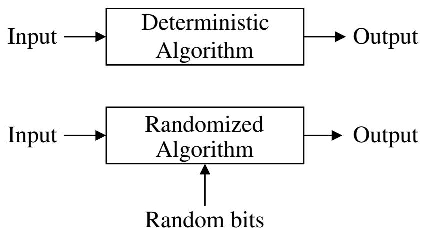
25.2 Randomized algorithms

Most of the algorithms that have appeared in this book so far are *deterministic* in the sense that there is a unique branch of computation for each given input. We have also studied an abstract class of algorithms known as *non-deterministic* algorithms, in which each step is allowed to make a choice from a finite set of possibilities. Such a choice is called a *guess*. If the input is accepted by the algorithm, a proper sequence of guesses leads to a *Yes* answer, whereas an improper sequence leads to a *No* answer. A hypothetical computer with infinite resources (like infinite number of processors) may emulate such a non-deterministic algorithm by a parallel algorithm. A practical computer can at best emulate a non-deterministic algorithm by a sequential exhaustive traversal of the computation tree, often leading to exponential blowup in the running time.

25.2.1 Monte Carlo and Las Vegas algorithms

A *randomized algorithm* depends, in addition to the input, upon a set of randomly (or pseudo-randomly) generated bits. For each input and for each sequence of random bits supplied to the algorithm, the computation is deterministic (that is, unique).

Figure 126: The difference between deterministic and randomized algorithms



A randomized algorithm may be viewed as an implementation of a non-deterministic algorithm. Instead of making *all* choices (in parallel or in an exhaustive tree traversal), a randomized algorithm makes choices guided by the input random bits. But then only one unique path in the computation tree is traversed for each sequence of random bits. Deciding the answer based upon only one path of computation may be faulty. Running a randomized algorithm multiple times (with different sequences of random bits) allows the algorithm to explore multiple computation paths. But the total number of computation paths may be exponential in the input size, and we do not seem to gain anything using a randomized algorithm.

There are situations where replacing non-deterministic choices by random choices may help us. Suppose that at least half of the computation paths lead to the answer *Yes* (for any input instance accepted by the algorithm). If we choose the computation path randomly, then the probability that we arrive at the conclusion *Yes* is at least $1/2$. If the algorithm is repeated t times with different randomly chosen computation paths, then the probability that we always reach the answer *No* is no more than $1/2^t$. That is, we obtain the answer *Yes* in at least one of the random runs of the algorithm, with probability at least $1 - \frac{1}{2^t}$. Even if t is a small constant (like 100), this probability of acceptance is overwhelmingly large. Since we started with a non-deterministic polynomial-time algorithm, the running time continues to remain polynomial. If the input is not accepted by the algorithm, then we obtain the answer *No* always, no matter how many random runs of the algorithm we carry out.

In order to illustrate this usefulness of randomized algorithms, let us take two examples. First, consider the non-deterministic algorithm of Section 24.1.1 for checking the compositeness of a positive integer n.

```
Let $l$ be the bit-length of $n$;
for ($i=0$; $i<l$; $++i$) randomly generate a bit $a_i$ from the set $\{0,1\}$;
Let $a=(a_{l-1}a_{l-2}\ldots a_1a_0)_2$;
if (($2\leqslant a\leqslant n-1$) and ($a$ divides $n$)) output "Yes"; else output "No";
```

What is the chance that we can randomly guess a non-trivial divisor of a composite integer n? If n is the square of a prime, then there exists exactly one non-trivial divisor of n, that is, among all the $n-2$ candidates, exactly one furnishes a proof of the compositeness of n. If the bits d_i are guessed randomly, the chance of hitting upon this particular divisor is $1/(n-2)$, which is exponentially small in the input size ($\log n$). Even when n is not of the above special form, it has only a very few non-trivial divisors. Searching for these divisors randomly is like searching a needle in a haystack.

A better characterization of composite numbers leads to a useful randomized algorithm. Let us recall the results of Section 22.5 on primality testing. If n is prime, then $a^{n-1}\equiv 1 \pmod n$ for every integer a coprime to n. On the other hand, if n is composite (but not a Carmichael number), then for at least half of the elements of $\mathbb{Z}_n^*$, we have $a^{n-1}\not\equiv 1 \pmod n$. This observation leads to the following randomized algorithm for deciding the compositeness of a positive integer n.

```
Let $l$ be the bit-length of $n$;
for ($i=0$; $i<l$; $++i$) randomly generate a bit $a_i$ from the set $\{0,1\}$;
Let $a=(a_{l-1}a_{l-2}\ldots a_1a_0)_2$;
if (($2\leqslant a\leqslant n-1$) and ($a^{n-1}\not\equiv 1 \pmod n$)) output "Yes"; else output "No";
```

If n is prime, the algorithm always says *No*. When n is composite (but not a Carmichael number), it outputs *Yes* if either $\gcd(a,n)>1$ or $a^{n-1}\not\equiv 1 \pmod n$. $\mathbb{Z}_n$ does not contain many elements not coprime to n, and no useful randomized algorithm can be based on locating such bases a. On the contrary, a randomly chosen $a\in\mathbb{Z}_n^*$ satisfies the Fermat congruence $a^{n-1}\equiv 1 \pmod n$ with probability no more than $1/2$. Thus, our second randomized algorithm recognizes a composite integer as composite with probability at least $1/2$. When this algorithm is repeated t times, the probability of recognizing a composite integer as prime is no less than $1-\frac{1}{2^t}$.

Carmichael numbers present an infinite family of composite integers to frustrate this randomized algorithm. In Section 22.5, this problem is repaired by an improved randomized algorithm called the Miller-Rabin test. I am not going to describe that algorithm again here, but only highlight that a properly framed non-deterministic algorithm can be converted to an efficient randomized algorithm.

The way I have presented randomized algorithms so far indicates that they may make occasional mistakes in their outputs. However, with an appropriate number of iterations of the algorithm, the probability of this error can be made arbitrarily small. Both the above randomized compositeness-testing algorithms make no mistakes when they output *Yes*. Their only errors pertain to recognizing a composite integer as prime. In that sense, the errors of these algorithms are one-sided.

Notice that when we use these algorithms for primality checking (instead of compositeness checking), the answer *No* comes with no scopes of error. The answer *Yes* comes with some (small) probability of error. We often say that these are *No-biased* algorithms for primality checking or *Yes-biased* algorithms for compositeness checking.

A randomized algorithm which may err on both *Yes* and *No* answers may be conceived of as follows. Let A and B be two randomized algorithms for solving the same problem. Suppose also that both A and B exhibit one-sided errors—A is Yes-biased, whereas B is No-biased. We run both the algorithms on an input. If A says *Yes*, we output *Yes*, whereas if B says *No*, we output *No*. Finally, if A outputs *No* and B outputs *Yes*, we take a random decision, that is, with probability $1/2$, we output *Yes*, and with probability $1/2$, we output *No*. This composite algorithm is associated with non-zero error probabilities for both *Yes* and *No* answers.

Definition A *Monte Carlo algorithm* is a randomized algorithm which produces an incorrect output with probability no more than a positive constant less than 1. For example, we can take the error probability to be $\leqslant 1/2$ or $\leqslant 1/3$ or $\leqslant 99/100$. If the error occurs for both *Yes* and *No* answers, we talk about *two-sided errors*. If the error pertains to only *Yes* answers or only *No* answers, the error is called *one-sided*.

In order that a Monte Carlo algorithm is useful, we usually demand it to have polynomial running time in the worst case. If so, we are essentially dealing with algorithms that are *always fast, but probably correct*. There is another class of randomized algorithms that are *always correct, but probably fast*.

Definition A *Las Vegas algorithm* is a randomized algorithm which always produces the correct output. In order that a Las Vegas algorithm is useful, we usually require its *expected* running time (as opposed to worst-case) to be polynomial.

As a first application of Las Vegas algorithms, we take a (large) odd prime p. We have the notations $\mathbb{Z}_p = \{0,1,2,3,\ldots,p-1\}$ and $\mathbb{Z}_p^* = \mathbb{Z}_p \setminus \{0\} = \{1,2,3,\ldots,p-1\}$. Recall that an element $a \in \mathbb{Z}_p^*$ is called a quadratic residue (modulo p) if $a \equiv b^2 \pmod{p}$ for some $b \in \mathbb{Z}_p^*$. An $a \in \mathbb{Z}_p^*$ which is not a quadratic residue is called a quadratic non-residue.

The problem of finding a random quadratic residue in $\mathbb{Z}_p^*$ is an easy problem. We can choose a $b \in \mathbb{Z}_p^*$ randomly, and output $b^2 \pmod{p}$.

The problem of finding a random quadratic nonresidue in $\mathbb{Z}_p^*$ is trickier. We know that half of the elements of $\mathbb{Z}_p^*$ are quadratic residues, and the remaining half are quadratic nonresidues. This situation is somewhat different from that of primality testing. A composite number n always contains bases which are witnesses to the compositeness of n in the Miller–Rabin test. Moreover, a large fraction of the bases turn out to be witnesses. On the other hand, a prime number cannot possess any such witness. The Miller–Rabin test is a Monte Carlo algorithm that tries to locate such witnesses. In the current case of finding random quadratic nonresidues in $\mathbb{Z}_p^*$, we know beforehand that there are many of them. The task is to locate one randomly.

Recall also that an element $a \in \mathbb{Z}_p^*$ is a quadratic residue if and only if $a^{(p-1)/2} \equiv 1 \pmod{p}$, and this modular exponentiation can be computed in time polynomial in $\log p$. Therefore we can keep on choosing a uniformly randomly from $\mathbb{Z}_p^*$ and checking whether $a^{(p-1)/2} \equiv 1 \pmod{p}$ or not. As soon as an a not satisfying this congruence is located, it is output as a random quadratic nonresidue. Since half of $\mathbb{Z}_p^*$ consists of quadratic nonresidues, each random a is a quadratic nonresidue with probability $\frac{1}{2}$. Therefore the expected number of choices of a is 2 (this is a case of geometric distribution), and the expected running time of this random-search algorithm is polynomial in $\log p$.

We may however be so unlucky that we need to draw a large number t of random elements from $\mathbb{Z}_p^*$ before locating a quadratic nonresidue. The probability of this is $\frac{1}{2^t}$ which decreases exponentially with t. Therefore large values of t are rather improbable. For example, the chance of $t = 20$ is less than once in a million, and that of $t = 100$ is once in less than 10^{30}.

For primes p of special forms, we can arrive at a worst-case polynomial-time algorithm for finding random quadratic nonresidues. For example, suppose that $p \equiv 3 \pmod 4$. We know that -1 (that is, $p-1$) is a quadratic nonresidue in this case. If we always return this particular element, one will rightfully object against the *randomness* of the output. We therefore take $a \in \mathbb{Z}_p^*$ uniformly randomly, and check whether a is a quadratic nonresidue modulo p. If that is the case, we output a. However, if a is a quadratic residue, we output the element $-a$ (that is, $p-a$). It is straightforward to establish that a is a uniformly random element of $\mathbb{Z}_p^*$ if and only if $-a$ is so too.

Next, we look at an application of Las Vegas algorithms in cryptography. Many public-key protocols use a large prime p with $p-1$ having a known prime factor q of a specified bit length (quite commonly 160 or 256). We want to find an element $g \in \mathbb{Z}_p^*$, $g \neq 1$, such that $g^q \equiv 1 \pmod p$. The cofactor $c = (p-1)/q$ can be easily computed from the knowledge of p and q. By Fermat's little theorem, $a^{p-1} \equiv 1 \pmod p$ for any $a \in \mathbb{Z}_p^*$. But then $(a^c)^q \equiv 1 \pmod p$, that is, $a^c \pmod p$, if not equal to 1, can serve the purpose of g.

Our plan is therefore to choose $a \in \mathbb{Z}_p^*$ uniformly randomly, and compute $g \equiv a^c \pmod p$ until we get a $g \neq 1$. The relevant question now is how many elements in $\mathbb{Z}_p^*$ satisfy $a^c \equiv 1 \pmod p$. It follows from a little bit of algebra that there are exactly q of them. Therefore a randomly chosen $a \in \mathbb{Z}_p^*$ satisfies $a^c \equiv 1 \pmod p$ with probability $\frac{q}{p-1} = \frac{1}{c}$. In cryptographic applications, one takes large primes p (of bit size 1024 or more), whereas the bit size of q is much smaller (like $\leqslant 256$). Consequently, c is a very large integer (of bit size 768 or more), and the probability $\frac{1}{c}$ is so low that a randomly chosen $a \in \mathbb{Z}_p^*$ gives $a^c \not\equiv 1 \pmod p$ with overwhelmingly large probability.

As a theoretically deeper achievement of Las Vegas algorithms, let us consider a polynomial $f(x)$ with integer coefficients. We are supposed to compute all the roots of $f(x)$ modulo an odd prime p. Let d be the degree of $f(x)$ modulo p. We may assume, without loss of generality, that the coefficients of $f(x)$ are members of $\mathbb{Z}_p$. Therefore, $\mathrm{O}(d \log p)$ bits are needed to represent $f(x)$, that is, we take $d \log p$ as the input size.

Polynomial root finding is not a decision problem; it is a functional problem, but that does not matter. The best deterministic algorithms known to solve this problem take time polynomial in d, but fully exponential in $\log p$. Using randomization, we can solve this problem in expected polynomial time (in both d and $\log p$).

The idea is again based on Fermat's little theorem: $a^{p-1} \equiv 1 \pmod p$ for all $a \in \mathbb{Z}_p^*$. This, in turn, implies that the polynomial $x^{p-1} - 1$ factors modulo p as

$$x^{p-1} - 1 \equiv \prod_{a \in \mathbb{Z}_p^*} (x - a) \pmod p.$$

Since $p-1$ is even, we can write $x^{p-1} - 1 = (x^{(p-1)/2} - 1)(x^{(p-1)/2} + 1)$. Let R be the set of the roots of $x^{(p-1)/2} - 1$ (the quadratic residues modulo p), and N the set of roots of $x^{(p-1)/2} + 1$ (the non-residues).

Let $a_1, a_2, \ldots, a_t$ (for some $t \leqslant d$) be all the roots of $f(x)$ modulo p. Any algorithm that splits these roots into two non-trivial subsets can be repeatedly used to obtain the individual roots a_i. To this end, we compute $f_1(x) = \gcd(x^{(p-1)/2} - 1, f(x))$ and $f_2(x) = \gcd(x^{(p-1)/2} + 1, f(x))$. If all the roots of $f(x)$ are quadratic residues, then $f_1(x) = (x-a_1)(x-a_2)\cdots(x-a_t)$ and $f_2(x) = 1$. On the other hand, if all the roots of $f(x)$ are quadratic non-residues, then $f_1(x) = 1$ and $f_2(x) = (x-a_1)(x-a_2)\cdots(x-a_t)$. In both these cases, we fail to split the set of roots of $f(x)$ in two non-trivial subsets. If, however, only some of the roots of $f(x)$ are quadratic residues, and the rest non-residues, then $f_1(x) = \prod_{a_i \in R}(x-a_i)$ and $f_2(x) = \prod_{a_i \in N}(x-a_i)$ is a desired non-trivial split in the roots of $f(x)$. We then need to split $f_1(x)$ and $f_2(x)$ recursively. However, all the roots of $f_1(x)$ are from R, and all the roots of $f_2(x)$ are from N, so the above strategy fails to split the roots further.

In order to avoid this problem, we resort to randomization. Let b be a randomly chosen member of $\mathbb{Z}_p^*$. Since $p \mid \binom{p}{k}$ for $1 \leqslant k \leqslant p-1$, and since $b^p \equiv b \pmod{p}$, we have $(x-b)^p - (x-b) \equiv x^p - x \pmod{p}$. On the other hand,

$$(x-b)^p - (x-b) = (x-b)((x-b)^{(p-1)/2} - 1)((x-b)^{(p-1)/2} + 1).$$

It is easy to check (Horner's rule) whether b is a root of $f(x)$. If so, we divide $f(x)$ by an appropriate power of $x-b$ until it no longer has the root b. We compute the two polynomials

$$f_1(x) \equiv \gcd((x-b)^{(p-1)/2} - 1, f(x)) \equiv \prod_{a_i \in R+b} (x-a_i) \pmod{p},$$

and

$$f_2(x) \equiv \gcd((x-b)^{(p-1)/2} + 1, f(x)) \equiv \prod_{a_i \in N+b} (x-a_i) \pmod{p}.$$

Even if the sets R and N fail to split the roots of $f(x)$, their random shifts $R+b$ and $N+b$ give us a handle to obtain a non-trivial split. Indeed, after trying a few random values of b, we expect to obtain a desired split with high probability (the analysis is not shown here). In other words, this algorithm does not produce an incorrect answer and so is a Las Vegas algorithm.

What remains is to establish that the running time of this algorithm is polynomial in both d and $\log p$. The polynomials $(x-b)^{(p-1)/2} \pm 1 \pmod{p}$ are of degree $(p-1)/2$, and computing them explicitly requires time exponential in $\log p$. However, these polynomials are themselves not important for the algorithm; only their gcds with $f(x)$ are needed. This means that one can compute the exponentiation $(x-b)^{(p-1)/2}$ modulo the polynomial $f(x)$. A standard square-and-multiply algorithm for this keeps the degrees of all intermediate polynomials bounded below $d = \deg f(x)$. Consequently, we obtain a running time polynomial in both d and $\log p$.

Example

As an illustration of the Las Vegas root-finding algorithm, take $p = 73$ and $f(x) = x^3 + 45x^2 + 5x + 28$. It is given that $f(x)$ is a product of three linear factors yielding three distinct roots. We have $x^{(p-1)/2} \equiv -1 \pmod{f(x)}$, that is, $f_1(x) = \gcd(x^{(p-1)/2} - 1, f(x)) = 1$ and $f_2(x) = \gcd(x^{(p-1)/2} + 1, f(x)) = f(x)$, that is, we fail to obtain a non-trivial split of $f(x)$, since all the roots of $f(x)$ are quadratic non-residues.

We try with random shifts of x by $x-b$. Let us first take $b = 38$. We have $(x-b)^{(p-1)/2} \equiv -1 \pmod{f(x)}$, that is, we again get $f_1(x) = 1$ and $f_2(x) = f(x)$.

Next, we try $b = 49$, for which $(x-b)^{(p-1)/2} \equiv 1 \pmod{f(x)}$, that is, now we get $f_1(x) = f(x)$ and $f_2(x) = 1$, that is, we still fail to split $f(x)$ non-trivially.

Trying $b = 23$ gives $(x-b)^{(p-1)/2} \equiv 40x^2 + 51x + 57 \pmod{f(x)}$. Now, taking gcd (modulo p) gives $f_1(x) = \gcd(40x^2 + 51x + 56, f(x)) = x^2 + 25x + 16$, that is, $f_2(x) =$

$f(x)/f_1(x) = x + 20$. Since $f_2(x)$ is linear, we immediately obtain the root $-20 \equiv 53 \pmod{p}$ of $f(x)$.

What remains is to split $f_1(x) = x^2 + 25x + 16$ non-trivially. To this end, we continue to choose random values of b. The choice $b = 56$ gives $x^{(p-1)/2} \equiv 21x + 7 \pmod{f_1(x)}$, and $\gcd(21x + 6, f_1(x)) = x + 42$ is a non-trivial factor of $f_1(x)$. This divulges the root $-42 \equiv 31 \pmod{p}$ of $f_1(x)$ (and so of $f(x)$ too). The third root of $f(x)$ is computed from the linear factor $f_1(x)/(x + 42) = x + 56$ as $-56 \equiv 17 \pmod{p}$.

25.2.2 A randomized approximation algorithm

Let $\phi(x_1, x_2, \ldots, x_n)$ be a Boolean formula in n variables $x_1, x_2, \ldots, x_n$. Suppose that ϕ is provided in CNF (conjunctive normal form, that is, AND of clauses). It is not necessary to assume that ϕ is in k-CNF for some $k \in \mathbb{N}$ (although there is no harm in making this assumption). Denote, by MAX-CNF-SAT, the problem of determining the maximum number of clauses that can be simultaneously satisfied by some truth assignment of the variables. If m is the number of clauses in ϕ, then this maximum value is m if and only if ϕ is satisfiable. We call MAX-CNF-SAT the *maximum satisfiability problem*.

MAX-CNF-SAT turns out to be a difficult optimization problem. Unless $\mathrm{P} = \mathrm{NP}$, this problem cannot be solved in polynomial time, since a polynomial-time algorithm for MAX-CNF-SAT clearly solves the problem CNF-SAT too in polynomial time. Suppose that for some input instance ϕ for MAX-CNF-SAT, the optimal solution is t^*. We have $t^* \leqslant m$. An obvious way to compute t^* is to evaluate all the clauses for all possible truth assignments of the variables $x_1, x_2, \ldots, x_n$. Since there are 2^n different truth assignments of the input variables, this exhaustive search may run in exponential time.

I am now going to supply a randomized algorithm for solving the maximum satisfiability problem. Let t be the output generated by this randomized algorithm. Since the algorithm is randomized, the output value t would vary over different executions of the algorithm. In other words, t can be treated as a random variable. What we concentrate upon is the expected value $\bar{t} = \mathrm{E}(t)$ of t. I will show that $\bar{t}/t^* \geqslant 1/2$. In that sense, our randomized algorithm is a randomized 1/2-approximation algorithm to solve the maximum satisfiability problem.

```
For each $i = 1, 2, \ldots, n$, take a random (uniform) truth assignment $T_i$ of $x_i$.
Evaluate the clauses $\phi_1, \phi_2, \ldots, \phi_m$ of $\phi$ for these truth assignments.
Return the number of clauses that evaluate to true.
```

It is hard to believe that this algorithm solves MAX-CNF-SAT in any sense at all. But it does. Let l be the number of literals in the clause ϕ_i. If ϕ_i contains both the literals x_j and $\bar{x}_j$, then it evaluates to 1, and we can remove the clause from ϕ. Likewise, we may assume that no literal is repeated in any clause of ϕ. This means that the l literals in ϕ_i belong to different variables. Since the variables are assigned random truth values, each literal is false with a probability of $1/2$. Since the variables are assigned truth values independently of one another, all of the l literals in ϕ_i evaluate to false with a probability of $(1/2)^l$. Therefore, the probability that ϕ_i evaluates to true is $1 - (1/2^l)$ which is $\geqslant 1/2$ since $l \geqslant 1$.

Let Z_i denote the random variable standing for the truth value of the clause ϕ_i, that is, $Z_i = 0$ or 1 according as whether ϕ_i evaluates to false or true. The number of true clauses in ϕ is $t = \sum_{i=1}^{m} Z_i$, and its expected value is

$$\bar{t} = \mathrm{E}(t) = \mathrm{E}\left(\sum_{i=1}^{m} Z_i\right) = \sum_{i=1}^{m} \mathrm{E}(Z_i) \geqslant \sum_{i=1}^{m} (1/2) = m/2.$$

On the other hand, $t^* \leqslant m$, so $\bar{t}/t^* \geqslant 1/2$, as desired. Notice that the random variables Z_i need not be independent of one another. The expectation of the sum of any (finite number of) random variables is always the sum of the expectations of the random variables.

25.2.3 Solving easy problems by randomized algorithms

Randomized algorithms are used, not necessarily for solving difficult (like NP-Complete) problems, but also for solving easy (like polynomial-time) problems. The use of randomization (for easy problems) may help us in many ways. First, we may achieve a reduction in the running time (over that achieved by deterministic algorithms). Second, the implementation of a randomized algorithm may be significantly simpler than the implementation of a deterministic algorithm for solving the same problem. Third, randomization may furnish provable statistical guarantees about the (expected) goodness in the performance of the algorithms.

Randomized quick sort

In the worst case, the quick-sort algorithm sorts an array of size n in $\Theta(n^2)$ time. However, in the best (or average) case, its performance is much better, namely $O(n \log n)$. When we talk about the average case of quick sort, we assume that the input array is a random permutation of the final sorted array. (For simplicity, assume that the elements in the input array are distinct.) Randomization helps us to come up with a version of quick sort that exhibits the average behavior irrespective of the distribution of elements in the input array. This implies that even if the input array is not random, we expect the randomized quick sort to run in $O(n \log n)$ time.

The performance of quick sort depends on the choice of the pivot. If the pivot always happens to be either the smallest or the largest element in the array, the partitioning is very skew. On the other hand, if the pivot is close to the median in the sorted array, then the partition is balanced, and we make recursive calls on two subarrays of sizes roughly half the size of the original array. Of course, we can use a linear-time median-finding algorithm to find the median to work as the pivot. But this algorithm is practically not very efficient.

In the randomized quick-sort algorithm, we choose the pivot as a random element in the array. That is, each of the n elements in the input array is equally likely to be chosen as the pivot. Let n_1 denote the number of elements in the array, that are smaller than the pivot, whereas n_2 the number of those larger than the pivot. The random choice of the pivot indicates that the n possible values $(0, n-1), (1, n-2), (2, n-3), \ldots, (n-1, 0)$ of the pair (n_1, n_2) are equiprobable (each having probability $1/n$). This, in turn, implies that the partitioning may be anything between the best and the worst, and we cannot straightaway apply an average-case analysis at this point.

Instead, we compute the expected running time $T(n)$ of randomized quick sort on an array of size n. The argument in the last paragraph leads to the following recurrence relation for $T(n)$.

$$\begin{aligned} T(n) &= \frac{1}{n}\Big[T(0)+T(n-1)\Big]+\frac{1}{n}\Big[T(1)+T(n-2)\Big]+\frac{1}{n}\Big[T(2)+T(n-3)\Big]+ \\ &\quad \cdots+\frac{1}{n}\Big[T(n-1)+T(0)\Big]+cn \\ &= \frac{2}{n}\Big[T(n-1)+T(n-2)+\cdots+T(1)+T(0)\Big]+cn, \end{aligned}$$

where the term cn stands for the time taken by the partitioning task. Since we never call quick sort on an empty array, let us take the boundary condition

$$T(0)=0.$$

Easy calculations show that

$$\frac{T(n)}{n+1}=\frac{T(n-1)}{n}+c\left[\frac{3}{n+1}-\frac{1}{n}\right].$$

Unfolding the recurrence yields the expression

$$T(n)=2c(n+1)H_n-3cn,$$

where H_n is the n-th harmonic number. But $\ln(n+1)\leqslant H_n\leqslant \ln n+1$, so $T(n)=\Theta(n\log n)$. This implies that although randomization fails to provide any guarantee about the goodness of each partitioning task, the expected running time of the randomized quick-sort algorithm is the best that a comparison-based sorting algorithm can achieve. Moreover, this guarantee pertains not just to some or most inputs (like random) but to all inputs (including those that force worst-case behavior of the deterministic quick-sort algorithm).

There is another way to derive the same expected running time. Let the array after sorting be $a_1,a_2,\ldots,a_n$. We take any two a_i and a_j with $1\leqslant i<j\leqslant n$, and denote by C_{ij} the random variable standing for whether a_i and a_j are compared during the sorting process. Indeed, $C_{ij}=1$ if a_i and a_j are compared. It is 0 otherwise. Then $C=\sum\limits_{i=1}^{n-1}\sum\limits_{j=i+1}^{n}C_{ij}$ is the random variable standing for the total number of comparisons made.

The running time of quicksort is dominated by the total cost of all partitions made during all the recursive calls. The time of each partitioning stage is tightly (big-Θ) bounded by the number of comparisons made during that stage. Therefore the overall running time is $\Theta(C)$. Consequently, it suffices to compute the expected value of C in order to obtain the expected running time of randomized quicksort. By linearity of expectation, we have

$$\mathrm{E}[C]=\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\mathrm{E}[C_{ij}].$$

To that end, we compute $\mathrm{E}[C_{ij}]=\Pr[C_{ij}=1]$.

Initially, all the elements of the array are together. Eventually, any two elements a_i and a_j are separated in two possible ways: (i) In some partitioning stage, a_i and a_j go to two different subarrays. (ii) In some partitioning stage, a_i or a_j is chosen as the pivot and goes to its correct position in the sorted array. After two elements are separated, no comparison is made between them. The elements a_i and a_j start together. So some partitioning stage separates them. During that partitioning stage, an element a_k, $i\leqslant k\leqslant j$, must be chosen as the pivot, so there are $j-i+1$ possibilities for the pivot. In quicksort, all comparisons are made with the pivots chosen at different partitioning stages. The partitioning stage that separates a_i from a_j compares a_i with a_j if and only if either a_i or a_j is chosen as the pivot. At no other time, a comparison between a_i and a_j is made. Since the elements are chosen as pivot with equal probability, we have

$$\Pr[C_{ij} = 1] = \frac{2}{j-i+1}.$$

It therefore follows that

$$\mathrm{E}[C] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathrm{E}[C_{ij}] = 2\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{1}{j-i+1} \leqslant 2\sum_{i=1}^{n-1} \sum_{j=1}^{n} \frac{1}{j} = 2nH_n = \Theta(n \log n).$$

Randomized min cut

Let $G = (V,E)$ be an undirected graph. A cut of G is a partition of V into two non-empty dusjoint subsets S,T. The size of the cut S,T, denoted $c(S,T)$, is the number of edges in G with one endpoint in S and with the other endpoint in T. The *minimum cut problem* deals with finding a cut S,T of G for which $c(S,T)$ is as small as possible. We use the notations $n = |V|$ and $m = |E|$. If G is not connected, then $c(S,T)$ is trivially 0. The connectivity in a graph can be efficiently checked by a traversal algorithm (BFS or DFS). So we assume that G is a connected graph.

By the max-flow-min-cut theorem, the minimum cut problem can be solved in polynomial time. Pick an aribatry vertex $s \in V$. Since S,T and T,S are essentially the same cut, we may assume that $s \in S^*$, where S^*,T^* is a minimal cut. We then compute the maximum s,t flow (with each edge weight equal to 1) for each $t \in V \setminus \{s\}$. The best-known algorithm for solving the maximum flow problem runs in $O(n^3)$ time. Therefore we obtain an $O(n^4)$-time deterministic algorithm for solving the minimum cut problem. Now, we introduce Monte Carlo algorithms for solving the same problem, and eventually arrive at a significantly better running time (albeit with some constant probability of failure).

An undirected *multigraph* is like an undirected graph with the exception that each edge can be present multiple times. An adjacency-matrix representation can store a multigraph, where the u,v-th entry in the matrix stores how many times the edge (u,v) appears in the multigraph. In the first randomized algorithm we discuss here, we start with the graph G, and make a sequence of edge contractions until only two vertices remain. Each edge contraction leads potentially to a multigraph.

Let $e = (u,v)$ be an edge (or multi-edge) in a graph (or multi-graph) G. Contracting e involves the following steps.

- Replace the two vertices u and v by a single vertex uv.
- Remove all the edges between u and v.
- Replace each edge (w,u) or (w,v) by an edge (w,uv).
- Keep all other edges as in G.

The resulting multigraph after this contraction is denoted by G/e.

With these notations, **Karger's min-cut algorithm** (proposed in 1993) can be stated as follows.

```
While $G$ contains more than two vertices, repeat:
    Pick a random (multi-)edge $(u,v)$ of $G$.
    Replace $G$ by $G/e$.
Return the two remaining vertices as the minimum cut.
The size of the cut is the number of edges between these vertices.
```

Figure 127: Illustration of Karger's min-cut algorithm

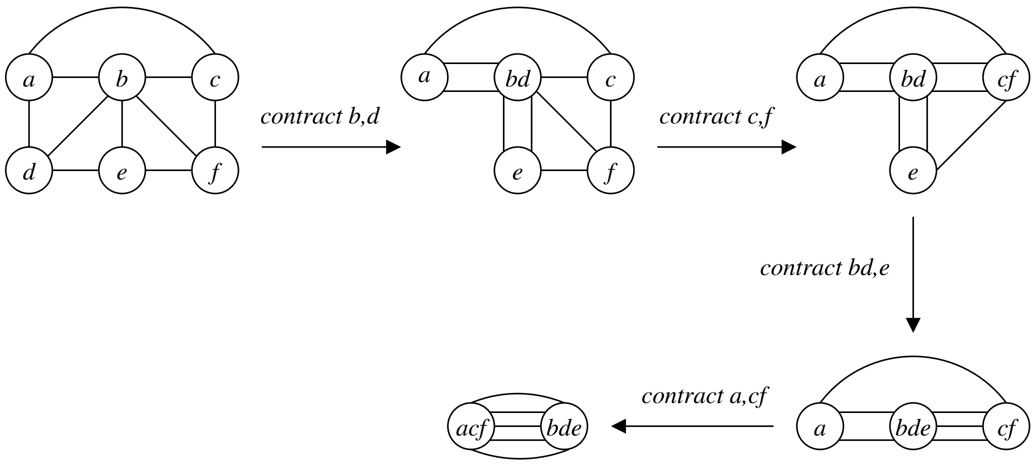


Figure 127 illustrates the working of Karger's algorithms. The multi-edges chosen for contraction are shown. After four contractions, the two vertices acf and bde remain. This corresponds to the cut with $S = \{a,c,f\}$ and $T = \{b,d,e\}$. The size of this cut is five, corresponding to the edges $(a,b),(a,d),(c,b),(f,b),(f,e)$ in the original graph. This solution is not optimal. For example, the cut $\{a\},\{b,c,d,e,f\}$ has size three.

The big question is therefore what the probability is for Karger's algorithm to produce a minimum cut. In order to estimate this probability, we note that by the degree-sum formula, the sum of the degrees of the n vertices of (the original graph) G is $2m$ (twice the number of edges). This implies that G contains at least one vertex v whose degree is $\leqslant 2m/n$. But then, $\{v\},V\setminus\{v\}$ is a cut of G of size $\leqslant 2m/n$. Therefore the minimum cut S^*,T^* of G has size $c^* \leqslant 2m/n$. G may contain multiple cuts with this size c^*, but in order to derive a lower bound on the probability of success of Karger's algorithm, let us assume that there is a unique cut of size c^*.

Karger's algorithm produces the cut S^*,T^* if and only if it never contracts an edge of this cut. In the first iteration, an edge is randomly chosen for contraction from the m available edges. Therefore the probability that neither of the c^* edges chosen is $1-\frac{c^*}{m} \geqslant 1-\frac{2m/n}{m} = 1-\frac{2}{n} = \frac{n-2}{n}$.

This analysis continues to hold after the contraction. In particular, after i edges not in the cut S^*,T^* are contracted, exactly $n-i$ vertices remain in G. In the $(i+1)$-st iteration, an edge not again in the optimal cut is chosen with probability $\geqslant 1-\frac{2}{n-i} = \frac{n-i-2}{n-i}$. Therefore the probability that Karger's algorithm produces the optimal cut is

$$\geqslant \left(\frac{n-2}{n}\right)\left(\frac{n-3}{n-1}\right)\left(\frac{n-4}{n-2}\right)\left(\frac{n-5}{n-3}\right)\cdots\left(\frac{2}{4}\right)\left(\frac{1}{3}\right) = \frac{2}{n(n-1)} \approx \frac{2}{n^2}.$$

This probability is low (for large n), but not too low. If we make $n^2/2$ independent runs of Karger's algorithm, then the probability that all these invocations fails to identify the minimum cut is

$$\leqslant \left(1 - \frac{2}{n^2}\right)^{n^2/2} \leqslant \frac{1}{e},$$

that is, the success probability is $\geqslant 1/e$. This implies that after $\frac{n^2}{2}\ln n$ independent runs of Karger's algorithm, the success probability is $\geqslant \left(\frac{1}{e}\right)^{\ln n} = \frac{1}{n}$, that is, we are almost certain to obtain the minimum cut after these many independent trials of Karger's algorithm.

We use the adjacency matrix representation of graphs and multigraphs. G always has $\leqslant n$ vertices. One can pick a random edge for contraction in $O(n)$ time (pick one endpoint randomly from the remaining vertices, and the other endpoint randomly from the adjacency list of the first endpoint). Subsequent contraction takes $O(n)$ time. Each invocation of Karger's algorithm makes $n-2$ contractions, yielding a running time of $O(n^2)$. Therefore the overall running time of Karger's algorithm with good success probability is $O(n^4 \log n)$. This is slightly poorer than the best flow-based algorithm for the same problem. But the flow-based algorithm is deterministic and has no probability of failure. Therefore we need to improve upon the running time of Karger's algorithm.

The **Karger–Stein algorithm** (proposed in 1996) achieves that. As more and more edges are contracted, the number of vertices reduces, and the probability of choosing an edge of the optimal cut increases in each iteration. In the last iteration, this probability is 2/3. The Karger–Stein algorithm keeps on contracting edges so long as the overall probability of avoiding these edges (starting from the first to the last iteration) remains $\geqslant 1/2$. If l vertices remain at this stage, this probability is

$$\geqslant \left(\frac{n-2}{n}\right)\left(\frac{n-3}{n-1}\right)\cdots\left(\frac{l-1}{l+1}\right) = \frac{l(l-1)}{n(n-1)} \approx \frac{l^2}{n^2}.$$

Now, $\frac{l^2}{n^2} \geqslant \frac{1}{2}$ implies $l = \frac{n}{\sqrt{2}}$, that is, the Karger–Stein algorithm contracts $n - \frac{n}{\sqrt{2}}$ edges. After this, the algorithm is called recursively. Two independent runs are made of this algorithm. The better (in terms of number of edges) of the two runs is output. Some authors run the contraction step only once, and make both the recursive calls on this graph.

```
KargerStein(G)
    If the number of vertices of $G$ is small,
        return a minimum cut by exhaustive search,
    else do the following:
        Contract $G$ to $n/\sqrt{2}$ vertices to get a multigraph $G_1$.
        Compute $S_1, T_1 =$ KargerStein($G_1$).
        Contract $G$ to $n/\sqrt{2}$ vertices to get a multigraph $G_2$.
        Compute $S_2, T_2 =$ KargerStein($G_2$).
        Return the better of the two cuts $S_1, T_1$ and $S_2, T_2$.
```

Let us now analyze the performance of the Karger–Stein algorithm. Let $P(n)$ be the success probability of the algorithm with G having n vertices. Since the initial contraction process avoids selecting an edge of the optimal cut with probability $\geqslant 1/2$, we have

$$P(n) \geqslant 1 - \left(1 - \frac{1}{2}P\left(\frac{n}{\sqrt{2}}\right)\right)^2.$$

This recurrence can be shown to satisfy $P(n) = \Omega(\frac{1}{\log n})$ (solve Exercise 6.89). Therefore $O(\log^2 n)$ independent runs of the Karger–Stein algorithm gives a success probablity $\geqslant 1 - \frac{1}{n}$.

Since each edge contraction can be done in $O(n)$ time, and there are $n - n/\sqrt{2}$ contractions performed initially, the running time $T(n)$ of the Karger–Stein algorithm satisfies the recurrence

$$T(n) = 2T\left(\frac{n}{\sqrt{2}}\right) + O(n^2).$$

By the master theorem, we have $T(n) = O(n^2 \log n)$. Therefore $O(\log^2 n)$ runs of the algorithm finds a minimum cut in $O(n^2 \log^3 n)$ time with a high success probability. This is much better than $O(n^4)$ or $O(n^4 \log n)$.

25.2.4 Randomized data structures

Like algorithms, we may think of data structures that are randomized. For example, think about inserting n elements in an (initially empty) binary search tree. If the input elements are inserted in a sorted sequence, the tree achieves a height of $n-1$. There are (many) other sequences that lead to this worst-case height of the tree. One possibility to avoid this problem is the use height-balanced trees (like AVL trees). The resulting implementation (for insertion) becomes somewhat involved.

We may think about a significantly simpler randomized strategy. For the time being, assume that all the n elements to be inserted in the tree are available to us from the beginning. (Exercise 4.197 handles an incremental construction.) We first make a random permutation of the input elements, and insert the elements in that permuted order. The random permutation may yield a sequence of insertions, that leads the tree to gain the worst-case height, but the probability of such an event is rather low. Indeed, using an analysis similar to the case of randomized quick sort, we can show that the expected height of the tree becomes $\Theta(\log n)$ (solve Exercise 6.107).

In this example, we may call the binary search tree randomized. Although this view can perhaps be upheld, one may argue that in this case we have actually randomized the insertion procedure, rather than the data structure. So let me present a better example.

Consider a hash table of size s with chaining, storing n elements. Assume that, for each input, the hash function produces an element from the set $\{0, 1, 2, \ldots, s-1\}$ with probability $1/s$. In that case, the expected length of each list is $\Theta(n/s)$ which is $\Theta(1)$ if $s = \Theta(n)$ (solve Exercise 6.108). The insertion of the elements may be carried out in an incremental way, that is, unlike binary search trees, it is not necessary to know all the elements beforehand and randomly permute them.

The above data structures are randomized in the Las Vegas sense, that is, they are expected to exhibit good performance irrespective of the choice of the input. When a subsequent search is carried out, we obtain the correct result with certainty.

It is also possible to conceive of data structures that are randomized in the Monte Carlo sense. This means that insertion and subsequent searching are necessarily efficient, but the output of a search request may be incorrect with some (small) probability of error. In what follows, I explain such a data structure known as the *Bloom filter* (designed by Burton H. Bloom in 1970). Bloom filters are often used as compressed storage of data.

A bloom filer consists of a bit array A of size s. Initially, all the bits of A are set to 0. We use k random and independent hash functions $h_0, h_1, \ldots, h_{k-1}$ each producing an integer output between 0 and $s-1$ (both inclusive).

A bloom filter allows only insertion and searching. In order to insert an element x in the bloom filter, we first compute the k hash values $y_i = h_i(x)$ for $i = 0, 1, \ldots, k-1$. We then set the bits in A at positions $y_0, y_1, \ldots, y_{k-1}$.

In order to search for an element u in the filter, we compute the k hash values $v_i = h_i(u)$ for $i = 0, 1, \ldots, k-1$. If any of the bits at positions $v_0, v_1, \ldots, v_{k-1}$ in A is 0, we output *No* (that is, u is not present). If all these positions in A hold the bit 1, we output *Yes* (u is present).

When this search algorithm outputs *No*, it is definitely correct. However, an output *Yes* does not necessarily imply that u has been inserted in the filter. All the bits v_i might have been set during insertions of other elements in the filter. Such a u is often called a *false positive*. By suitably selecting s (the size of the bit table) and n (the number of elements inserted in the filter), we can make the probability of false positives as low as we desire.

Example

Suppose we want to store 3-digit integers in a bloom filter. Take a bit array A of size $s = 16$. For an input $x = (x_2x_1x_0)_{10}$ (in base 10), we compute the hash values $h_0(x) = (3^{x_0} \pmod{17}) - 1$, $h_1(x) = (5^{x_1} \pmod{17}) - 1$, and $h_2(x) = (7^{x_2} \pmod{17}) - 1$. Initially, the filter does not contain any element, so the bit array A is:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Now, we insert 825 in the filter. We have $h_0(825) = 4$, $h_1(825) = 7$ and $h_2(825) = 15$. So the insertion of 825 updates A as:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	1

Then, we insert 357 for which the hash values are 10, 13 and 2, so A changes to:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	1	0	1	0	0	1	0	0	1	0	0	1	0	1

Finally, let us insert 471 with hash values $2, 9, 3$. The bit at position 2 in A is already set. Setting the bits at positions 3 and 9 leaves A as:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	1	1	1	0	0	1	0	1	1	0	0	1	0	1

Let us now see how we can search in the filter. Let us first search for 789. Applications of the hash functions give the indices 13, 15 and 11. The bits at indices 13 and 15 are set, but the bit at index 11 is not set, implying that 789 has not been inserted in the filter.

Then, we search for 513. The hash values in this case are 9, 4 and 10. The bits of A at all these three positions are set, so we output *Yes*. This is a false positive, since 513 has not been inserted in the filter. The bits at indices 4, 9 and 10 were set during the insertions of 825, 471 and 357, respectively.

Let me now make an analysis of the probability of false-positive errors. Let s be the size of the bit table, k the number of hash functions, and n the number of elements inserted in the filter. During each insertion, each bit in A is set by each hash function with probability $1/s$. Therefore, the probability that a bit in A is set (at least once) during n insertions is

$$1-\left(1-\frac{1}{s}\right)^{kn}.$$

Now, consider an element u not inserted in the filter is searched. The k hash functions produce k uniformly random locations in A, each containing a 1-bit with the above probability. Therefore, the probability that u is a false positive is

$$\left(1-\left(1-\frac{1}{s}\right)^{kn}\right)^{k} \approx \left(1-e^{-kn/s}\right)^{k}.$$

This probability is minimized with respect to k for

$$k \approx \frac{s}{n}\ln 2,$$

and the corresponding false positive probability is about

$$2^{-k} \approx (2^{-\ln 2})^{s/n} = p,$$

where p is the allowed error probability suitable for an application. This lets us choose the size s of the bit table A as

$$s \approx -\frac{n\ln p}{\ln^2 2}.$$

25.3 Backtracking and branch-and-bound algorithms

A non-deterministic algorithm is associated with a computation tree for every instance of input. Branching in the tree represents non-deterministic choices. Each leaf in the tree is marked by *Yes* or *No* in order to indicate the decision of the unique computation path from the root to that leaf. A systematic traversal of this computation tree visits all the leaves of the tree and lets us conclude about the decision on the given input instance. Indeed, this is precisely how we emulated a non-deterministic algorithm by a deterministic one (See Chapter 24). This emulation may lead to exponential running time (and even exponential space), but may be the only known way to solve certain problems in NP. Use of suitable heuristic strategies during the traversal may often curtail the search considerably, achieving significant practical speedup.

25.3.1 Backtracking

I now elaborate this idea formally. Each node in the computation tree is uniquely specified by the path from the root to that node, that is, by the non-deterministic choices made so far. We represent this information by a string C. In addition to C, we may need to maintain some other information D associated with the current sequence of choices. Against every node in the computation tree, we maintain the pair (C,D). Initially, no choices are made, so C is the empty string ε, and D_{init} represents the initial bookkeeping information.