ALGORITHM THEORY

An *algorithm* is a systematic method to solve computational problems. The term itself comes from the name of a ninth century Persian mathematician, al-Khowarizmi, who wrote a book describing how to carry out addition, subtraction, multiplication, and division in the decimal number system, which was new at the time. Although the word has come into more widespread use after the advent of computers, the notion of a step-by-step solution has always existed (e.g., Euclid's method to compute the greatest common divisor of two integers is an algorithm). The inputs to the algorithm are called an *instance* of the problem. The aim in algorithm design is to come up with a procedure that is as efficient as possible in its use of resources. The number of steps, or time, and memory space needed by an algorithm are important measures of its efficiency. Both these quantities are typically expressed as functions of the *input size, n*.

The efficiency can greatly depend on, among other things, the *data structures* used by the algorithm. These are the means by which the algorithm stores and manipulates its data. Arrays, linked lists, and trees are some of the most basic data structures. Many sophisticated algorithms require data structures more advanced than these. For a much more thorough treatment of algorithms and data structures refer to Refs. 1, 2, and 3.

An algorithm for a problem provides us with an *upper bound* for its hardness. For example, if time is used as a measure of the hardness of a problem, an algorithm that takes time $T(n)$ on instances of size n tells us that this much time is *sufficient* to solve the problem. An equally interesting question is: How much time is *necessary* to solve the problem? The discipline of Complexity Theory is concerned with such issues.

Even though algorithms capture notions of mechanical processing of data, their design itself is essentially a creative process. There are several general paradigms for algorithmic design. We will describe three important ones here—divide-and-conquer, greedy, and dynamic programming. This will be followed by a section on a few important data structures. We then touch on the basics of Complexity Theory and end with an introduction to randomized and approximation algorithms, two variations on the standard notion of an algorithm that have gained in importance in recent years. First, we define some terms and explain notation that might not be standard.

DEFINITIONS AND NOTATION

$\mathbb{R}$ stands for the field of real numbers. Let f,g be functions from the integers to $\mathbb{R}$. Then, $f = O(g)$ if there exist constants c and n_0 such that $f(n) \leq cg(n)$ for all $n \geq n_0$; $f = \Omega(g)$ if there exist constants c and n_0 such that $f(n) \geq cg(n)$ for all $n \geq n_0$;

$f = \Theta(g)$ if both $f = O(g)$ and $f = \Omega(g)$. Clearly, $f = O(g)$ if and only if $g = \Omega(f)$.

An *alphabet,* Σ, is a finite set of symbols. A *string* over an alphabet is obtained by concatenating (zero, finitely many, or infinitely many) symbols from the alphabet. In this article, Σ^* denotes the set of all finite strings (including the empty string) over the alphabet {0, 1}. A *language* is a subset of Σ^* and can be thought of as a function $f : \Sigma^* \to \{0, 1\}$, where x is in the language if and only if $f(x) = 1$.

An *undirected* graph G consists of a set of vertices V and a set of edges E, where each edge connects a pair of vertices. An edge connecting u and v is written (u, v). In a *directed* graph, edges also have a direction, and an edge written (u, v) is directed from u to v. (Unless otherwise stated all graphs in this article are undirected.) Another variation is a *weighted* graph where every edge is given a numerical weight. A (directed) graph is said to be *connected* if any two vertices in it are connected by a (directed) path. An *acyclic* graph is a graph that has no cycles, i.e., no path from a node back to itself. A connected, acyclic graph is called a *tree.*

DIVIDE-AND-CONQUER ALGORITHMS

These algorithms exploit the recursive nature of some problems. The problem is split into smaller subproblems whose structures are identical or made similar to that of the original problem. These subproblems are recursively solved, and the solutions are combined (if needed). Most of the work is done in either the splitting stage (as in quicksort) or the combining stage (as in mergesort) but usually not in both. The recursion involved may be routine or intricate, but the subproblems solved recursively generally have size a constant fraction of the original. Some examples of divide-and-conquer algorithms follow.

Binary Search

Computers need to manipulate massive amounts of data quickly. Often the data are a collection of items, each identified by a particular key. A basic operation is to search for a particular item by searching for its key. When the data are unorganized, we can do little better than look at half the items on the average. Consequently, various ways to speed up searching by organizing the data more effectively have been invented. The simplest is to store the data in an array, sorted according to the key. Sorting the data efficiently is another problem altogether. We present a divide-and-conquer sorting algorithm later. Here is a simple divide-and-conquer algorithm that can be used to do the search in sorted data.

Let A be an array of n sorted items, and assume we want to search for a key k. By comparing k with the midpoint of A, we can remove half of the array from further consideration. So, the size of the array to be searched is now reduced by half. Repeating this process, we can find k, or determine that it is absent, in at most $\log_2 n$ comparisons. At every stage, the size of the subproblem that needs to be solved is halved.

Mergesort

The mergesort algorithm is a typical application of the divide-and-conquer paradigm. Let A be an n-element array to be sorted. Recursively sort the left and right halves of A. Then, merge the two sorted subarrays. The recursion bottoms out when the subarray to be sorted has only one element. The merging process is done as follows. Let A and B be the two $n/2$-element sorted subarrays to be merged. Let C be an n-element array that will store the output. It is initially empty. Let p_A and p_B be two pointers that run through A and B, respectively. Initially, they point at their first elements. At each step, we look at the elements pointed to by p_A and p_B, write the smaller of the two in the next empty location of C, and advance that pointer one location to the right. It is clear that in n steps the sorted output appears in C. The running time of this algorithm is measured by the number of comparisons performed. Let $T(n)$ be the number of comparisons required, in the worst case, to sort an n-element array using mergesort. We can write a recurrence for $T(n)$ as follows:

$$T(n) = \begin{cases} 1 & \text{if } n = 2 \\ 2T(n/2) + n & \text{otherwise} \end{cases}$$

As in most divide-and-conquer algorithms, the second term above is the sum of three quantities—the time required to perform the divide (nothing), the time required to solve the subproblems $[2T(n/2)]$, and the time required to combine their solutions (n). Solving this recurrence, we can see that $T(n)$ is $\Theta(n \log n)$.

Quicksort

The quicksort algorithm was invented by Hoare (4). Unlike mergesort, which has a running time of $\Theta(n \log n)$ in the worst case, quicksort has a worst-case running time of $\Theta(n^2)$. However, on the average, its running time is $\Theta(n \log n)$, and it sorts in place, that is, the amount of extra storage it requires does not depend on n. Good implementations of quicksort are often significantly faster than mergesort. Let $A[1..n]$ be the array to be sorted. The elements of A are rearranged and an index p computed so that all the elements of the subarray $A[1..p]$ are smaller than all the elements of the subarray $A[(p + 1)..n]$. These two subarrays are then recursively sorted (in place). An array with only one element is the base case of the recursion and is already sorted. The key step in this algorithm is the partitioning. It is done by choosing an appropriate element (e.g., a random element) as the pivot and shuffling A so that elements smaller than the pivot lie in the left subarray and those larger than it lie in the right one.

Strassen's Matrix Multiplication Algorithm

This algorithm, due to Strassen (5), is a very famous application of the divide-and-conquer technique. The naive algorithm to multiply two $n \times n$ matrices requires time $\Theta(n^3)$ because n multiplications are required to compute each of the n^2 entries in the product matrix. Strassen's divide-and-conquer algorithm uses seven recursive multiplications of $n/2 \times n/2$ matrices and an additional $\Theta(n^2)$ operations, yielding a total time of $\Theta(n^{\log_2 7}) \approx \theta(n^{2.81})$. We now describe how the algorithm works.

Let A and B be two $n \times n$ matrices to be multiplied. Assume that n is a power of 2. Partition the matrices into four $(n/2) \times (n/2)$ submatrices as follows:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \qquad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

Strassen discovered the amazing fact that their product C can be expressed as

$$C = \begin{bmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 + M_3 - M_2 + M_6 \end{bmatrix}$$

where

$$\begin{aligned} M_1 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\ M_2 &= (A_{21} + A_{22})B_{11} \\ M_3 &= A_{11}(B_{12} - B_{22}) \\ M_4 &= A_{22}(B_{21} - B_{11}) \\ M_5 &= (A_{11} + A_{12})B_{22} \\ M_6 &= (A_{21} - A_{11})(B_{11} + B_{12}) \\ M_7 &= (A_{12} - A_{22})(B_{21} + B_{22}) \end{aligned}$$

The best matrix multiplication algorithm invented so far runs in time $n^{2.376}$ (6). Because the product matrix has n^2 entries, no algorithm can do better than $\Theta(n^2)$.

Polynomial Multiplication

Two linear polynomials $ax + b$ and $cx + d$, where a, b, c, d are integers, can be multiplied using only three integer multiplications (instead of four). To see this, notice that $(ax + b)(cx + d) = acx^2 + bd + [(a + b)(c + d) - ac - bd]x$, and so computing ac, bd, and $(a + b)(c + d)$ suffices. The same idea can be used to multiply two arbitrary polynomials. Let $p(x) = \sum_{i=0}^{n} p_i x^i$ and $q(x) = \sum_{i=0}^{n} q_i x^i$. We can assume that n is odd. Rewriting, $p(x) = p_1(x)x^{(n+1)/2} + p_2(x)$ and $q(x) = q_1(x)x^{(n+1)/2} + q_2(x)$, where p_1, p_2, q_1, q_2 are degree $(n - 1)/2$ polynomials. They play the role of a, b, c, d. Thus, two degree n polynomials can be multiplied in time $\Theta(n^{\log_2 3})$ by a straightforward divide-and-conquer approach. This, however, is not the best possible algorithm. Using more advanced techniques (the Fast Fourier Transform (FFT) algorithm), we can do the multiplication in $\Theta(n \log n)$ time. The FFT algorithm itself is a divide-and-conquer algorithm.

GREEDY ALGORITHMS

Greedy algorithms are typically used for optimization problems, where one is interested in selecting the best solution from among a large set. For example, consider the problem of computing a spanning tree of a weighted graph while minimizing the sum of the weights on the tree edges (see below). Greedy algorithms go through several stages. At each stage they make the choice that seems best at a local level. Thus, very often these algorithms have a simple construction, but it takes much greater effort to prove rigorously that the final solution produced is indeed optimal. Many problems amenable to the greedy approach can be represented as a specific problem on structures known as *weighted matroids*. A weighted matroid is a pair (S, I) where S is the ground set and I is a family of subsets of S, which obeys certain properties. Each element of S has a positive weight. The problem is to find a member of I with maximum weight, where the weight of a subset is the sum of the weights of its elements. We now consider a few important examples of the greedy method.

Single Source Shortest Path

Given a directed graph G with nonnegative weights on the edges and a distinguished vertex v, find shortest (minimum-weight) paths from v to every vertex of G. The following algorithm, due to Dijkstra, solves this problem. It maintains a set F of vertices whose shortest paths have been found. At each step, it chooses an appropriate vertex outside F and adds it to F. Each vertex is given a weight, which is updated as the algorithm proceeds, as long as the vertex is outside F. This weight is the cost of the best path found so far from v to the vertex. After a vertex is in F, its weight is final and is the cost of the shortest path from v to it. Initially, F is empty, v is given a weight 0, and every other vertex of G has a weight infinity. The vertex chosen at every step is the one with minimum weight, a greedy choice. This vertex (call it u) is then added to F and for every edge (u, w) leaving u, the weight of w is updated to the value min[wt(w), wt(u) + edge_wt(u, w)], where wt(w) is the weight of w and edge_wt(u, w) is the weight of the edge from u to w. The running time is $O(V^2)$, where V is the number of vertices in G. This can be improved for sparse graphs by using advanced data structures.

Minimum Spanning Tree

A *spanning tree* of a connected graph G is a subgraph that is a tree that connects every vertex of G. The goal is to find a spanning tree of a weighted graph that minimizes the sum of the weights of the edges in the tree. Let V be the number of vertices and E, the number of edges in G. Two important algorithms for this problem are those by Kruskal and Prim.

Kruskal's algorithm maintains a forest (a collection of disjoint trees) initialized to the set of vertices of G. The edges of G are sorted by weight and at every step, a minimum-weight edge connecting two trees in this forest is added to it. The algorithm ends when all the edges of G have been considered. A running time of $O(E \log E)$ can be achieved by using suitable data structures.

Prim's algorithm "grows" a single tree starting from an arbitrary vertex of G. At each step, an edge connecting a vertex in the tree to a vertex outside it is added. The edge chosen is a least-weight edge with this property. The total running time is $O(E + V \log V)$ with appropriate data structures.

Maximum Flow

A *flow network* is a directed graph with two distinguished nodes, a *source* s and a *sink* t and capacities on the edges. The source has no edges entering it, and the sink has no edges leaving it. The capacity of an edge is the maximum number of flow units that can be pushed along that edge. The law of flow conservation holds at every node except s and t. This means that the total flow coming into a node equals the total flow leaving it. Flow networks model many real-life situations, like communication networks and electrical networks. A key problem in this model is to determine how many units of flow the network can support. A simple greedy algorithm to accomplish this is the Ford–Fulkerson algorithm. It repeatedly finds a path from s to t (called an *augmenting path*) that can admit more flow and increases the flow along this path to the extent possible. It terminates when no augmenting path can be found. It can be shown that this procedure can be inefficient if the choice of the augmenting path is not done prop-

erly. A modified version of this algorithm, called the Edmonds–Karp algorithm, chooses the shortest (in length, not by capacity) such path as the augmenting path. Such a choice guarantees a running time of $O(VE^2)$, where V is the number of vertices and E is the number of edges of G.

An important result in this context is the *max-flow min-cut theorem.* A *cut* is a partition of the vertex set into two parts S and $T = V \backslash S$ such that $s \in S$ and $t \in T$. The *capacity* of a cut is the maximum amount that can flow from S to T across the cut (i.e., the sum of the capacities of all the directed edges going from S to T). The theorem states that the value of the maximum flow in the network equals the minimum value of any cut. This theorem is the key to proving the correctness of many maximum-flow algorithms.

Activity Selection

Assume that we are given a set $a_1, \ldots, a_n$ of activities each with a start time and a finish time, all competing for a single resource. Our task is to select a maximum set S of activities that can use the resource without conflict. This problem can be modeled as follows: a set of potentially overlapping intervals in $\mathbb{R}$ is given, and the goal is to select a maximum set S of mutually nonoverlapping intervals.

A simple greedy algorithm can be used to solve this problem. Initially, S is empty. At each stage, from among the activities that do not conflict with the current S, an activity with the earliest finish time is picked. It is clear that this algorithm requires a running time of $\Theta(n)$ with an additional $\Theta(n \log n)$ time for initially sorting the activities in nondecreasing order of finish time.

Job Sequencing

Suppose that there are n jobs $j_1, \ldots, j_n$, each requiring unit time to complete on a single processor. Each job has a deadline d_i associated with it, and a penalty p_i if it is not completed by its deadline. We are required to sequence the jobs starting at time $t = 0$, so that the total penalty incurred is minimized. The algorithm picks a maximal subset of jobs that can be scheduled in such a way that no job is late. This subset is chosen in a greedy way, by picking jobs in nonincreasing order of penalty as long as the jobs chosen can be sequenced without incurring any penalty. To check that the current subset (that has k jobs, say) can be sequenced with no penalty, check that for every i, $1 \le i \le k$, there are at most i jobs with a deadline at time-step i or earlier. The final schedule is this maximal subset ordered by nondecreasing deadlines followed by all the other jobs in any order. Because the checking procedure can be done in time $O(n)$, the total time required is $O(n^2)$.

DYNAMIC PROGRAMMING

Like greedy algorithms, dynamic programming algorithms are also used to solve optimization problems. Like divide-and-conquer algorithms, they too use solutions to smaller subproblems to compute a solution to the given problem. But unlike divide-and-conquer algorithms, they solve many, though carefully chosen, subproblems. For example, in the matrix chain multiplication problem that follows, only products of matrices in a consecutive sequence are evaluated. Moreover, these subproblems might not be independent, that is, they may have still smaller subproblems in common. This makes a recursive implementation highly inefficient. So, in practice, these algorithms are written in a bottom-up fashion, smaller problems being solved first, with their solutions being stored to be used later. Here are some examples of dynamic programming.

Matrix Chain Multiplication

Given n matrices $M_1, \ldots, M_n$, such that M_i has dimension $d_i \times d_{i+1}$ for $1 \le i \le n$, it is required to find a parenthesization of the product $M_1 \cdots M_n$ that minimizes the number of matrix entries multiplied. Let p_{ij} be the least number of multiplications required to compute the product $M_i \cdots M_j$. A recursive formula for p_{ij} is

$$p_{ij} = \begin{cases} 0 & \text{if } i = j \\ \min_{k=i}^{j-1}(p_{ik} + p_{k+1,j} + d_i d_{k+1} d_{j+1}) & \text{otherwise} \end{cases}$$

The index k is the place where a parenthesization splits $M_i \cdots M_j$. Because the implementation is bottom-up, i.e., the p_{ij} values are evaluated in increasing order of the difference $j - i$, the values p_{ik} and $p_{k+1,j}$ have already been computed and stored when computing p_{ij}. Therefore, p_{ij} can be evaluated in $\Theta(j - i)$ time. Because there are $\Theta(n^2)$ such i,j pairs, the entire algorithm runs in time $O(n^3)$.

All-Pairs Shortest Paths

Dijkstra's algorithm solves the single-source shortest path problem for graphs with no negative weight edges in time $O(n^2)$, where n is the number of vertices in the graph. The simple dynamic programming algorithm in this section produces the weight of the minimum-weight paths between all pairs of vertices even if the graph has negative weight edges. However, it is assumed that the graph does not have any negative weight cycles; otherwise, the problem is not well defined. The running time of this algorithm is $O(n^4)$. A small modification improves this to $O(n^3 \log n)$.

Let the vertices of the graph be labeled $1 \cdots n$. Let w_{ij}^m denote the weight of the min-weight path from vertex i to vertex j with at most m edges, $m \ge 1$. Let $w_{ii}^0 = 0$ and $w_{ij}^0 = \infty$ for i $\ne j$. For any vertex $k \ne j$, we can express the minimum-weight of a path from i to j with at most m edges ($m \ge 1$) and with k as j's predecessor, as the sum w_{ik}^{m-1} + edge-wt(k, j), where edge-wt(k, j) is the weight of the edge (k, j) (infinity if k and j are not connected by an edge). Therefore, to get the weight of a min-weight path of length at most m from i to j, we minimize this expression with respect to k, including $k = j$ with the understanding that edge-wt(j, j) = 0. Writing it out as a formula, we get

$$w_{ij}^m = \min_{k=1}^{n} \left(w_{ik}^{m-1} + \text{edge-wt}(k, j)\right)$$

Let P_1 be the matrix whose (i, j)th entry is the weight of the edge (i, j) if one exists and is infinity otherwise. Define an operation $\circ$ on matrices A, B as follows: $A \circ B = C$, where $C(i, j) = \min_{k=1}^{n}[A(i, k) + B(k, j)]$. Define $P_m = P_{m-1} \circ P_1$, for $m > 1$. Then, $P_m(i, j)$ is the weight of an optimal path from i to j among those having at most m edges. Because any optimal path from i to j has at most $n - 1$ edges, $P_{n-1}(i, j)$ is the weight of the optimal path from i to j. We can compute P_{n-1}

by a matrix-multiplication-like procedure, by replacing the usual $\cdot$ and + operations by + and min, respectively. This gives us a $O(n^4)$ algorithm. We can do better by only computing P_{2^i} for $i = 0, 1, 2, \ldots, t$, where t is the least integer such that $2^t > n - 1$. Clearly, $P_{2^t} = P_{n-1}$. This modified algorithm runs in time $O(n^3 \log n)$. The actual paths can be constructed by a simple extension of the algorithm.

A more efficient algorithm for this problem is the Floyd–Warshall algorithm (7), which also uses dynamic programming, but only requires time $O(n^3)$. It "inducts" on the maximum-numbered vertex on a path from i to j. Let S_{ij}^k be the weight of a min-weight path from i to j that has no vertex numbered higher than k as an intermediate vertex. Then, $S_{ij}^k = \min(S_{ij}^{k-1}, S_{ik}^{k-1} + S_{kj}^{k-1})$ if $k \geq 1$ and $S_{ij}^0 =$ edge-wt(i, j). We are interested in S_{ij}^n for all i, j. Because i, j, k are all bounded by n, this is an $O(n^3)$ algorithm. For graphs with no negative weight edges, we could use Dijkstra's algorithm n times, a total time of $O(n^3)$.

Polygon Triangulation

Given an n-sided convex polygon $\langle v_1, \ldots, v_n\rangle$ in the plane (the polygon edges are $v_1v_2, v_2v_3, \ldots, v_{n-1}v_n, v_nv_1$ in counterclockwise order), partition the polygon into triangles of the form $\Delta v_iv_kv_j$ so as to minimize the sum of the perimeters of the triangles (called the *weight* of the triangulation).

Again, we consider the general problem of optimally triangulating the polygon $\langle v_i, \ldots, v_j\rangle$, $j > i$. Let T_{ij} denote the weight of an optimal triangulation of this polygon. T_{1n} is our final answer. We proceed in a bottom-up manner, in increasing order of $j - i$. If $j = i + 1$, then $T_{ij} = 0$. Otherwise, consider the triangle $v_iv_kv_j$ for some k. This divides the remaining part of the polygon $\langle v_i, \ldots, v_j\rangle$ into two subpolygons, $\langle v_i, \ldots, v_k\rangle$ and $\langle v_k, \ldots, v_j\rangle$. Thus, $T_{ij} = \min_{k=i+1}^{j-1}(T_{ik} + T_{kj} +$ perimeter of $\Delta v_iv_kv_j)$. The quantities T_{ik} and T_{kj} have already been computed, so T_{ij} can be evaluated in time $\theta(j - i - 1)$. Therefore, this is an $O(n^3)$-time algorithm, but one that computes only the weight of an optimal triangulation. However, the triangulation itself can also be easily constructed by storing, for all i, j, the value of k that minimizes the previous expression for T_{ij}.

0-1 Knapsack

The knapsack problem can be illustrated by the following hypothetical scenario. A thief breaks into a house that has n items of values $v_1, \ldots, v_n$ and integer weights $w_1, \ldots, w_n$. His knapsack, however, can carry only a total weight of M. How should he choose the items so that the value of his booty is maximized? This is called the 0-1 knapsack problem to distinguish it from a variation where the thief is allowed to take a fraction of an item. Of course, this problem is applicable in many other situations, more important and wholesome.

There is an easy dynamic programming algorithm for this problem. Let $P(i, W)$, for $1 \leq i \leq n$ and $1 \leq W \leq M$, be the optimal profit if the thief restricts himself to the first i items and a maximum weight of W, and let $S(i, W)$ denote the set of items he chooses. For the base case, we have, $P(1, W) = v_1$ and $S(1, W) = \{1\}$ if $w_1 \leq W$, and $P(1, W) = 0$ and $S(1, W) = \Phi$ otherwise. For $i > 1$, if $w_i > W$, item i cannot be taken. So $P(i, W) = P(i - 1, W)$ and $S(i, W) = S(i - 1, W)$. If $w_i \leq W$, item i may or may not be taken, so $P(i, W) = \max[P(i - 1, W - w_i) + v_i, P(i - 1, W)]$, the two terms inside the parentheses corresponding to the two possibilities. If the first term is greater, $S(i, W) = S(i - 1, W - w_i) \cup \{i\}$, else $S(i, W) = S(i - 1, W)$. Since $1 \leq i \leq n$ and $1 \leq W \leq M$, this is an $O(nM)$ algorithm. It works well if M is relatively small (i.e., polynomially bounded in n). In general, because M is part of the input, the input size is proportional to $\log |M|$; therefore, the running time of this algorithm is exponential in terms of the input size. Such algorithms are called *pseudo-polynomial* algorithms. In fact, the 0-1 knapsack problem with no restriction on M is a well-known NP-hard problem and is not expected to be solvable in polynomial time (refer to the section Complexity Theory).

DATA STRUCTURES

Arrays and Linked Lists

The array is one of the simplest and most fundamental data structures, provided as a primitive in most programming languages and used to build other data structures. An array A is a set of items $A[1], \ldots, A[n]$ stored in successive memory locations. The element $A[i]$ can be accessed in constant time, knowing its index i. This property is known as *direct addressing* and is the primary benefit of using an array.

The disadvantage of arrays is that, once their size is fixed, it cannot grow during the execution of a program. Insertions and deletions are also inconvenient. To insert an item in position i, we must make space for it by moving one place to the right, all the items currently in positions i and greater. Linked lists provide a way out of these problems. Although not a primitive data structure in many languages, they can be easily implemented using pointers. Linked lists are essentially chains of objects that can grow and shrink dynamically. This ability saves memory because, unlike for arrays, we do not need to reserve space in advance and so need not know their maximum length beforehand. Operations like inserting an element can be carried out much more efficiently with linked lists by manipulating a constant number of pointers. However, linked lists do not offer the facility of direct addressing. To locate a particular element, we must traverse the linked list, a process that takes time $\Theta(n)$ for an n-element list.

Stacks

This is a dynamic data structure that uses a LIFO (last in, first out) mechanism for the data. It supports two operations, *push* and *pop*. Push takes an item and inserts it at the top of the stack, whereas pop returns the item that is at the top of the stack. It is clear that only the top of the stack can be accessed by either operation and that the item returned by pop is the item that was last inserted by a push. There is also a query operation that can be used to test whether the stack is empty.

A stack that is expected to have no more than n elements during its existence can be implemented using an array $S[1..n]$. A variable *top* is used to store the index of the item at the top of the stack. If $top = 0$, the stack is empty; otherwise, it is represented by the elements $S[1], S[2], \ldots, S[top]$. All the operations can be done in constant time in this implementation. If we have no a priori knowledge of the maximum size of the stack, a linked list implementation can be used to save memory.

Queues

A queue is a dynamic data structure that uses a FIFO (first in, first out) policy. Again there are two operations, here called *insert* and *delete.* Insert adds an item to the tail of the queue. Delete returns the item at the head of the queue. As in a stack, a query operation that determines if the queue is empty is also provided.

An n-element queue can be implemented using an array $Q[0..n]$. Two variables *head* and *tail* keep track of where the queue begins and ends. *Head* stores the location of the first element in the queue, whereas *tail* stores the location of the first empty spot in the queue. The array is circular, that is, it wraps around. The queue is full if $tail + 1 \equiv head \bmod(n + 1)$, and it is empty if $tail = head$. To insert *item* into the queue, check whether it is full, and if not, store item in $Q[tail]$ and update *tail* to $(tail + 1) \bmod(n + 1)$. A similar procedure can do the deletion. Thus, both operations can be done in constant time.

Binary Rooted Trees

Trees are connected, acyclic graphs. A rooted tree is a tree with a distinguished node called the *root.* It is a leveled graph with the root at level 0. Edges only connect nodes at adjacent levels. If there is an edge connecting a node u at level i and v at level $i + 1$, u is called the *parent* of v and v is called a *child* of u. Two nodes with the same parent are called *siblings.* The *height* of a tree is the maximum level at which there is a node. A *leaf* is a node with no children. A nonleaf node is called an *internal* node. By acyclicity, every node has exactly one parent, except the root, which has none. A *binary tree* is a rooted tree where every node has at most two children. A *full* binary tree is a binary tree in which every internal node has exactly two children. A *complete* binary tree is a full binary tree with a block of rightmost leaves missing.

Binary Search Trees. Data structures can be studied from at least two different perspectives. One is a low-level perspective that deals with how the data structure is implemented. In this section, we adopt a high-level approach where we are less concerned with implementation issues and more interested in the operations the structure can support.

A *binary search tree* (BST) is a data structure organized as a rooted binary tree, which allows efficient storage and retrieval of data. Among the operations it supports are insertion, deletion, search (commonly called *dictionary operations*), and finding the maximum or the minimum among the items stored in it. Every node in a BST, stores a data item. It also has pointers *left, right,* and *parent* that point to its left child, right child, and parent, respectively. We use $key[u]$ to denote the key value of the item stored in node u. The tree is set up in such a way that, if w is a node in the tree, all the nodes in the left subtree of w have a key smaller than $key[w]$ and all the nodes in the right subtree of w have a key greater than $key[w]$. This is called the *BST-property.* Alternatively, we could store the data items in the leaf nodes and only the keys in the internal nodes. We now describe how the operations of insertion, deletion, and search are performed on a binary search tree. It will be seen that these operations can be done in time $O(h)$, where h is the height of the tree.

To *search* for a particular key, start at the root, and compare its key value with the key being searched. If the two keys don't agree, go left if the root has the higher key and right otherwise. Repeat this process at each node until either the key is located or the last node reached was a leaf. In the latter case, the item being searched for is not present in the tree. Clearly this procedure takes $O(h)$ time.

Inserting a given item with a key k is similar in spirit to searching. Starting from the root, go down the tree, comparing keys as before, until a leaf node is reached. Now insert the item in a new node to the left or right of the leaf node as appropriate. The time taken is again $O(h)$.

Deleting an item from a binary search tree is slightly more complicated. First, using the search procedure, we locate the item in the tree. Let v be the node where the item resides. If v is a leaf, we just delete it, that is the appropriate child pointer of its parent is set to the special value NIL. If v has only one child w, the child pointer of v's parent that used to point to v is now made to point to w. (If v is the root, the sole child of v is made the new root.) Call a node s the *successor* of v if the key of s is the next key immediately following the key of v in the sorted order of all the keys. It can be easily seen that when v has two children, s is the leftmost descendant in the right subtree of v. To delete v in this case, s is deleted and put in v's place. Clearly, deletion can be done in time $O(h)$.

Because all these primitive operations require time $O(h)$, it is important that the height of the tree be kept small. The minimum height a binary tree on n nodes can have is $\log n$, which is achieved in the case of a full binary tree. But in the worst case, it could be as bad as n, for instance, if the items are inserted in sorted order of keys. Variants of the basic binary search tree schema are used to keep the height from growing too much (e.g., red-black trees, AVL-trees, B-trees, and splay trees).

Heaps

A heap can be visualized as a complete binary tree with the property that if v is the parent of u, $key[v] > key[u]$. Thus, the node with the largest key value is the root of the heap. An n-node heap can be represented as an array $H[1..n]$ with $H[i]$ storing the contents of the ith node in the tree. The root is numbered 1 and the nodes at each level are numbered from left to right. With such a numbering the children of node i are numbered $2i$ and $2i + 1$ and the parent of node i is numbered $\lfloor i/2 \rfloor$. A heap can be used to implement a *priority queue,* which is a data structure that maintains a set of items, each with an associated key. It supports the operations of inserting an element into the set, finding (without deleting) the maximum element in the set, and extracting the maximum element from the set, all of which can be performed on an n-element heap in time $O(\log n)$. In fact, the find operation takes only constant time, and this is perhaps the main advantage of a heap over a BST. Among the algorithms that require the use of a priority queue are Prim's and Dijkstra's algorithms.

Hash Tables

Hash tables are a simple generalization of the notion of an array. The primary advantage of an array is that it provides direct addressing. In constant time, we can access an array element if we know its address (i.e., its index). A set of n items with (distinct) keys from a set $\{1, \ldots, m\}$, $m \geq n$, can be stored in an array A of size m, with an item whose key is

k stored in position $A[k]$. The basic operations—insert, delete, and search by key—can all be done in constant time. However, this method is too wasteful if m is very large. In such cases, we use a *hash function h,* which is a map from the universe of keys to a smaller set S. An item with key k is now stored in position $A[h(k)]$. This approach requires only $|S|$ storage.

There is the possibility that two keys might hash to the same value, creating a *collision.* A simple technique to handle collisions is *chaining,* where all items hashing to the same index are put in a linked list (i.e., $A[h(k)]$ is actually a list). The time to search for an item is, on the average, about $O(1 + n/|S|)$, assuming $h(k)$ can be computed in constant time. An approach that avoids collisions altogether, when $n \leq |S|$, is *probing.* This involves examining all locations in the array, one by one, until an empty location is found. The main idea here is that the sequence of locations visited depends on the key value being hashed.

COMPLEXITY THEORY

Broadly speaking, computational complexity theory is the study of the hardness of problems. It tries to classify problems according to their intrinsic difficulty, which means how efficient *any* algorithm for them can be. Efficiency of an algorithm is usually measured in terms of the time and the space that the algorithm uses. Time is the number of steps an algorithm takes, and space is the amount of memory it needs, both as functions of the size of the input to the algorithm. Often, we are interested in only the worst-case complexity of problems, that is, the amount of resources (space and time) used by any algorithm in the worst case.

Complexity theory mainly deals with *decision problems,* which are functions $f : \Sigma^* \rightarrow \{0, 1\}$. They are also called *languages* and commonly denoted by the letter L when looked upon as subsets of Σ^*. The function f defining L is then notated as χ_L. A string $x \in L$ iff $\chi_L(x) = 1$. In this section, problem means decision problem, unless explicitly stated otherwise.

An algorithm M is said to *decide* L if M when given input x returns 1 [in symbols, $M(x) = 1$] if $x \in L$ and 0 [in symbols, $M(x) = 0$) otherwise. The time complexity of a problem is said to be $T(n)$ if there is an algorithm to decide it that takes no more than $T(n)$ steps on any input of length n bits. Similarly, the space complexity of a problem is $S(n)$ if there is an algorithm for the problem that uses no more than space $S(n)$ for any n-bit input. This enables us to define complexity classes [e.g., TIME(n^3) is the class of problems of time complexity $O(n^3)$, SPACE(n^2) is the class of problems of space complexity $O(n^2)$ and so on]. Traditionally, problems of time complexity $O(n^k)$ for some constant k [which also implies a space complexity of $O(n^k)$] have been considered efficiently solvable. These collectively form the well-known complexity class P (for "polynomial").

Turing Machines

The most important computation model used in complexity theory is the *Turing machine* (TM) proposed by A. M. Turing in 1936. Many variations on the basic model are equivalent for computability and are polynomially equivalent for complexity measures. This means that anything that can be done in polynomial time in one model can also be done in polynomial time in any other model, although the polynomials bounding the running times may be different.

Here, we describe a specific model in which the TM M is a finite-state machine that consists of a semi-infinite tape, each cell of which can hold one symbol of the machine's *tape alphabet.* A *head* moves over this tape. The machine has a *finite control* that determines what action to take at every step. This depends on the *state* M is in and the symbol being scanned by the head. The action taken is to possibly overwrite the cell being scanned with another symbol and then to move the tape head left or right or not at all. At any point, the state, the contents of the tape, and the position of the head together constitute the current *configuration* of M. The configuration M starts with, is called its initial configuration. A computation can be visualized as a series of transitions from one configuration to another. M halts if and when it reaches one of its several *finish* states. If it halts in an *accepting* state (the corresponding configuration is called an accepting configuration), it is said to have accepted its input but to have rejected it otherwise. Note that a computation could go on forever. A Turing machine M accepts a language L, if M accepts exactly those inputs x that belong to L. If a language L is accepted by a machine that halts on *all* inputs, L is called a recursive language. In complexity theory, we deal with recursive languages only, and so we can assume that a TM always halts. A detailed presentation on Turing machines can be found in Refs. 8 and 9.

The Turing machine model is surprisingly powerful. Any computation that can be done with a program in any programming language can be done in this model and vice versa. This is the essential content of the *Church–Turing Thesis.* We do not go further into this here because it belongs to the subject of computability theory.

The definition of the class P can now be made more precise as follows. A language L belongs to P if and only if there is a positive integer k and a Turing machine M that runs for no more than n^k steps on any input of length n and accepts L. All problems considered from an algorithmic viewpoint in earlier sections are solvable in polynomial time. Strictly speaking, those problems are not decision problems, but suitable decision versions of them can be formulated and these lie in P.

Nondeterminism and NP-Completeness

Turing machines defined previously are of the *deterministic* variety. That is, at every step there is only one possible action that they can take. A very important extension of this is to allow the machine to *nondeterministically* choose among a set of possible actions. Machines with this capability are called nondeterministic Turing machines (NTMs). An NTM N *accepts* an input x if it can make the choices in such a way that it finally halts in an accepting state; it *rejects* x if no set of choices causes it to halt in an accepting state. In the following, we use NTIME and NSPACE for the time and space complexity classes defined by nondeterministic machines and DTIME and DSPACE for those defined by deterministic ones. The notation co-NSPACE [$f(n)$] is used for the set of languages whose complement belongs to NSPACE [$f(n)$].

The class corresponding to P in the nondeterministic model is NP, the set of languages L such that there exists an NTM

that runs in polynomial time and accepts L. NP is potentially a much bigger class than P, but nobody knows this for sure. The P $\stackrel{?}{=}$ NP question is the most famous open problem in computer science today.

An important notion in the study of NP is *NP-completeness.* An NP-complete problem L captures the complexity of the entire class NP, in that, it is the hardest problem in the class to solve in polynomial time. A polynomial-time algorithm for any problem in NP can be constructed if a polynomial-time algorithm for L exists, and it follows that P = NP. Because it is commonly believed that P $\neq$ NP, this means that a polynomial-time algorithm for L is unlikely. A problem L is defined to be *NP-complete* if $L \in$ NP and L is NP-hard, which means that for any $L' \in$ NP, there is a polynomial-time computable function f, called a *reduction,* from Σ^* to Σ^* so that, for every $x \in \Sigma^*$,

$$x \in L' \leftrightarrow f(x) \in L$$

Thus, if L has a polynomial-time algorithm A, then the membership of x in L' can be determined by first computing $f(x)$ and then using A to decide if $f(x) \in L$. More generally, the term NP-hard can be used for any computational problem, such that a polynomial-time algorithm for it can produce a polynomial-time algorithm for all problems in NP.

Hundreds of NP-complete problems are known today. The first language to be shown NP-complete (by Cook in Ref. 10) was the language of all satisfiable formulas of propositional logic in conjuctive normal form (CNF), commonly called SAT. Each of the conjuncts is called a clause. To see that this is in NP, notice that an NTM running in polynomial-time could nondeterministically guess an assignment to the variables of the formula, and then verify that this assignment satisfies the formula. Some other well-known NP-complete problems are

- *Traveling Salesperson Problem (TSP)*—Given a weighted, directed graph, and a positive number d, determine whether there exists a cycle in the graph that involves every vertex exactly once and such that the sum of the weights of the edges in it is at most d.
- *Vertex Cover*—Given a graph G and an integer n, determine if there is a subset S of vertices of size at most n, such that every edge of G is incident on some vertex in S.
- *Graph Coloring*—Given a graph G and integer k, determine if the vertices of the graph can be colored with k or fewer colors so that any pair of vertices connected by an edge are colored differently.

For much more on *NP*-completeness, refer to the book by Garey and Johnson (11).

Space Complexity

So far we have seen two important time complexity classes, P and NP. To define space complexity precisely, we assume that the Turing machine model has a read-write worktape separate from the input tape, which is read-only. The space used by a computation is defined as the number of cells used on the worktape. This definition enables us to consider classes of problems that require workspace less than n for inputs of length n. The most important among these classes are Logspace, written L, and its nondeterministic counterpart NL. L is the class of languages accepted by deterministic machines that use space $O(\log n)$. NL is the class of languages accepted by nondeterministic machines that use space $O(\log n)$ regardless of the nondeterministic choices made. DSPACE $[f(n)]$ and NSPACE $[f(n)]$ are defined analogously.

Somewhat more is known about the interaction of nondeterministic and deterministic space. *Savitch's Theorem* states that for any "proper" function $f(n) \geq \log n$, NSPACE$[f(n)]$ is contained in DSPACE$[f^2(n)]$. The properness of a function is a technical notion that we will omit here, but all functions commonly encountered in complexity theory are proper.

The *s-t connectivity problem* is the problem of determining, given a directed graph G and two distinguished nodes s and t whether there is a directed path from s to t. If G has n nodes, this can be accomplished by a nondeterministic machine using space $\Theta(\log n)$. The machine tries to incrementally guess a path from s to t, at every step merely writing down a next vertex and checking if it is connected to the current vertex by an edge. If at any time there is no edge connecting the two vertices, it halts and rejects. If it succeeds in reaching t, it accepts. Because it needs only two vertices at any time, and space can be reused, it uses $\Theta(\log n)$ space. Savitch showed that the $s - t$ connectivity problem can be solved *deterministically* in space $O(\log^2 n)$.

A computation of a nondeterministic machine M on an input x can be viewed as a graph whose vertices are configurations of M, there being an edge from u to v if M can go from u to v in one step. M accepts x if and only if some accepting configuration is reachable from the initial configuration. This means that to decide whether M accepts x is equivalent to solving the $s - t$ connectivity problem on this graph and Savitch's Theorem follows. Savitch's theorem implies that NL $\subseteq$ DSPACE$(\log^2 n)$. However, it is still open whether L [= DSPACE $(\log n)$] equals NL.

Another extremely significant result in space complexity is the *Immerman–Szelepcsényi Theorem,* which proves that for any proper complexity function $f(n) \geq \log n$, NSPACE$[f(n)]$ = co-NSPACE$[f(n)]$. The heart of this theorem is a result that given a graph G and a node x, the number of nodes reachable from x in G can be computed by an NL machine. Let $L \in$ NSPACE$[f(n)]$. Using the algorithm to count the number of nodes reachable from a given node, it can be shown that there exists an NSPACE$[f(n)]$ machine M' that recognizes $\overline{L}$, the complement of L.

A much bigger space complexity class is PSPACE, the class of languages that can be recognized by machines that use polynomial space. By Savitch's theorem it follows that PSPACE equals its nondeterministic version, i.e., PSPACE = NPSPACE. From known results linking space and time complexity classes, we get the tower of inclusions L $\subseteq$ NL $\subseteq$ P $\subseteq$ NP $\subseteq$ PSPACE. It is also known that L is different from PSPACE, so at least one of these inclusions is proper. It is not known which of them are; it is quite possible that all are. Beyond PSPACE lie EXP, deterministic exponential time and NEXP, nondeterministic exponential time. Inside P, there are a host of complexity classes defined by Boolean circuits. A detailed overview of all aspects of computational complexity is provided in (12).

RANDOMIZATION

Randomized Algorithms

One of the most important developments in the fields of algorithms and complexity theory is the use of randomization. A

randomized algorithm can *toss coins,* figuratively speaking, and depending on the outcome of the coin tosses, decide its next move. In reality, these algorithms use a source of *pseudo-randomness* that approximates perfect randomness. Some problems have randomized algorithms that are provably better than any deterministic algorithm for them, even one not yet discovered (e.g., a version of network routing). A very good exposition on various aspects of randomized algorithms is given in Ref. 13.

We have already seen an example of a randomized algorithm, namely, randomized quicksort where the choice of the pivot is made randomly. It can be shown, that with such a choice, the expected number of comparisons quicksort performs for any input is $O(n\log n)$, whereas the worst-case running time of quicksort for any deterministic strategy of choosing the pivot, e.g., picking the first element, is $\Theta(n^2)$.

Another simple example of a randomized algorithm is the one for the Min-Cut problem. Let G be a connected, undirected multigraph (i.e., a graph that may contain multiple edges between pairs of vertices). A *cut* $(S, V\backslash S)$ is a partition of the vertices of G into two subsets, S and $V\backslash S$, and the size of the cut is the number of edges that connect a vertex in S to a vertex in $V\backslash S$. Using the max-flow algorithm and the max-flow min-cut theorem, we can deterministically compute a minimum cut in G. But there is a simple randomized way to do this, with high probability. The algorithm repeatedly picks an edge uniformly at random and merges the two vertices that are its end-points. To merge the end-points of edge (x, y), replace vertices x and y by a single vertex w, replace all edges (x, u) and (y, v) by (w, u) and (w, v) for $u \neq y$ and $v \neq x$. The contraction of an edge decreases the number of vertices in the graph by 1. This process is repeated until only two vertices, v_1 and v_2, remain. Let S be the set of vertices of G that were involved in some contraction among those that eventually produced v_1. Then, $(S, V\backslash S)$ is the cut produced by the algorithm.

Now we prove that, with nonnegligible probability, this is a min-cut for G. It can be seen that any cut of an intermediate graph is also a cut of G but may not be a minimum cut for it. Let C be some minimum cut of G. We will compute the probability that this algorithm outputs C (which happens if and only if no edge of C is contracted at any stage). Clearly, this is a lower bound on the probability of correctness of the algorithm. If C has k edges, then every vertex v of G has degree at least k [otherwise, $(\{v\}, V\{v\})$ would be a smaller cut], and so G has at least $kn/2$ edges. Let G_i be the graph produced after i contractions. G_i has $n - i$ vertices, and if no edge of C has been contracted in the first i contractions, a minimum cut has at least k edges in it. Therefore, G_i has at least $[k(n - i)]/2$ edges. Thus, the probability that an edge of C is not contracted in the $(i + 1)$st contraction, given that no edge of C has been contracted in the first i contractions, is $1 - 1/(n - i)/2 = 1 - 2/(n - i)$. Consequently, the probability that after $n - 2$ contractions all edges of C remain intact is

$$\prod_{j=0}^{n-3}\left(1-\frac{2}{n-j}\right)=\frac{2}{n(n-1)}>\frac{2}{n^2}$$

Therefore, the error probability of this algorithm is at most $(1 - 2/n^2)$. By doing sufficiently many independent tries, this probability can be made exponentially small.

Suppose now that we are asked to find a maximum cut in a graph, that is to find a subset S of the vertex set V so that the number of edges between vertices in S and $V\backslash S$ is maximized. On the surface, this might look similar to Min-Cut, but it is in fact NP-hard to solve optimally. The following simple randomized algorithm achieves a good approximation, in that, the expected number of edges in the cut it finds is $m/2$. It starts with two empty sets A and B and, for each vertex in turn, puts it into one of the two sets with equal probability. Clearly, the expected number of edges in the cut defined by A and B is $m/2$. We can then use standard *tail inequalities* like Markov's inequality to prove that, with high probability, the size of the cut produced is at least $m/4$, say. Because a maximum cut can have at most m edges in it, this randomized approximation algorithm achieves an approximation ratio of 4 with high probability.

Randomized Complexity

To study randomization from a complexity point of view, an extension is made to the deterministic Turing machine model. *Probabilistic* Turing machines (PTMs) have an extra tape that has perfectly random bits written on it. The machine can read this tape when needed and use the bit read to determine its next move. Reading a bit from the tape is like tossing a coin. Probabilistic machines have more than one definition of acceptance and rejection, and this fact gives rise to three different complexity classes corresponding to the class P of deterministic computation. The class RP (Randomized Polynomial time) consists of those languages L for which there exists a polynomial-time PTM M such that

$$x \in L \Rightarrow \Pr(M \text{ accepts } x) \geq \frac{1}{2} \quad \text{and}$$
$$x \notin L \Rightarrow \Pr(M \text{ accepts } x) = 0$$

Here, the probabilities are over the coin tosses of the machine. The probability of acceptance when $x \in L$ can be made $1/p(n)$, for any polynomial $p(n)$, without changing the definition of the class. It can also be boosted to $1 - 2^{-n}$ by repeated tries. RP resembles NP in the sense that, if $x \in L$, there are *witnesses* for this fact, namely the sequences of coin tosses in the case of RP, and the nondeterministic choices in the case of NP, that cause the machine to accept. However, RP demands that the witnesses be abundant. Trivially, P $\subseteq$ RP $\subseteq$ NP. The complement of RP is called co-RP. The class ZPP, consisting of languages that can be recognized with *zero error* by a PTM running in expected polynomial time, is equal to RP $\cap$ co-RP.

A language L is in the class BPP (Bounded-error Probabilistic Polynomial time) if there is a polynomial-time PTM M such that

$$x \in L \Rightarrow \Pr(M \text{ accepts } x) \geq \frac{3}{4} \quad \text{and}$$
$$x \notin L \Rightarrow \Pr(M \text{ accepts } x) \leq \frac{1}{4}$$

These probabilities can be replaced with $1/2 + 1/[p(n)]$ and $1/2 - 1/[p(n)]$ for any polynomial $p(n)$. Thus, a BPP machine may err in both directions, but the error is bounded away from 1/2 by at least an inverse polynomial amount. The error probability can be reduced by doing many independent tries and taking the majority outcome. By the symmetry of the

definition, BPP = co-BPP. The relationship of NP and BPP is still open.

A language L is in the class PP (Probabilistic Polynomial time) if there is a polynomial-time PTM M such that,

$$x \in L \Rightarrow \Pr(M \text{ accepts } x) > \frac{1}{2} \quad \text{and}$$
$$x \notin L \Rightarrow \Pr(M \text{ accepts } x) \leq \frac{1}{2}$$

The error probabilities in either direction may be exponentially close to 1/2, so polynomially many independent tries may fail to decrease the error substantially. It can be shown that PP = co-PP and that NP $\subseteq$ PP. Also, RP $\subseteq$ BPP $\subseteq$ PP.

Derandomization

Consider the *hash* function $h_{a,b}:\{0, 1\}^n \rightarrow \{0, 1\}^m$, $m \leq n$, computed by taking the first m bits of $ax + b$, where a, $b \in \{0, 1\}^n$, and the addition and multiplication are done in the field GF[2^n], the elements of which can be put in 1-1 correspondence with $\{0, 1\}^n$. Such a hash function can be randomly picked by picking a, b randomly from $\{0, 1\}^n$, using $2n$ random bits. For any $x_1, x_2 \in \{0, 1\}^n$, $y_1, y_2 \in \{0, 1\}^m$, $x_1 \neq x_2$, $\Pr_{a,b}(h_{a,b}(x_1) = y_1$ and $h_{a,b}(x_2) = y_2) = 1/2^{2m}$. Because of this property, $h_{a,b}$ are called two-universal hash functions. Let H_x be the random variable that takes the value $h_{a,b}(x)$ where a,b are chosen randomly. Then, this property means that H_{x_i} and H_{x_j} are independent random variables when $x_i \neq x_j$; that is, the collection $\{H_x | x \in \{0, 1\}^n\}$ is a set of *pairwise independent* random variables.

Hash functions can be used to make certain randomized algorithms deterministic, a process called *derandomization.* Consider the randomized algorithm for the Max-Cut problem on an n-vertex graph. It uses a total of n random bits, one per vertex. Given an edge (x, y), the probability that it is in the cut is 1/2 because x and y are uniformly and independently put in one of the sets A and B. The only property of the random assignment used is that, for any distinct x and y, the assignment of a set is done independently. The same effect can be achieved by making the assignments only in a pairwise independent manner. As seen previously, a family of hash functions mapping $\{0, 1\}^{\lceil \log n \rceil}$ to $\{0, 1\}$ can be used for this purpose. A random assignment would be choosing a and b randomly and mapping vertex x to $h_{a,b}(x)$ (where 0 means x is put in A and 1 means it is put in B). However, because the total number of hash functions in this family is only $\Theta(n^2)$, we can exhaustively try them all in polynomial time. Because the expected value of a cut is $m/2$, there exists one with at least so many edges, and we are assured of finding it. This derandomizes the algorithm and yields a deterministic polynomial-time algorithm, which gives a reasonably good approximate solution to an NP-hard problem.

APPROXIMATION AND INAPPROXIMABILITY

Approximation Algorithms

In the absence of a proof settling the P $\stackrel{?}{=}$ NP question, researchers have turned their attention to finding polynomial-time algorithms that provide approximate solutions for hard problems. The problems to which such algorithms generally apply are optimization problems. Many of these have been proven to be NP-hard and are often closely related to NP-complete decision problems. For example, MAXSAT is the problem of finding, for a given logical formula in CNF, an assignment to its variables that maximizes the number of satisfied clauses. Clearly, if this problem can be solved in polynomial time, so can SAT. We also saw previously an approximation algorithm for the NP-hard problem Max-Cut. For more on this subject see Ref. 14.

To analyze the performance of an approximation algorithm on an instance of an optimization problem, we associate a value with each solution to that instance. Given an instance I, the task is to find a solution S that maximizes/minimizes this value $v(S)$. Let $OPT(I)$ denote the optimal value of a solution to instance I of a minimization (maximization) problem and $A(I)$ be the value of the solution produced by an approximation algorithm A. The performance of A is measured, most often, as the ratio $[A(I)/OPT(I)][(OPT(I)/A(I))]$. Clearly, this number, called the *approximation ratio,* is at least 1.

Multiprocessor Scheduling. Earlier, we saw a greedy algorithm that optimally scheduled unit-time jobs with deadlines on a single processor, so that the penalty incurred for jobs finishing after their deadline was minimized. Here is a slightly different version of this problem. There are n jobs $J_1, \ldots, J_n$ with runtimes $t_1, \ldots, t_n$. These are to be assigned to m identical processors so that the total runtime of the system is minimized. This is defined as the maximum that any processor needs to run before all the jobs are completed. This problem is known to be NP-hard even in the case of two processors.

There is a simple greedy approximation algorithm for this problem, due to Graham, that achieves an approximation ratio at most $2 - 1/m$. This considers the jobs in turn, assigning the current job to the least-loaded processor. The load on a processor is the total running time of the jobs scheduled on it so far. Let $A(I) = T$ be the total runtime of the system in the solution returned by the algorithm. Let P be a processor that has a total load of T, and let t be the running time of the last job scheduled on P. Because this job was given to P, each of the other processors must have a load of at least $T - t$. Thus, the sum of the runtimes of all the jobs is at least $T + (m - 1)(T - t)$. Because any solution, including the optimal, must have a value not less than the average running time of the jobs,

$$\begin{aligned} OPT(I) &\geq \frac{T + (m-1)(T-t)}{m} \\ &= T - \left(1 - \frac{1}{m}\right) t \\ &\geq A(I) - \left(1 - \frac{1}{m}\right) OPT(I) \end{aligned}$$

The last inequality uses the trivial fact that $OPT(I) \geq t$ and the performance bound on the algorithm follows. It can also be shown that this bound is tight, that is, there exists an instance for which this bound is actually achieved.

Polynomial-Time Approximation Schemes

For some problems, there is a family of algorithms $\{A_\epsilon | \epsilon > 0\}$ such that, A_ϵ achieves an approximation ratio $\leq 1 + \epsilon$ but has

a running time that increases as ϵ decreases but that is still polynomial in the input size (but not necessarily in $1/\epsilon$). This means we can achieve as good an approximation as we want, but at the expense of running time. Such a family of algorithms is called a *polynomial-time approximation scheme* (PTAS). Such a scheme exists for the multiprocessor scheduling problem [again due to Graham (15)], but the running time of A_ϵ is exponential in $1/\epsilon$. Thus, we cannot use it to get a really good approximation.

Fully Polynomial-Time Approximation Schemes

A *fully polynomial-time approximation scheme* (FPTAS) is a PTAS where the running time of A_ϵ is polynomial in both the input size and $1/\epsilon$. This is the best we can achieve, short of solving the problem exactly in polynomial time. However, not many NP-hard problems are known to have an FPTAS. Knapsack is one of the few exceptions. This shows that NP-hard problems may not behave alike when it comes to finding approximate solutions. An instance of knapsack consists of n items with weights $w_1, \ldots, w_n$ and values $p_1, \ldots, p_n$, respectively. Let the maximum capacity of the knapsack be M. The FPTAS for knapsack (16) uses a pseudo-polynomial-time algorithm whose running time is polynomial in n, $P = \max_{i=1}^{n} p_i$ and $\log M$. Note that this algorithm is different from the one given in the section on Dynamic Programming whose running time was polynomial in n and M.

Now we define an algorithm B_k, where k is any number. It first constructs a new input instance by scaling the values by a factor $1/k$ (i.e., the new values are $\lfloor p_1/k \rfloor, \ldots, \lfloor p_n/k \rfloor$, respectively) and applies the pseudo-polynomial-time algorithm to this instance. Then, $A_\epsilon = B_k$, where $k = P/[(1 + \epsilon^{-1})n]$. Clearly, the running time of A_ϵ is polynomial both in the input size and in $1/\epsilon$. It can also be shown that the approximation ratio of A_ϵ is at most $1 + \epsilon$.

The PCP Theorem and Inapproximability

Let $L \in$ NP. Then, there is a nondeterministic polynomial-time machine M that accepts L. If $x \in L$, there is a sequence of nondeterministic choices of M that results in x being accepted. These nondeterministic choices "prove" that $x \in L$. Given these choices, the membership of x in L can be checked in polynomial time. Call this sequence of choices an *accepting path* of M on input x. Let V be a polynomial-time machine that accepts the following language of pairs:

$$L' = \{(x, y) | y \text{ encodes an accepting path of } M \text{ on input } x\}$$

V is called a *verifier* for L. If $x \in L$, there is a y such that V accepts (x, y), and if $x \notin L$, there is no such y. Clearly, L is in NP if and only if it has a polynomial-time verifier.

A new restricted verifier model was defined by Arora and Safra (17). In this definition, a $[r(n), q(n)]$-restricted verifier P for a language L is given a pair (x, y) as input, where y is supposed to be a polynomial-length proof that $x \in L$. The verifier has access to a random bit string R of $r(n)$ bits. Using x and R, it computes $q(n)$ integers $a_1, \ldots, a_{q(n)}$, each at most $|y|$. The bits of y in positions a_i, $1 \le i \le q(n)$, are then written on a tape of the verifier. In polynomial time, V then accepts or rejects its input, without using the other bits of y. The following holds: if $x \in L$, then there is a y that causes V to accept (x, y) with probability 1, and if $x \notin L$, no y can make V accept (x, y) with a probability greater than 1/2. With this model, it was proved by Arora et al. (18) that NP is exactly the class of languages that have a $(c \log n, q)$-restricted verifier for some constants c and q. As a corollary, they showed that there is a constant $\epsilon > 0$ such that approximating MAX3-SAT (which is a restriction of MAXSAT where each clause has at most three literals) within a factor $1 + \epsilon$ is NP-hard. In other words, if MAX3SAT has a PTAS, then P = NP. The theorem characterizing NP in terms of the existence of a $(c \log n, q)$-restricted verifier is called the PCP theorem. It has been instrumental in producing inapproximability results for a wide variety of optimization problems.

BIBLIOGRAPHY

1. D. E. Knuth, *The Art of Computer Programming,* vols. 1–3, Reading, MA: Addison-Wesley, 1968.
2. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms,* Reading, MA: Addison-Wesley, 1974.
3. T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms,* Cambridge, MA: MIT Press, 1990.
4. C. A. R. Hoare, Quicksort, *Comput. J.,* **5** (1): 10–15, 1962.
5. V. Strassen, Gaussian elimination is not optimal, *Numerische Mathematik,* **14** (3): 354–356, 1969.
6. D. Coppersmith and S. Winograd, Matrix multiplication via arithmetic progressions, *J. Symbolic Comput.,* **9**: 251–280, 1990.
7. R. W. Floyd, Algorithm 97 SHORTEST PATH, *Commun. ACM,* **5** (6): 345, 1962.
8. J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation,* Reading, MA: Addison-Wesley, 1979.
9. H. R. Lewis and C. H. Papadimitriou, *Elements of the Theory of Computation,* Upper Saddle River, NJ: Prentice-Hall, 1997.
10. S. Cook, The complexity of theorem-proving procedures, *Proc. 3rd ACM Symp. Theory of Computat.,* 1971, pp. 151–158.
11. M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-completeness,* San Francisco: Freeman, 1979.
12. C. H. Papadimitriou, *Computational Complexity,* Reading, MA: Addison-Wesley, 1994.
13. R. Motwani and P. Raghavan, *Randomized Algorithms,* Cambridge: Cambridge Univ. Press, 1995.
14. D. Hochbaum, ed., *Approximation Algorithms for NP-hard Problems,* Boston: PWS Publishing Company, 1997.
15. R. L. Graham, Bounds for certain multiprocessing anomalies, *Bell Syst. Tech. J.,* **45**: 1563–1581, 1966.
16. O. Ibarra and C. E. Kim, Fast approximation algorithms for the knapsack and sum of subset problems, *J. ACM,* **22** (4): 463–468, 1975.
17. S. Arora and S. Safra, Probabilistic checking of proofs; a new characterization of NP, *J. ACM,* **45** (1): 70–122, 1998. Preliminary version appeared in *Proc. 33rd IEEE Symp. Foundations Comput. Sci., FOCS,* 1992, pp. 2–13.
18. S. Arora et al., Proof verification and hardness of approximation problems, *Proc. 33rd IEEE Symp. Foundations Comput. Sci. FOCS,* 1992, pp. 14–23.

JIN-YI CAI
AJAY NERURKAR
SUNY Buffalo

ALIGNMENT CHART. See NOMOGRAMS.