Uses of Randomness in Computation*

Richard P. Brent
Computer Sciences Laboratory
Australian National University
rpb@cslab.anu.edu.au

April 1994

rpb147t typeset using LaTeX

Abstract

Random number generators are widely used in practical algorithms. Examples include simulation, number theory (primality testing and integer factorization), fault tolerance, routing, cryptography, optimization by simulated annealing, and perfect hashing.

Complexity theory usually considers the worst-case behaviour of deterministic algorithms, but it can also consider average-case behaviour if it is assumed that the input data is drawn randomly from a given distribution. Rabin popularised the idea of "probabilistic" algorithms, where randomness is incorporated into the algorithm instead of being assumed in the input data. Yao showed that there is a close connection between the complexity of probabilistic algorithms and the average-case complexity of deterministic algorithms.

In this talk I give examples of the uses of randomness in computation, discuss the contributions of Rabin, Yao and others, and mention some open questions.

Checking out Galileo

The Galileo spacecraft is somewhere near Jupiter, but its main radio antenna is not working, so communication with it is very slow. Suppose we want to check that a critical program in Galileo's memory is correct, and has not been corrupted by a passing cosmic ray. How can we do this without transmitting the whole program to or from Galileo ?

Here is one way[1]. The program we want to check (say N_1) and the correct program on Earth (say N_2) can be regarded as multiple-precision integers. Choose a random prime number p in the interval $(10^9, 2 \times 10^9)$. Transmit p to Galileo and ask it to compute

$$r_1 \leftarrow N_1 \bmod p$$

and send it back to Earth. Only a few bits (no more than 64 for p and r_1) need be transmitted between Earth and Galileo, so we can afford to use good error correction/detection.

[1] Rabin's "Library of Congress on Mars" problem.

On Earth we compute $r_2 \leftarrow N_2 \bmod p$, and check if $r_1 = r_2$. There are two possibilities:

- $r_1 \neq r_2$. We conclude that $N_1 \neq N_2$. Galileo's program has been corrupted ! If there are only a small number of errors, they can be localised by binary search using $O(\log\log N_1)$ small messages.

- $r_1 = r_2$. We conclude that Galileo's program is *probably* correct. More precisely, if Galileo's program is *not* correct there is only a probability of less than 10^{-9} that $r_1 = r_2$, i.e. that we have a "false positive". If this probability is too large for the quality-assurance team to accept, just repeat the process (say) ten times with different random primes $p_1, p_2, \ldots, p_{10}$. If $N_1 \neq N_2$, there is a probability of less than

  $$10^{-90}$$

  that we get $r_1 = r_2$ ten times in a row. This should be good enough.

The Structure

Our procedure has the following form. We ask a question with a yes/no answer. The precise question depends on a random number. If the answer is “no”, we can assume that it is correct. If the answer is “yes”, there is a small probability of error, but we can reduce this probability to a negligible level by repeating the procedure a few times with *independent* random numbers.

We call such a procedure a *probabilistic* algorithm; other common names are *randomised* algorithm and *Monte Carlo* algorithm.

Disclaimer

It would be much better to build error correcting hardware into Galileo, and not depend on checking from Earth.

Testing Primality

Here is another example[2] with the same structure. We want an algorithm to determine if a given odd positive integer n is prime. Write n as $2^k q + 1$, where q is odd and $k > 0$.

Algorithm P

1. Choose a random integer x in $(1, n)$.
2. Compute $y = x^q \bmod n$. This can be done with $O(\log q)$ operations mod n, using the binary representation of q.
3. If $y = 1$ then return "yes".
4. For $j = 1, 2, \ldots, k$ do

   if $y = n - 1$ then return "yes"

   else if $y = 1$ then return "no"

   else $y \leftarrow y^2 \bmod n$.
5. Return "no".

---

[2] Due to M. O. Rabin, with improvements by G. L. Miller. See Knuth, Vol. 2, §4.5.4.

Fermat's Little Theorem

To understand the mathematical basis for Algorithm P, recall Fermat's little Theorem: if n is prime and $0 < x < n$, then

$$x^{n-1} = 1 \bmod n.$$

Thus, if $x^{n-1} \neq 1 \bmod n$, we can definitely say that n is composite.

Unfortunately, the converse of Fermat's little theorem is false: if $x^{n-1} = 1 \bmod n$ we can not be sure that n is prime. There are examples (called *Carmichael numbers*) of composite n for which x^{n-1} is always 1 mod n when $\mathrm{GCD}(x, n) = 1$. The smallest example is

$$561 = 3 \cdot 11 \cdot 17$$

Another example is[3]

$$n = 1729 = 7 \cdot 13 \cdot 19$$

[3] Hardy's taxi number, $1729 = 12^3 + 1^3 = 10^3 + 9^3$.

An Extension

A slight extension of Fermat's little Theorem is useful, because its converse is *usually* true.

If $n = 2^k q + 1$ is an odd prime, then either $x^q = 1 \bmod n$, or the sequence

$$\left(x^{2^j q} \bmod n\right)_{j=0,1,\ldots,k}$$

ends with 1, and the value just preceding the first appearance of 1 must be $n - 1$.

Proof: If $y^2 = 1 \bmod n$ then $n|(y-1)(y+1)$. Since n is prime, $n|(y-1)$ or $n|(y+1)$. Thus $y = \pm 1 \bmod n$. $\square$

The extension gives a *necessary* (but not sufficient) condition for primality of n. Algorithm P just checks if this condition is satisfied for a random choice of x, and returns "yes" if it is.

Reliability of Algorithm P

Algorithm P can not give false negatives (unless we make an arithmetic mistake), but it can give false positives (i.e. "yes" when n is composite). However, the probability of a false positive is less than 1/4. (Usually much less – see Knuth, ex. 4.5.4.22.) Thus, if we repeat the algorithm 10 times there is less than 1 in 10^6 chance of a false positive, and if we repeat 100 times the results should satisfy anyone but a pure mathematician.

Algorithm P works fine even if the input is a Carmichael number.

Use of Randomness

Note that in both our examples randomness was introduced into the algorithm.

We did not make any assumption about the distribution of inputs.

Summary of Algorithm P

Given any $\varepsilon > 0$, we can check primality of a number n in

$$O((\log n)^3 \log(1/\varepsilon))$$

bit-operations[4], provided we are willing to accept a probability of error of at most ε.

By way of comparison, the best known *deterministic* algorithm takes

$$O((\log n)^{c \log\log\log n})$$

bit-operations, and is much more complicated. If we assume the *Generalised Riemann Hypothesis*, the exponent can be reduced to 5. (But who believes in GRH with as much certainty as Algorithm P gives us ?)

---

[4]We can factor n deterministically in $O(\log n)$ *arithmetic* operations, but this result is useless because the operations are on numbers as large as 2^n. Thus, it is more realistic to consider bit-operations.

Error-Free Algorithms

The probabilistic algorithms considered so far (*Monte Carlo* algorithms) can give the wrong answer with a small probability. There is another class of probabilistic algorithms (*Las Vegas* algorithms) for which the answer is always correct; only the runtime is random[5]. An interesting example is H. W. Lenstra's *elliptic curve method* (ECM) for integer factorisation. To avoid trivial cases, suppose we want to find a prime factor $p > 3$ of an odd composite integer N.

To motivate ECM, consider an earlier algorithm, Pollard's "$p - 1$" method. This works if $p - 1$ is "smooth", i.e. has only small prime factors. $p - 1$ is important because it is the order of the multiplicative group G of the field F_p. The problem is that G is fixed.

[5] In practical cases the expected runtime is finite. It is possible that the algorithm does not terminate, but with probability zero.

Lenstra's Idea

Lenstra had the idea of using a group $G(a,b)$ which depends on parameters (a,b). By randomly selecting a and b, we get a large set of different groups, and some of these should have smooth order.

The group $G(a,b)$ is the group of points on the *elliptic curve*

$$y^2 = x^3 + ax + b \bmod p,$$

and by a famous theorem[6] the order of $G(a,b)$ is an integer in the interval

$$(p - 1 - 2\sqrt{p}, \;\; p - 1 + 2\sqrt{p})$$

The distribution in this interval is not uniform, but it is "close enough" to uniform for our purposes.

[6] The "Riemann hypothesis for finite fields". $G(a,b)$ is known as the "Mordell-Weil" group. The result on its order follows from a theorem of Hasse (1934), later generalised by A. Weil and Deligne.

Runtime of ECM

Under plausible assumptions ECM has expected run time

$$T = O\left(\exp(\sqrt{c\log p\log\log p})(\log N)^2\right),$$

where $c \simeq 2$.

Note that T depends mainly on the size of p, the factor found, and not very strongly on N. In practice the run time is close to an exponentially distributed random variable with mean and variance about T.

ECM Example

ECM is the best known algorithm for finding moderately large factors of very large numbers.

Consider the 617-decimal digit Fermat number $F_{11} = 2^{2^{11}} + 1$. Its factorisation is:

$$\begin{aligned} F_{11} &= 319489 \cdot 974849 \cdot \\ &\quad 167988556341760475137 \cdot \\ &\quad 3560841906445833920513 \cdot p_{564}, \end{aligned}$$

where p_{564} is a 564-decimal digit prime.

In 1989 I found the 21-digit and 22-digit prime factors using ECM. The factorisation required about 360 million multiplications mod N, which took less than 2 hours on a Fujitsu VP 100 vector processor.

Minimal Perfect Hashing

Hashing is a common technique used to map words into a small set of integers (which may then be used as indices to address a table). Thus, the computation $r_1 \leftarrow N_1 \bmod p$ used in our "Galileo" example can be considered as a hash function.

Formally, consider a set

$$W = \{w_0, w_1, \ldots, w_{m-1}\}$$

of m words w_j, each of which is a finite string of symbols over a finite alphabet Σ. A *hash function* is a function

$$h : W \to I,$$

where $I = \{0, 1, \ldots, k-1\}$ and k is a fixed integer (the table size).

Collisions

A *collision* occurs if two words w_1 and w_2 map to the same address, i.e. if $h(w_1) = h(w_2)$. There are various techniques for handling collisions. However, these complicate the algorithms and introduce inefficiencies. In applications where W is fixed (e.g. the reserved words in a compiler), it is worth trying to avoid collisions.

Perfection

If there are no collisions, the hash function is called *perfect*.

Minimal Perfection

For a perfect hash function, we must have $k \geq m$. If $k = m$ the hash function is *minimal*.

Problem

Given a set W, how can we compute a minimal perfect hash function ?

The CHM Algorithm

Czech, Havas and Majewski (CHM) give a probabilistic algorithm which runs in expected time $O(m)$ (ignoring the effect of finite word-length). Their algorithm uses some properties of *random graphs*.

Take $n = 3m$, and let

$$V = \{1, 2, \ldots, n\}.$$

CHM take two independent pseudo-random functions[7]

$$f_1 : W \to V, \quad f_2 : W \to V,$$

and let

$$E = \{(f_1(w), f_2(w)) \mid w \in W\}.$$

We can think of $G = (V, E)$ as a random graph with n vertices V and (at most) m edges E.

[7]How ? This is a theoretical weak point of their algorithm, but in practice their solution is satisfactory.

Acyclicity

If G has less than m edges or G has cycles, CHM reject the choice of f_1, f_2 and try again. Eventually they get a graph G with m edges and no cycles. Because $n = 3m$, the expected number of trials is a constant (about $\sqrt{3}$, or more generally $\sqrt{\frac{n}{n-2m}}$, for large m and $n > 2m$).

The Perfect Hash Function

Once an acceptable G has been found, it is easy to compute (and store in a table) a function

$$g : V \to 0, 1, \ldots, m - 1$$

such that

$$h(w) = g(f_1(w)) + g(f_2(w)) \bmod m$$

is the desired minimal perfect hash function. We can even get

$$h(w_j) = j$$

for $j = 0, 1, \ldots, m - 1$. All this requires is a depth-first search of G.

Implementation

CHM report that on a Sun SPARCstation 2 they can generate a minimal perfect hash function for a set of $m = 2^{19}$ words in 33 seconds. Earlier algorithms required time which (at least in the worst case) was an exponentially increasing function of m, so could only handle very small m.

Permutation Routing

A network G is a connected, undirected graph with N vertices $0, 1, \ldots, N-1$.

The permutation routing problem on G is: given a permutation π of the vertices, and a message (called a *packet*) on each vertex, route packet j from vertex j to vertex $\pi(j)$. It is assumed that at most one packet can traverse each edge in unit time, and that we want to minimise the time for the routing.

In practice we only want to consider *oblivious* algorithms, where the route taken by packet j depends only on $(j, \pi(j))$.

For simplicity, assume that the G is a d-dimensional hypercube, so $N = 2^d$. Similar results apply to other networks.

Example: Leading Bit Routing

A simple algorithm for routing packets on a hypercube chooses which edge to send a packet along by comparing the current address and the destination address and finding the highest order bit position in which these addresses differ.

For example, consider the bit-reversal permutation $01001001 \rightarrow 10010010$. Each "$\downarrow$" corresponds to traversal of an edge in the hypercube.

$$
\begin{array}{cccccccc}
0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\
\downarrow & & & & & & & \\
1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\
 & \downarrow & & & & & & \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
 & & & \downarrow & & & & \\
1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\
 & & & & \downarrow & & & \\
1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\
 & & & & & & \downarrow & \\
1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\
 & & & & & & & \downarrow \\
1 & 0 & 0 & 1 & 0 & 0 & 1 & 0
\end{array}
$$

Borodin and Hopcroft's bound

The following result says that there are no "uniformly good" deterministic algorithms for oblivious permutation routing:

Theorem: For any deterministic, oblivious permutation routing algorithm, there is a permutation π for which the routing takes $\Omega(\sqrt{N/d^3})$ steps.

Example: For the leading-bit routing algorithm, take π to be the bit-reversal permutation, i.e.

$$\pi(b_0 b_1 \ldots b_{d-1}) = b_{d-1} \ldots b_1 b_0 \ .$$

Suppose d is even. Then at least $2^{d/2}$ packets are routed through vertex 0. To prove this, consider the routing of

$$xx \ldots xx00 \ldots 00 \ ,$$

where there are at least $d/2$ trailing zeros.

Valiant and Brebner's algorithm

We can do much better with a probabilistic algorithm. Valiant suggested:

1. Choose a random mapping σ (not necessarily a permutation).

2. Route message j from vertex j to vertex $\sigma(j)$ using the leading bit algorithm (for $0 \le j < N$).

3. Route message j from vertex $\sigma(j)$ to vertex $\pi(j)$.

This seems crazy[8], but it works ! Valiant and Brebner prove:

Theorem: With probability greater than $1 - 1/N$, every packet reaches its destination in at most $14d$ steps.

Corollary: The expected number of steps to route all packets is less than $15d$.

[8] I don't know of any manufacturer who has been persuaded to implement it. Probably it would be hard to sell.

Pseudo-deterministic Algorithms

Some probabilistic algorithms use many independent random numbers, and because of the "law of large numbers" their performance is very predictable. One example is the *multiple-polynomial quadratic sieve* (MPQS) algorithm for integer factorisation.

Suppose we want to factor a large composite number N (not a perfect power). The key idea of MPQS is to generate a sufficiently large number of congruences of the form

$$y^2 = p_1^{\alpha_1} \cdots p_k^{\alpha_k} \bmod N,$$

where $p_1, \ldots, p_k$ are small primes in a precomputed "factor base", and y is close to $\sqrt{N}$. Many y are tried, and the "successful" ones are found efficiently by a sieving process.

Making some plausible assumptions, the expected run time of MPQS is

$$T = O(\exp(\sqrt{c \log N \log \log N})),$$

where $c \simeq 1$. In practice, this estimate is good and the variance is small.

MPQS Example

MPQS is currently the best general-purpose algorithm for factoring moderately large numbers N whose factors are in the range $N^{1/3}$ to $N^{1/2}$. For example, A. K. Lenstra and M. S. Manasse recently found

$$\begin{aligned} 3^{329}+1 &= 2^2 \cdot 547 \cdot 16921 \cdot 256057 \cdot \\ & \quad 36913801 \cdot 177140839 \cdot \\ & \quad 1534179947851 \cdot p_{50} \cdot p_{67}\ , \end{aligned}$$

where the penultimate factor p_{50} is a 50-digit prime 24677078822840014266652779036768062918372697435241, and the largest factor p_{67} is a 67-digit prime.

The computation used a network of workstations for "sieving", then a super-computer for the solution of a very large linear system.

A "random" 129-digit number (RSA129) has just been factored in a similar way to win a \$100 prize offered by Rivest, Shamir and Adleman in 1977.

Complexity Theory of Probabilistic Algorithms

Do probabilistic algorithms have an advantage over deterministic algorithms ? If we allow a small probability of error, the answer is **yes**, as we saw for the Galileo example. If no error is allowed, the answer is (probably) **no**.

A. C. Yao considered probabilistic algorithms (modelled as decision trees) for testing properties P of undirected graphs (given by their adjacency matrices) on n vertices. He also considered deterministic algorithms which assume a given distribution of inputs (i.e. a distribution over the set of graphs with n vertices).

Definitions

Yao defines

randomized complexity $F_R(P)$ as an

infimum (over all possible algorithms) of a
maximum (over all graphs with n vertices) of the
expected runtime.

and

distributional complexity $F_D(P)$ as a

supremum (over input distributions) of a
minimum (over all possible deterministic algorithms) of the
average runtime.

Informally, $F_R(P)$ is how long the best probabilistic algorithm takes for testing P; and $F_D(P)$ is the average runtime we can always guarantee with a good deterministic algorithm, provided the distribution of inputs is known.

Yao's Result

Yao (1977) claims that $F_D(P) = F_R(P)$ follows from the minimax theorem of John von Neumann (1928). The minimax theorem is familiar from the theory of two-person zero-sum games.

So What ?

Yao's result should not discourage the use of probabilistic algorithms – we have already given several examples where they out-perform known deterministic algorithms, and there are many similar examples.

Yao's computational model is very restrictive. Because n is fixed, table lookup is permitted, and the maximum complexity of any problem is $O(n^2)$.

Adleman and Gill's result

Less restrictive models have been considered by Adleman and Gill. Without going into details of the definitions, they prove:

Theorem: If a Boolean function has a randomised, polynomial-sized circuit family, then it has a deterministic, polynomial-sized circuit family.

There are two problems with this result:

- The deterministic circuit may be larger (by a factor of about n, the number of variables) than the original circuit.

- The transformation is not "uniform" – it can not be computed in polynomial time by a Turing machine. The proof of the theorem is by a counting argument applied to a matrix with 2^n rows, so it is not constructive in a practical sense.

The Class RP

We can formalise the notion of a probabilistic algorithm and define a class RP of languages L such that $x \in L$ is *accepted* by a probabilistic algorithm in polynomial time with probability $p \geq 1/2$ say[9], but $x \notin L$ is never accepted. Clearly

$$P \subseteq RP \subseteq NP,$$

where P and NP are the well-known classes of problems which are accepted in polynomial time by deterministic and nondeterministic (respectively) algorithms.

It is plausible that

$$P \subset RP \subset NP,$$

but this would imply that $P \neq NP$, so it is a difficult question.

[9] Any fixed value in $(0, 1)$ can be used in the definition.

Perfect Parties

B. McKay (ANU) and S. Radziszowski (Rochester) are interested in the size of the largest "perfect party". Because people at parties tend to cluster in groups of five, we consider a party to be *imperfect* if there are five people who are mutual acquaintances, or five who are mutual strangers. A *perfect* party is one which is not imperfect[10].

McKay *et al* have performed a probabilistic computation which shows that, with high probability, the largest perfect party has 42 people.

Ramsey Numbers

$R(s, t)$ is the smallest n such that each graph on n or more vertices has a clique of size s *or* an independent set of size t.

Examples: $R(3, 3) = 6$, $R(4, 4) = 18$, $R(4, 5) = 25$, and $43 \le R(5, 5) \le 49$.

Perfect party organisers would like to know $R(5, 5) - 1$.

[10]Thanks to John Slaney for motivating this definition.

The Computation

A $(5,5,n)$-graph is a graph with n vertices, no clique of size 5, and no independent set of size 5. There are 328 known $(5,5,42)$-graphs, not counting complements as different. McKay *et al* generated 5812 $(5,5,42)$-graphs using simulated annealing, starting at random graphs. All 5812 turned out to be known.

If there were any more $(5,5,42)$-graphs, and if the simulated annealing process is about equally likely to find any $(5,5,42)$-graph[11], then another such graph would have been found with probability greater than

$$0.99999998$$

Thus, there is convincing evidence that all $(5,5,42)$-graphs are known. None of these graphs can be extended to $(5,5,43)$-graphs. Thus, it is very unlikely that such a graph exists, and it is very likely that

$$R(5,5) - 1 = 42$$

[11]There is no obvious way to prove this.

A Rigorous Proof ?

A rigorous proof that $R(5,5) - 1 = 42$ would take thousands of years of computer time[12], so the probabilistic argument is the best that is feasible at present, unless we can get time on *Deep Thought*.

[12] Based on the fact that it took seven years of Sparcstation time to show that $R(4,5) = 25$.

Omissions

We did not have time to mention applications of randomness to serial or parallel algorithms for:

- sorting and selection,
- computer security,
- cryptography,
- computational geometry,
- load-balancing,
- collision avoidance,
- online algorithms,
- optimisation,
- numerical integration,
- graphics and virtual reality,
- avoiding degeneracy,
- and many other problems.

Another Omission

We did not discuss algorithms for generating pseudo-random numbers – that would require another talk.

> Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin.
>
> *John von Neumann, 1951*

Conclusion

- Probabilistic algorithms are useful.
- They are often simpler and use less space than deterministic algorithms.
- They can also be faster, if we are willing to live with a minute probability of error.

- Give good lower bounds for the complexity of probabilistic algorithms (with and without error) for interesting problems.
- Show how to generate independent random samples from interesting structures (e.g. finite groups defined by relations, various classes of graphs, ...) to provide a foundation for probabilistic algorithms on these structures.
- Consider the effect of using pseudo-random numbers instead of genuinely random numbers.
- Extend Yao's results to a more realistic model of computation.
- Give a uniform variant of the Adleman-Gill theorem.
- Show that $P \neq RP$ (hard).

References

[1] L. M. Adleman, "Two theorems on random polynomial time", *Proc. 19th Annual Symposium on Foundations of Computer Science*, IEEE, New York, 1978, 75–83.

[2] L. M. Adleman, "On distinguishing prime numbers from composite numbers (extended abstract)", *Proc. IEEE Symp. Found. Comp. Sci.* 21 (1980), 387–406.

[3] L. M. Adleman and M. A. Huang, "Recognizing primes in random polynomial time", *Proc. Nineteenth Annual ACM Symposium on the Theory of Computing*, ACM, New York, 1987, 462–469.

[4] S. L. Anderson, "Random number generators on vector supercomputers and other advanced architectures", *SIAM Review* 32 (1990), 221-251.

[5] P. van Emde Boas, "Machine models, computational complexity and number theory", in [34], 7–42.

[6] B. Bollobás, *Random Graphs*, Academic Press, New York, 1985.

[7] A. Borodin and J. E. Hopcroft, "Routing, merging, and sorting on parallel models of computation", *J. Computer and System Sciences* 30 (1985), 130–145.

[8] R. P. Brent, "Factorisation of the eleventh Fermat number (preliminary report)", *AMS Abstracts* 10 (1989), 89T–11–73.

[9] R. P. Brent, "Parallel algorithms for integer factorisation", in *Number Theory and Cryptography* (edited by J. H. Loxton), Cambridge University Press, 1990. Preliminary version available by anonymous ftp from `nimbus.anu.edu.au:/pub/Brent/rpb115.*`

[10] R. P. Brent, "Vector and parallel algorithms for integer factorisation", *Proc. Third Australian Supercomputer Conference*, Melbourne, 1990. Preliminary version available by ftp from `nimbus.anu.edu.au:/pub/Brent/rpb122.*`

[11] G. Buffon, "Essai d'arithmétique morale", *Supplément à l'Histoire Naturelle* 4, 1777.

[12] T. R. Caron and R. D. Silverman, "Parallel implementation of the quadratic sieve", *J. Supercomputing* 1 (1988), 273–290.

[13] Z. J. Czech, G. Havas and B. S. Majewski, "An optimal algorithm for generating minimal

perfect hash functions", *Information Processing Letters* 43 (1992), 257–264.

[14] P. Erdös and J. Spencer, *The Probabilistic Method in Combinatorics*, Academic Press, New York, 1974.

[15] P. Erdös and A. Renyi, "On random graphs, I", *Publicationes Mathematicae* 6 (1959), 290–297.

[16] R. W. Floyd and R. L. Rivest, "Expected time bounds for selection", *Comm. ACM* 18 (1975), 165–172.

[17] R. Freivalds, "Fast probabilistic algorithms", in *Mathematical Foundations of Computer Science* (Lecture Notes in Computer Science, 74), Springer-Verlag, Berlin, 1979.

[18] J. Gill, "Computational complexity of probabilistic Turing machines", *SIAM J. Computing* 6 (1977), 675–695.

[19] S. Goldwasser and J. Kilian, "Almost all primes can be quickly certified", *Proc. 18th Annual ACM Symposium on Theory of Computing*, 1986, 316–329.

[20] R. L. Graham, B. L. Rothschild and J. H. Spencer, *Ramsey Theory*, John Wiley, New York, 1980.

[21] R. M. Karp, “The probabilistic analysis of some combinatorial search algorithms”, in [56], 1–19.

[22] R. M. Karp, “An introduction to randomized algorithms”, *Discrete Applied Mathematics* 34 (1991), 165–201.

[23] R. M. Karp and M. O. Rabin, “Efficient randomized pattern-matching algorithms”, *IBM J. Research and Development* 31 (1987), 249–260.

[24] R. M. Karp and V. Ramachandran, “Parallel algorithms for shared memory machines”, in [28], 869–941.

[25] D. E. Knuth, *The Art of Computer Programming*, Vol. 2, 2nd edition, Addison-Wesley, Menlo Park, 1981, §4.5.4.

[26] D. E. Knuth, *The Art of Computer Programming*, Vol. 3, Addison-Wesley, Menlo Park, 1973.

[27] K. de Leeuw, E. F. Moore, C. E. Shannon and N. Shapiro, “Computability by probabilistic machines”, in *Automata Studies* (C. E. Shannon and J. McCarthy, eds.), Princeton Univ. Press, Princeton, NJ, 1955, 183–212.

[28] J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, Elsevier, Amsterdam, 1990.

[29] A. K. Lenstra and H. W. Lenstra (editors), *The Development of the Number Field Sieve*, Lecture Notes in Mathematics 1554, Springer-Verlag, Berlin, 1993.

[30] A. K. Lenstra, H. W. Lenstra, Jr., M. S. Manasse and J. M. Pollard "The factorization of the ninth Fermat number", *Mathematics of Computation* 61 (1993), 319–349.

[31] A. K. Lenstra and H. W. Lenstra, Jr., "Algorithms in number theory", in [28], 675–715.

[32] H. W. Lenstra, Jr., "Primality testing", in [34], 55–77.

[33] H. W. Lenstra, Jr., "Factoring integers with elliptic curves", *Annals of Math.* (2) 126 (1987), 649–673.

[34] H. W. Lenstra, Jr. and R. Tijdeman (editors), *Computational Methods in Number Theory*, I Math. Centre Tracts 154, Amsterdam, 1982.

[35] B. D. McKay and S. P. Radziszowski, "A new upper bound for the Ramsey number $R(5,5)$", *Australasian J. Combinatorics* 5 (1991), 13–20.

[36] B. D. McKay and S. P. Radziszowski, "Linear programming in some Ramsey problems", *J. Combinatorial Theory, Ser. B*, to appear.

[37] G. L. Miller, "Riemann's hypothesis and tests for primality", *J. Comp. System Sci.* 13 (1976), 300–317.

[38] L. Monier, "Evaluation and comparison of two efficient probabilistic primality testing algorithms", *Theoret. Comput. Sci.* 12 (1980), 97–108.

[39] R. Motwani and P. Raghavan, *Randomized Algorithms*, Cambridge University Press, 1995, to appear.

[40] K. Mulmuley, *Computational Geometry: An Introduction Through Randomized Algorithms*, Prentice-Hall, New York, 1993.

[41] J. von Neumann, "Zur Theorie der Gesellschaftsspiele", *Math. Annalen* 100 (1928), 295–320. Reprinted in *John von Neumann Collected Works* (A. H. Taub, editor), Pergamon Press, New York, 1963, Vol. 6, 1–26.

[42] J. von Neumann and O. Morgenstern, *Theory of Games and Economic Behavior*, Princeton Univ. Press, Princeton, NJ, 1953.

[43] C. Pomerance, J. W. Smith and R. Tuler, "A pipeline architecture for factoring large integers with the quadratic sieve algorithm", *SIAM J. on Computing* 17 (1988), 387–403.

[44] V. Pratt, "Every prime has a succinct certificate", *SIAM J. Computing* 4 (1975), 214–220.

[45] M. O. Rabin, "Probabilistic automata", *Information and Control* 6 (1963), 230–245.

[46] M. O. Rabin, "Probabilistic algorithms", in [56], 21–39.

[47] M. O. Rabin, "Complexity of computations (1976 Turing Award Lecture)", *Comm. ACM* 20 (1977), 625–633. *Corrigendum ibid* 21 (1978), 231.

[48] M. O. Rabin, "Probabilistic algorithms for testing primality", *J. Number Theory* 12 (1980), 128–138.

[49] M. O. Rabin and Shallit, "Randomized algorithms in number theory", *Comm. Pure Appl. Math.* 39 (1986).

[50] P. Raghavan, *Lecture Notes on Randomized Algorithms*, Yale University, January 1990.

[51] R. L. Rivest, A. Shamir and L. Adleman, "A method for obtaining digital dignatures and public-key cryptosystems", *Comm. ACM* 21 (1978), 120–126.

[52] J. Seberry and J. Pieprzyk, *Cryptography: An Introduction to Computer Security*, Prentice Hall, Sydney, 1989.

[53] A. Shamir, "Factoring numbers in $O(\log n)$ arithmetic steps", *Information Processing Letters* 8 (1979), 28–31.

[54] R. D. Silverman, "The multiple polynomial quadratic sieve", *Mathematics of Computation* 48 (1987), 329–339.

[55] R. Solovay and V. Strassen, "Fast Monte Carlo test for primality", *SIAM J. on Computing* 6 (1977), 84–85; *erratum* 7 (1978), 118.

[56] J. F. Traub (editor), *Algorithms and Complexity*, Academic Press, New York, 1976.

[57] L. G. Valiant and G. J. Brebner, "Universal schemes for parallel communication", *Proc. 13th Annual ACM Symposium on the Theory of Computing*, ACM, New York, 1981, 263–277.

[58] A. C. Yao, "Probabilistic computations: towards a unified measure of complexity", *Proc. 18th Annual Symposion on Foundations of Computer Science*, IEEE, New York, 1977, 222–227.

[59] G. Yuval, "Finding nearest neighbours", *Information Processing Letters* 5 (1976), 63–65.