1. Introduction

J. Michael Steele,[1] University of Pennsylvania
and
David Aldous, University of California at Berkeley

The theory of algorithms has undergone an extraordinarily vigorous development over the last 20 years, and probability theory has emerged as one of its most vital partners. University courses, research monographs, and specialized journals have been developed to serve this partnership, yet there is still a need to communicate the progress in this area to the wider audience of computer scientists, mathematicians, and individuals who have a stake in the contributions that mathematical research can make to technology.

The main purpose of this report is to give an understanding of the power that comes from applying probability in the theory of algorithms, but an equally essential aim is to point out the *variety* of ways in which probability plays a role. One useful step in understanding this variety comes from making a clear distinction between the subject of *probabilistic algorithms* and the subject of *probabilistic analysis of algorithms.* Confusion sometimes arises over what methods are properly called "probabilistic (or randomized) algorithms"—and indeed there are some gray areas. Still, if one does not press for too fine a point, there are considerable organizational and conceptual benefits in drawing the distinction between probabilistic algorithms and the probabilistic analysis of a (possibly deterministic) algorithm.

One common distinction is that probabilistic algorithms, unlike deterministic ones, make random choices when computing. They are commonly referred to as "coin-flipping algorithms." Such algorithms will produce (possibly) different results for the same problem when posed in different circumstances. On the other hand, the probabilistic analysis of an algorithm

[1] J. Michael Steele's research was supported in part by NSF grant DMS88-12868, AFOSR grant 89-0301, ARO grant DAAL03-89-G-0092, and NSA grant MDA-904-H-2034.

incorporates randomness into the *data* processed by an algorithm. Properly speaking, we are considering the pair *(algorithm, problem instance)* and probabilistically exploring the algorithm behavior over a large variety of problem instances. Typically, the analyst can make statements about the probability of selecting a particular instance, or focus attention on the distribution of suitable variables that describe the problem instance. The task is then to relate the algorithm performance to these variables.

This introductory chapter provides several simple illustrations of the distinction between the design of probabilistic algorithms and the probabilistic analysis of algorithms. An excellent examples-oriented survey of probabilistic algorithms is that of Karp (1991).

1.1 Probabilistic Algorithms

1.1.1 Everyday Examples

Before developing more serious examples that require detailed mathematical description, it seems useful to provide a couple of everyday analogies. Mathematics often speaks best for itself, and one should not read too much into analogies, but because there are important uses of probabilistic algorithms that can be explained without any technical prerequisites, it seems appropriate to look at them early on.

We have all had the experience of walking along a narrow path and encountering someone who is approaching on the same side of the path. As the individuals draw closer, one moves to the other side of the path in order to let the other person pass, but often the other person does the same simultaneously. This little shuffle continues one or two more times before the two people end up on opposite sides of the path and can pass each other.

Although people resolve such difficulties with little more cost than a moment's embarrassment, the analogous situation is more serious when packets of information end up vying for the use of the same place at the same time in a communication channel. When one tries to program machines to avoid such deadlocks, there are decided drawbacks to most deterministic rules. The dogged consistency of machines is such that the shuffling from side to side can go on unabated until an outside action halts the dance. This is an unacceptable state of affairs that one ought to be able to avoid through thoughtful design.

One theme of this report is that a natural course of action when one is faced by the disadvantages of purely deterministic rules is to introduce some

sort of randomness into the protocol. Each process could continue to follow its own **randomized** rule without regard for the rule being followed by the other process, and the eternal dance would be avoided. This simple idea is at the heart of many of the communication protocols in use in the world today.

The larger idea onto which this flavor of application can be attached might be called "randomization to avoid special coincidences." Mathematicians can find analogs of this phenomenon in many other fields, and, as more technical examples will illustrate later, there are more profound consequences to the idea than are done justice to by this first quick, everyday example.

Our second everyday example involves opinion polls. These are so widely discussed and so well understood that it is not easy to evoke a proper awareness that there is actually quite a bit of magic in the technology—enough so that there is still cutting-edge research in computer science related to the subject. Still, in order not to get ahead of our story, let us first consider a question that is typical of the sort addressed by Gallup and other famous pollsters: What percentage of the U.S. voting population feels that the recent pay raise voted for itself by the U.S. Senate was well deserved? A fact that was completely unavailable to the founding fathers, but which is much a part of modern political life, is that one can obtain a useful answer to the question at a cost that is a small fraction of the cost of conducting a census of the whole voting populace.

One point that deserves to be underscored in this example is that the precise question that has been posed has nothing a priori to do with probability. If we are totally faithful to the precise phrasing of the question, we have to agree that it is one that can be answered only by a complete census as long as one insists on an *exact* answer. Still, a useful answer need not be perfectly exact, and the full complement of political insight is likely to be gained by knowing the answer to within plus or minus two percentage points. This fact is easily seen and accepted in this context, yet when one reviews the theory of algorithms one finds that tremendous effort is often expended to get exact answers where approximate ones might serve just as well. Thus, willingness to accept approximation seems to be one door for probability to enter. More surprising mathematical examples to be considered below will show that this is not the only door.

Still, in the context of our polling question, an approximation clearly serves the policy purpose just as well as a complete census. If one selects a random sample of about 2,500 from the target population and asks, "Do you

agree that the pay raise that the U.S. Senate gave itself is well deserved?", the percentage of individuals who say yes provides an estimate of the percentage of the whole populace that would answer in the same way. It is a consequence of elementary probability theory that our estimate will be correct to within ± 2 percent on better than 95 percent of the occasions on which such a sampling experiment is conducted.

This is a familiar tale, even if told a bit more precisely than usual, and the reader may legitimately ask why this old tale is being told here. The retelling is not occasioned to celebrate polling, which certainly draws enough of its own attention. The point is not even to recall the often missed subtlety that the required sample size depends only on the precision required, and not on the size of the population about which the inference is being made; though it is interesting that the senators from Rhode Island find it as costly to obtain information about the voting populace of that state as it is for the National Committee to get the corresponding information about the nation as whole.

The point is rather to see that the sampling techniques of political pollsters are actually probabilistic algorithms; one uses exogenous randomization to obtain an approximation to a problem that would be prohibitively expensive to answer exactly. The key point for us is not the approximation issue. Approximation is a common but not inevitable feature of probabilistic algorithms. The key point is that one pours in randomness that was not there to begin with. In the pollster's case the randomness that was added was that used in making the random selection. What was purchased at the price of this extra complication was the applicability of the probability theory that enabled us to quantify the quality of our solution. These are features that one finds throughout the theory of probabilistic algorithms.

To be sure, there are differences in flavor between the pollsters' techniques and those applied in computer science. The pollsters' methods are applied in a social rather than a computational context, and, perhaps as a result, the process looks unsophisticated, or even straightforward. Individuals will sometimes disagree, but it seems useful to hold that the pollsters' techniques are just as much a probabilistic algorithm as any being studied in computer science. The difference of context does not alter the logical structure, and the perceived difference in sophistication comes in good measure from the fact that our example did not go into any subtleties such as how one might really draw the sample and whether some trickier sampling scheme might do better than a uniform random sample.

This pollster problem completes the second of our two everyday exam-

ples. Both were illustrations of randomization applied in algorithms. In one case the randomization offered escape from the "special situation" traps into which deterministic processes can fall. In the second case, we gained great savings of cost at the price of accepting an approximation, and we were able to quantify the quality of our approximation because we were able to add just the kind of extra randomness to which a compelling theory could be applied.

1.1.2 Hashing

Some of the earliest examples of probabilistic algorithms were invented in the context of storage algorithms, and one of most influential of the ideas to be developed to speed up the access of stored information is **hashing**. This device is used in many important computational contexts and even lives at the heart of some computer operating systems. Still, it can be described in the style of an everyday example.

Consider an office where 50 employees receive mail. The thoughtful employer typically provides a set of somewhat more than 50 mailboxes so each employee can be assigned one. Conceivably, the assignment of physical mailboxes to individuals could be done in lots of different ways. But the conventional method is so familiar that most people would call it obvious. The tradition is to physically label mailboxes with employees' names, in alphabetical order (this suggests an ordering of the boxes, usually left-to-right within rows and top-to-bottom across rows, but we digress). To see that this arrangement is not the only conceivable one, consider instead a hotel that has 50 rooms and receives letters addressed to its guests. The hotel could use the arrangement above, but that would be silly because guests arrive and leave so rapidly that the staff would waste time moving name labels. Instead, hotels label their mailboxes by room number, and when letters arrive they look up the addressee's room number. Hashing abstracts the hotel's procedure. Suppose we have n storage locations numbered $1, 2, \ldots, n$ and wish to store $m \leq n$ items, where each item has (say) a name in plain English. We specify some **hash function** $f : \{\text{all names}\} \rightarrow \{1, 2, \ldots, n\}$. For each item in turn we compute $f(\text{name}) = i$, say, and attempt to store the item in location i. We then need a rule to tell us what to do if location i is already occupied; the simplest rule, **hashing with linear probing**, is to look at locations $i + 1$, $i + 2$, ... until an empty location is found and store the item there. Note that once all the items are stored, they are simple to locate. Given the name, compute $f(\text{name}) = i$, say, and look at locations

$i, i+1, \ldots$ until the item is found or an empty location is found, in which case the item cannot be present.

This algorithm is appropriate if storage is cheap and we desire to locate items very quickly. Under plausible assumptions, the mean number of locations checked in searching for an item (averaged over items) is approximately a function of the density m/n only; that is to say, it does not depend on the total number of items to be stored. The conceptual idea is to choose a hash function that "scrambles" the name so that the resulting value can be regarded as random and uniform on $1, 2, \ldots, n$. Thus, the mathematical analysis of the performance of hashing with linear probing can be done under the assumption that the hashed values of the m items are independent uniform random variables. To justify this assumption for a particular hash function would consequently involve the same issues as justifying a pseudorandom number generator.

Though it is not difficult to analyze the scheme just described (see Knuth, 1973, §6.4), the main features of this scheme can be seen from a trivial-to-analyze but less practical variant in which one specifies a sequence of hash functions and resolves collisions by applying the next hash function. Here, after inserting the $(i+1)$st item, one will need to search a mean number $\frac{n}{n-i}$ of locations until finding an empty location. The same number is needed to search for the item, so the mean search length (over all m items) is

$$\frac{1}{m}\sum_{i=1}^{m}\frac{n}{n-i+1} \sim \frac{n}{m}\log\left(\frac{1}{1-\frac{n}{m}}\right).$$

The linear probing scheme turns out to require more searching because the occupied locations tend to cluster, and its analysis is more taxing. Still, both linear probing and multiple hashing share the fundamental qualitative feature of depending only on the ratio m/n.

1.1.3 Geometry

One of the areas in which probabilistic algorithms have recently proved to be exceptionally powerful is computational geometry. The body of this report does not pursue these applications except somewhat in passing; the reader is referred to Seidel (1992), which recounts several applications and gives references to further work. To give some sense of the way that randomization can come into play in computational geometry, we look briefly at one of the examples treated by Seidel.

The **Delaunay triangulation** of a set S of n points in the plane is the graph with vertex set S that puts an edge between points x and y if and only if there is a disc with x and y on its boundary that contains no other points of S. Algorithms for computing the Delaunay triangulation have been well studied. There is a known deterministic algorithm for computing the tessellation in $O(n \log n)$ steps, and in the worst case this order cannot be improved.

For the special case where S consists of the vertices of a convex polygon, a simple probabilistic algorithm is available. The key idea is to let s_n be one of the points in S and let $S_{n-1} = S \setminus \{s_n\}$. In this special case S_{n-1} still forms the vertices of a convex polygon. Suppose we are given the tessellation of S_{n-1} and want to compute the tessellation of S. There is an algorithm for doing this (see Seidel, 1992) which we shall not describe here: if we are lucky, we may only have to add two edges to s_n, but in general we must delete some existing edges and retriangulate.

Obviously, given such an algorithm we can order the points of S arbitrarily as $s_1, s_2, \ldots, s_n$ and apply the algorithm to update the tessellation as each new point is added. The efficiency of this process will clearly depend on the order of the s_i, but it is not easy to envisage an optimal order. The key idea is to proceed as in Quicksort and put S in random order. Then the final update can be rephrased as follows: pick s_n at random from S, suppose we have constructed the triangulation on $S \setminus \{s_n\}$, and apply the update algorithm. A geometrical argument shows that, for each choice of s_n, the number of steps needed is on the order of the degree of s_n in the tessellation on S. But it is easy to see that the average degree of the vertices in a Delaunay tessellation is less than 4, so because s_n is random we conclude that the expected number of steps in the final update is $O(1)$, and hence the total number of steps is $O(n)$.

1.1.4 Competitive Analysis

Traditionally, algorithms have been compared using **worst-case** analysis or **average-case** analysis, in other words by considering the worst possible input or by putting some probability measure (perhaps chosen for mathematical simplicity rather than realism) on inputs. Recently, attention has been paid to a different mode of analysis called **competitive analysis**. In particular, this has been applied to on-line algorithms (defined in Chapter 7). The idea is to compare a given on-line algorithm with the optimal off-line algorithm and define the **competitiveness coefficient** to be the

smallest c such that, for some b,

$$\{\textit{Cost using on-line algorithm}\} \leq c \times \{\textit{Cost using optimal algorithm}\} + b,$$

for all inputs. One specific problem where this method of analysis has been used is the **caching problem.**[2] Imagine a cache (fast memory) that can hold n items and a slower external memory that we can model as having unlimited capacity. A requested item that is not in the cache must be retrieved from the main memory at unit cost. To use these memories efficiently, an algorithm is needed to specify when items are switched between cache and main memory. Fiat et al. (1991) invented a simple probabilistic algorithm, the **marker algorithm.** Initially, let none of the n items in the cache be marked. If an item in the cache is requested, then it is marked. If an item outside the cache is requested, then an unmarked item in the cache is chosen uniformly at random and switched with the requested item, which is then marked. Eventually, all items in the cache are marked, at which time remove all marks and start again. Fiat et al. proved this algorithm has competitiveness coefficient at most $2\sum_{i=1}^{n} 1/i \sim 2\log n$. Note the simplicity of the randomization: it is hard to imagine a simple deterministic algorithm performing so well.

1.1.5 Random Constructions

A surprising connection between mathematical probability and the theory of algorithms is the idea of proving mathematical results about random objects by studying the behavior of an algorithm for constructing the random object. We illustrate this with the classical topic of random permutations. Suppose we want to generate, inductively on n, the uniform random permutation of n objects into n positions. Picturing the objects as physical files in a file cabinet, the natural update algorithm for adding the nth object is: pick p uniformly on $1, \ldots, p$, move objects at positions $i = p, p+1, \ldots, n$ to positions $i+1$, and put object n at position p. In a computer the natural update algorithm is: put object n at position n, pick p uniformly on $1, \ldots, p$, and switch the objects at positions n and p.

Though the latter algorithm is essentially the standard optimal algorithm for computer simulation of a random permutation, here is a more

[2] Our treatment follows Raghavan (1990).

complicated algorithm that turns out to be useful for mathematical purposes:

The Chinese restaurant process. Mathematicians at a conference agree to dine at a Chinese restaurant.[3] They arrive separately, and so the ith arrival has i choices of where to sit: next (clockwise, say) to one of the $i-1$ mathematicians already present, or at an unoccupied table. Suppose the choice is random and uniform. After n arrivals, the pattern of mathematicians (labeled by arrival order) at tables can be regarded as the cycle representation of a permutation, and it is easy to see that the sequential uniform random choices create a uniform random permutation.

Suppose we want to study the random number C_n of cycles in a random permutation. In terms of the Chinese restaurant algorithm, C_n is just the number of occupied tables. But the ith arrival has chance $1/i$ of starting a new table, so without calculation we can see that the expectation of C_n is $\sum_{i=1}^{n} 1/i$. (Classical probability theory gives a normal limit distribution.)

1.1.6 Testing Equality and Faith

Suppose one has a polynomial in n variables $p(x_1, x_2, \ldots, x_n)$ that represents a possible identity. Naturally, it is desirable to know if the identity is valid for all $x_1, x_2, \ldots, x_n$. For example, suppose one did not know about the modulus identity for quaternions but had somehow come to suspect that, for $\alpha, \beta, \gamma, \delta, \alpha', \beta', \gamma', \delta' \in \mathrm{R}$,

$$\begin{aligned}(\alpha^2+\beta^2+\gamma^2+\delta^2)(\alpha'^2+\beta'^2+\gamma'^2+\delta'^2) = \\ (\alpha\alpha'-\beta\beta'-\gamma\gamma'-\delta\delta')^2+(\alpha\beta'+\beta\alpha'+\gamma\delta'-\delta\gamma')^2 \\ +(\alpha\gamma'+\gamma\alpha'+\delta\beta'-\beta\delta')^2+(\alpha\delta'+\delta\alpha'+\beta\gamma'-\gamma\beta')^2\,.\end{aligned}$$

Is there a quick and painless way to decide if this conjectured identity is indeed valid for all values of the arguments?

There is way to answer this question that suggests itself rather naturally from the probabilistic point of view and that also serves to point out some subtleties in the algorithmic uses of probability. The idea can also be used to test one's faith in probability theory.

Because nonzero polynomials in n variables cannot vanish on a set of positive measure, the brave probabilist can check identity almost trivially.

[3] Chinese restaurants often have large circular tables.

All one has to do is choose a point $(\alpha, \beta, \gamma, \delta, \alpha', \beta', \gamma', \delta')$ at random from $[0,1]^8$ to see if the point satisfies our would-be identity. If the expression in question is not an identity, the two sides will evaluate to different values with probability one. Thus, without any call on sampling theory or repeated trials, one can answer the point in question with probabilistic certainty and very little arithmetic.

Still, the answer is too easy not to provoke doubts. Our ability to choose numbers at random from sets like $[0,1]^8$ comes belatedly into question, and we are reminded that computers do not actually operate on real numbers. There is clearly some need to discretize the probabilists' suggestion, and there are important details to be resolved. Our first help is to be found in a discrete version of our statement that a nontrivial polynomial in n variables cannot vanish on a set of positive measure in $\mathbf{R}^n$.

Lemma 1 *Suppose $p(x_1, x_2, \ldots, x_n)$ is a polynomial of degree d with integer coefficients. If values $X_1, X_2, \ldots, X_n$ are chosen independently from the uniform distribution on $\{0, 1, 2, \ldots, d\}$, then*

$$\mathrm{P}[p(X_1, X_2, \ldots, X_n) = 0] < 1/2\,.$$

This lemma can be found in Schwartz (1980), and it can be proved by induction on the number of variables. The main point of the lemma is that it tells us that we can put our relation to the test k times, and, if it is not a true identity, then we will discover the fact with probability at least $1 - 2^{-k}$. After suffering prior doubts about the possibility of selecting points at random from $[0,1]^8$, one might worry about how we can make random choices out of $\{0, 1, \ldots, d\}$. Clearly, the situation is better, but still it may not be fully resolved. This question is dealt with at length in Chapter 6.

1.2 Probabilistic Analysis of Algorithms

1.2.1 Sample Analyses: The Assignment Problem

Probabilistic analysis is inevitably mathematical, so no everyday example can help illustrate this second of the great gates through which probability theory enters the theory of algorithms. Still, for our first such example, we can use a task that often arises as a module in larger computational problems and that is evocatively expressed in language that reflects its origins in industrial engineering. It requires a slight increase in technical level over our everyday examples, but it is still friendly and nontechnical. An additional

benefit of the problem is that its analysis offers several surprises that have only recently been discovered.

We suppose that we have a set of n jobs labeled $\{1, 2, \ldots, n\}$ and a set of n machines on which to perform the jobs that are similarly labeled. We further suppose that there is a matrix of numbers c_{ij} that describe the cost of doing job i on machine j. A natural and important computational task is to specify a one-to-one assignment of jobs to machines that minimizes the total cost. In other words, the task is to determine a permutation σ that minimizes

$$A(\sigma) = \sum_{i=1}^{n} c_{i\sigma(i)} \cdot$$

This is called the **assignment problem**, and one can see how it could come about easily on its own or in the context of a larger computational task.

One natural way to try to solve this problem is to use a "greedy" strategy. There are two natural ways to proceed, so we will consider the simplest one first. We look at job 1 and assign it to machine j, where j is chosen to minimize c_{1j} over all $1 \leq j \leq n$. We then successively assign jobs $2, 3, \ldots, n$ by choosing at each step we take the least costly among the unassigned machines.

There is a slightly more sophisticated algorithm that takes a more globally greedy perspective. For the first assignment, one chooses the assignment $i \rightarrow j$, where (i, j) are chosen to minimize c_{ij} over all n^2 possible choices. The second assignment then chooses the cheapest possibility from the remaining set of $(n-1)^2$ pairs of feasible assignments. One then continues in this globally greedy way until one has a complete assignment of jobs to machines. The natural problem in this context is to determine which of the two methods is better. Another compelling problem is to determine if there are still other methods that do substantially better than these two greedy procedures.

One way to approach these problems is to assume some probability model on the input data c_{ij} and to calculate appropriate measures of performance under this model. An issue that emerges at this point is that for algorithms that do not return an exact solution, there are two compelling dimensions along which to measure performance: the quality of the solution obtained and the amount of time needed to obtain that solution. For the simple and global greedy algorithms for the assignment problem, the running time analysis does not depend on probability theory, so we will consider it first.

Because one can find the smallest element in a list of k items in time that

is bounded by a constant times k, the running time of the first algorithm is bounded by $c[n + (n-1) + (n-2) + \cdots + 1] = O(n^2)$. The second algorithm requires a little more knowledge to implement efficiently, but for people who have had some exposure to the subject of data structures there are some off-the-shelf tools that come to mind almost immediately. One such tool is a data structure called a **heap**. Given m items that have a total order, one can build this structure in time $O(m)$, and it has the remarkable property that one can delete the largest item in the heap at a cost bounded by $c \log m$ and be left with a new heap that is now of size $m - 1$. There are other structures besides heaps that have this property, but unless one is ready to get down to coding, there is no need to do more than note that there is some *abstract* data structure that can be built at the specified cost and that permits deletion at the specified cost.

We can now go about measuring the running time cost of our greedy algorithms. If we put the n^2 costs c_{ij} into a heap at a cost of $O(n^2)$, then even if we have to delete all the items in the heap to build our assignment, the total time used is still only $O(n^2 \log n)$. The bottom line of these two running time analyses is that both of the algorithms are quite practical, just as we expected them to be, and the global greedy algorithm is more costly in time to the tune of a factor of $\log n$. This second result is also much as one might have expected.

A bigger surprise comes when one looks at the quality of the solutions that are obtained by the two algorithms. Here, a natural way to proceed is to make some probabilistic assumptions about the cost c_{ij}. One is not likely to find any assumption that is more natural than to take the c_{ij} to be independent, identically distributed random variables. Further, because we are just looking for *some* computable feature of merit that casts light on the quality of the assignments we obtain, we might as well go the rest of the way in making our model easy by assuming that the c_{ij} are uniformly distributed on $[0, 1]$.

This setup does indeed make it easy to compute the expected value of the total assignment cost A_n that is obtained using the simple greedy algorithm. Because the expected value of the minimum of k independent, uniformly distributed random variables is exactly equal to $1/(k+1)$, we see

$$E(A_n) = 1/(n+1) + 1/n + 1/(n-1) + \ldots + 1/2 \sim \log_e(n).$$

The analysis of the expected value of A'_n, the cost of the assignment obtained under the global greedy method, is a little bit more difficult to obtain. The

problem is that one loses all independence in the model after the very first step. One can nevertheless show by setting up a simple integral equation based on dynamic programming that one again has that $E(A_n') \sim \log_e(n)$. The punchline is that even though the global greedy algorithm takes a little more time to compute and seems to be a more powerful strategy, the value of the assignment that it yields is typically no better than that one gets by the simple greedy strategy.

Before leaving this example, we note that these heuristics fall short of providing the deepest understanding of $E(A_n^{OPT})$, where $E(A_n^{OPT})$ denotes the least cost of any assignment. Karp (1987) showed by a remarkable argument that $E(A_n^{OPT}) \leq 2$, and the insight provided by Karp's proof has been shown by Dyer et al. (1986) to apply to many other problems that can be expressed as linear programs. Finally, Mézard and Parisi (1987) have offered intriguing calculations based on methods of statistical mechanics that suggest $E(A_n^{OPT}) \sim \pi^2/6$ as $n \to \infty$. For further information on the history of A_n^{OPT} and conjectures concerning its behavior, one can consult Steele (1990).

1.2.2 Quick Lessons from Quicksort

For the second example of a probabilistic analysis of a deterministic algorithm, we have to increase the technical level somewhat. We will first sketch some salient features of an archetypical example: **Quicksort**. This example is a little shopworn, because it is considered in almost every introductory course in the theory of algorithms. Still, Quicksort is an algorithm that has remarkable intrinsic richness, and it is one of the few algorithms to be the subject of an entire monograph (Sedgewick, 1980).

Given a list of n records $(R_1, R_2, \ldots, R_n)$ that are associated with n keys $(K_1, K_2, \ldots, K_n)$ for which there is a total ordering, Quicksort is elegantly expressed as a recursive process that returns an ordered list of the records $(R_{i_1}, R_{i_2}, \ldots, R_{i_n})$ such that $K_{i_1} \leq K_{i_2} \leq \cdots \leq K_{i_n}$.

In its simplest form, Quicksort takes the key K_1 and divides the records into those whose key is less than K_1 and those whose key is greater. The algorithm then recursively calls itself on each of the two resulting sublists. If one resists the temptation to add all of the refinements that help make Quicksort a truly practical procedure, it is a joy to analyze its "average case" behavior.

To sketch the analysis, we first suppose that the list that we begin with is in random order; that is, we suppose that we are equally likely to begin

with the list in any of the $n!$ possible orderings. The simplest consequence of this assumption is that the key K_1 is equally likely to have any of the n possible ranks, and, moreover, the two sublists produced by splitting at K_1 are again in random order.

If we let Q_n denote the expected number of comparisons that are needed to sort a list of n items under the model that the input lists are in random order, then it is easy to check that Q_n satisfies the recursion

$$Q_n = (n-1) + \frac{1}{n}\sum_{k=0}^{n-1}\{Q_k + Q_{n-k-1}\}.$$

This equation expresses the fact that the first split requires n–1 comparisons, and if k is the number of keys less than K_1, one then has to call the Quicksort procedure on two lists of size k and n–k–1, respectively. The only remaining point is that by our randomness assumption, K_1 has probability $1/n$ of having $0, 1, \ldots, n-1$ keys less than itself.

To gain any insight about the efficiency of the Quicksort process, one still needs to solve the recursion, or at least to extract the asymptotics of Q_n from it. This can be done without great difficulty, and one finds that $Q_n \sim 2n \log_e n$.

This analysis provides the basis of a number of insights into the behavior of the Quicksort process. Although we are more concerned with the pointers it provides about basic theoretical structures, it is worth noting quickly that our analysis already suggests when Quicksort will perform poorly. The random divisions are efficient when they almost cut the list in half, and they are inefficient when the division is highly lopsided. It is easy to check that if one begins with a sorted list, the Quicksort process we have described would make $(n-1) + (n-2) + \cdots + 1 = n(n-1)/2$ comparisons, which is poor indeed in comparison with the expected performance on a random list.

1.3 How To Use This Survey

Chapters 2 through 11 stake out a very large part of the territory at the interface of probability and algorithms. Each of these chapters is more technical than this introduction. Still, each is designed to be accessible to individuals who are considering new directions of research.

Since each of these chapters has its own introduction and abstract, browsing is encouraged. The chapters can be read independently, and overlap has been kept to a minimum.

References

Dyer, M.E., A.M. Frieze, and C.J.H. McDiarmid (1986), On linear programs with random costs, *Math. Programming* **35**, 3–16.

Fiat, A., R. Karp, M. Luby, L. McGeoch, D. Sleator, and N. Young (1991), On competitive algorithms for paging problems, *J. Algorithms* **12**, 685–699.

Karp, R.M. (1987), An upper bound on the expected cost of an optimal assignment, in *Discrete Algorithms and Complexity: Proceedings of the Japan–U.S. Joint Seminar*, D. Johnson et al., eds., Academic Press, New York.

Karp, R.M. (1991), An introduction to randomized algorithms, *Discrete Appl. Math.* **34**, 165–201.

Knuth, D.E. (1973), *The Art of Computer Programming*, vol. 3, Addison-Wesley, Reading, Mass.

Mézard, M., and G. Parisi (1987), On the solution of the random link matching problem, *J. Physique* **48**, 1451–1459.

Raghavan, P. (1990), Lecture notes on randomized algorithms, Research Report, IBM, Yorktown Heights, N.Y., unpublished.

Schwartz, J.T. (1980), Fast probabilistic algorithms for verification of polynomial identities, *J. Assoc. Comput. Mach.* **27**, 701–717.

Sedgewick, R. (1980), *Quicksort*, Garland, New York.

Seidel, R. (1992), Backwards analysis of randomized geometric algorithms, in *New Trends in Discrete and Computational Geometry*, J. Pach, ed., Springer-Verlag, New York, to appear.

Steele, J.M. (1990), Probability and statistics in the service of computer science: Illustrations using the assignment problem, *Commun. Stat.* **19**, 4315–4329.