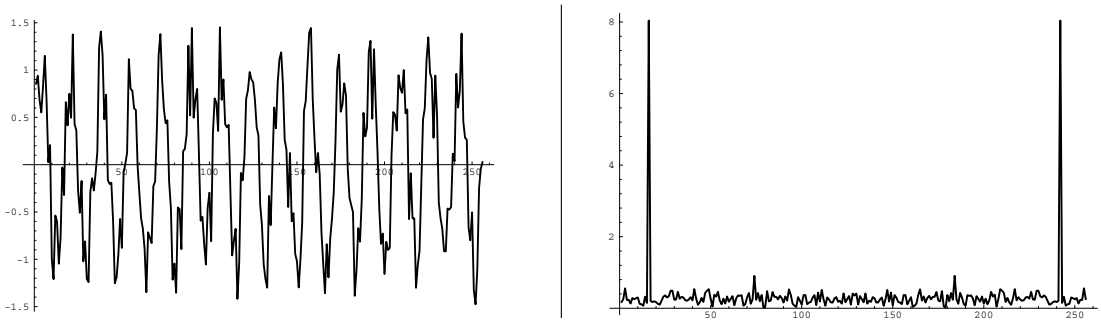
Notes: Keller, Pferschy, and Pisinger [KPP04] is the most current reference on the knapsack problem and variants. Martello and Toth's book [MT90a] and survey article [MT87] are standard references on the knapsack problem, including both theoretical and experimental results. An excellent exposition on integer programming approaches to knapsack problems appears in [SDK83]. See [MPT00] for a computational study of algorithms for 0-1 knapsack problems.

A polynomial-time approximation scheme is an algorithm that approximates the optimal solution of a problem in time polynomial in both its size and the approximation factor ϵ.

This very strong condition implies a smooth tradeoff between running time and approximation quality. Good expositions on polynomial-time approximation schemes [IK75] for knapsack and subset sum includes [BvG99, CLRS01, GJ79, Man89].

The first algorithm for generalized public key encryption by Merkle and Hellman [MH78] was based on the hardness of the knapsack problem. See [Sch96] for an exposition.

Related Problems: Bin packing (see page 595), integer programming (see page 411).



INPUT OUTPUT

13.11 Discrete Fourier Transform

Input description: A sequence of n real or complex values h_i, $0 \leq i \leq n-1$, sampled at uniform intervals from a function h.

Problem description: The discrete Fourier transform $H_m = \sum_{k=0}^{n-1} h_k e^{2\pi ikm/n}$ for $0 \leq m \leq n-1$.

Discussion: Although computer scientists tend to be fairly ignorant about Fourier transforms, electrical engineers and signal processors eat them for breakfast. Functionally, Fourier transforms provide a way to convert samples of a standard time-series into the *frequency domain*. This provides a dual representation of the function in which certain operations become easier than in the time domain. Applications of Fourier transforms include:

- *Filtering* – Taking the Fourier transform of a function is equivalent to representing it as the sum of sine functions. By eliminating undesirable high- and/or low-frequency components (i.e. , dropping some of the sine functions) and taking an inverse Fourier transform to get us back into the time domain, we can filter an image to remove noise and other artifacts. For example, the sharp spike in the figure above represents the period of the single sine function that closely models the input data. The rest is noise.
- *Image compression* – A smoothed, filtered image contains less information than the original, while retaining a similar appearance. By eliminating the coefficients of sine functions that contribute relatively little to the image, we can reduce the size of the image at little cost in image fidelity.
- *Convolution and deconvolution* – Fourier transforms can efficiently compute convolutions of two sequences. A *convolution* is the pairwise product of elements from two different sequences, such as in multiplying two n-variable

polynomials f and g or comparing two character strings. Implementing such products directly takes $O(n^2)$, while the fast Fourier transform led to a $O(n \lg n)$ algorithm.

Another example comes from image processing. Because a scanner measures the darkness of an image patch instead of a single point, the scanned input is always blurred. A reconstruction of the original signal can be obtained by deconvoluting the input signal with a Gaussian point-spread function.

- *Computing the correlation of functions* – The *correlation function* of two functions $f(t)$ and $g(t)$ is defined by

$$z(t) = \int_{-\infty}^{\infty} f(\tau)g(t+\tau)d\tau$$

and can be easily computed using Fourier transforms. When two functions are similar in shape but one is shifted relative to the other (such as $f(t) = \sin(t)$ and $g(t) = \cos(t)$), the value of $z(t_0)$ will be large at this shift offset t_0. As an application, suppose that we want to detect whether there are any funny periodicities in our random-number generator. We can generate a large series of random numbers, turn them into a time series (the ith number at time i), and take the Fourier transform of this series. Any funny spikes will correspond to potential periodicities.

The discrete Fourier transform takes as input n complex numbers h_k, $0 \leq k \leq n-1$, corresponding to equally spaced points in a time series, and outputs n complex numbers H_k, $0 \leq k \leq n-1$, each describing a sine function of given frequency. The discrete Fourier transform is defined by

$$H_m = \sum_{k=0}^{n-1} h_k e^{-2\pi ikm/n}$$

and the inverse Fourier transform is defined by

$$h_m = \frac{1}{n} \sum_{k=0}^{n-1} H_k e^{2\pi ikm/n}$$

which enables us move easily between h and H.

Since the output of the discrete Fourier transform consists of n numbers, each of which is computed using a formula on n numbers, they can be computed in $O(n^2)$ time. The fast Fourier transform (FFT) is an algorithm that computes the discrete Fourier transform in $O(n \log n)$. This is arguably the most important algorithm known, for it opened the door to modern signal processing. Several different algorithms call themselves FFTs, all of which are based on a divide-and-conquer approach. Essentially, the problem of computing the discrete Fourier transform on

n points is reduced to computing two transforms on $n/2$ points each, and is then applied recursively.

The FFT usually assumes that n is a power of two. If this is not the case, you are usually better off padding your data with zeros to create $n = 2^k$ elements rather than hunting for a more general code.

Many signal-processing systems have strong real-time constraints, so FFTs are often implemented in hardware, or at least in assembly language tuned to the particular machine. Be aware of this possibility if the codes prove too slow.

Implementations: FFTW is a C subroutine library for computing the discrete Fourier transform in one or more dimensions, with arbitrary input size, and supporting both real and complex data. It is the clear choice among freely available FFT codes. Extensive benchmarking proves it to be the "Fastest Fourier Transform in the West." Interfaces to Fortran and C++ are provided. FFTW received the 1999 J. H. Wilkinson Prize for Numerical Software. It is available at *http://www.fftw.org/*.

FFTPACK is a package of Fortran subprograms for the fast Fourier transform of periodic and other symmetric sequences, written by P. Swartzrauber. It includes complex, real, sine, cosine, and quarter-wave transforms. FFTPACK resides on Netlib (see Section 19.1.5 (page 659)) at *http://www.netlib.org/fftpack*. The GNU Scientific Library for C/C++ provides a reimplementation of FFTPACK. See *http://www.gnu.org/software/gsl/*.

Algorithm 545 [Fra79] of the *Collected Algorithms of the ACM* is a Fortran implementation of the fast Fourier transform optimizing virtual memory performance. See Section 19.1.5 (page 659) for further information.

Notes: Bracewell [Bra99] and Brigham [Bri88] are excellent introductions to Fourier transforms and the FFT. See also the exposition in [PFTV07]. Credit for inventing the fast Fourier transform is usually given to Cooley and Tukey [CT65], but see [Bri88] for a complete history.

A cache-oblivious algorithm for the fast Fourier transform is given in [FLPR99]. This paper first introduced the notion of cache-oblivious algorithms. The FFTW is based on this algorithm. See [FJ05] for more on the design of the FFTW.

An interesting divide-and-conquer algorithm for polynomial multiplication [KO63] does the job in $O(n^{1.59})$ time and is discussed in [AHU74, Man89]. An FFT-based algorithm that multiplies two n-bit numbers in $O(n \lg n \lg \lg n)$ time is due to Schönhage and Strassen [SS71] and is presented in [AHU74].

It is an open question of whether complex variables are really fundamental to fast algorithms for convolution. Fortunately, fast convolution can be used as a black box in most applications. Many variants of string matching are based on fast convolution [Ind98].

In recent years, wavelets have been proposed to replace Fourier transforms in filtering. See [Wal99] for an introduction to wavelets.

Related Problems: Data compression (see page 637), high-precision arithmetic (see page 423).