

Randomized algorithm

From Wikipedia, the free encyclopedia

Jump to: [navigation](#), [search](#)

"Randomized algorithms" redirects here. It is not to be confused with [Algorithmic randomness](#).

A **randomized algorithm** is an [algorithm](#) that employs a degree of [randomness](#) as part of its logic. The algorithm typically uses [uniformly random](#) bits as an auxiliary input to guide its behavior, in the hope of achieving good performance in the "average case" over all possible choices of random bits. Formally, the algorithm's performance will be a [random variable](#) determined by the random bits; thus either the running time, or the output (or both) are random variables.

One has to distinguish between algorithms that use the random input so that they always terminate with the correct answer, but where the expected running time is finite ([Las Vegas algorithms](#), example of which is [Quicksort](#)^[1]), and algorithms which have a chance of producing an incorrect result ([Monte Carlo algorithms](#), example of which is [Monte Carlo algorithm](#) for [MFAS](#)^[2]) or fail to produce a result either by signaling a failure or failing to terminate.

In the second case, random performance and random output, the term "algorithm" for a procedure is somewhat questionable. In the case of random output, it is no longer formally [effective](#).^[3] However, in some cases, probabilistic algorithms are the only practical means of solving a problem.^[4]

In common practice, randomized algorithms are approximated using a [pseudorandom number generator](#) in place of a true source of random bits; such an implementation may deviate from the expected theoretical behavior.

Contents

- [1 Motivation](#)
- [2 Computational complexity](#)
- [3 History](#)
- [4 Examples](#)
 - [4.1 Quicksort](#)

Part of [a series](#) on
**Probabilistic
data structures**

- [Bloom filter](#)
- [Count–min sketch](#)
- [Quotient filter](#)
- [Skip list](#)

[Random trees](#)

- [Random binary tree](#)
- [Treap](#)
- [Rapidly exploring random tree](#)

Related

- [Randomized algorithm](#)

 **[Computer science portal](#)**

-
-
-

[v](#)
[t](#)
[e](#)

- [4.2 Randomized incremental constructions in geometry](#)
- [4.3 Min cut](#)
- [5 Derandomization](#)
- [6 Where randomness helps](#)
- [7 See also](#)
- [8 Notes](#)
- [9 References](#)

Motivation[[edit](#)]

As a motivating example, consider the problem of finding an ‘*a*’ in an [array](#) of *n* elements.

Input: An array of $n \geq 2$ elements, in which half are ‘*a*’s and the other half are ‘*b*’s.

Output: Find an ‘*a*’ in the array.

We give two versions of the algorithm, one [Las Vegas algorithm](#) and one [Monte Carlo algorithm](#).

Las Vegas algorithm:

```
findingA_LV(array A, n)
begin
  repeat
    Randomly select one element out of n elements.
  until 'a' is found
end
```

This algorithm succeeds with probability 1. The number of iterations varies and can be arbitrarily large, but the expected number of iterations is

$$\lim_{n \rightarrow \infty} \sum_{i=1}^n \frac{i}{2^i} = 2$$

Since it is constant the expected run time over many calls is $\Theta(1)$. (See [Big O notation](#))

Monte Carlo algorithm:

```
findingA_MC(array A, n, k)
begin
  i=0
  repeat
    Randomly select one element out of n elements.
    i = i + 1
  until i=k or 'a' is found
end
```

If an ‘*a*’ is found, the algorithm succeeds, else the algorithm fails. After *k* iterations, the probability of finding an ‘*a*’ is:

$$\Pr(\text{find } a) = 1 - \left(\frac{1}{2}\right)^k$$

This algorithm does not guarantee success, but the run time is bounded. The number of iterations is always less than or equal to k . Taking k to be constant the run time (expected and absolute) is $\Theta(1)$.

Randomized algorithms are particularly useful when faced with a malicious "adversary" or [attacker](#) who deliberately tries to feed a bad input to the algorithm (see [worst-case complexity](#) and [competitive analysis \(online algorithm\)](#)) such as in the [Prisoner's dilemma](#). It is for this reason that [randomness](#) is ubiquitous in [cryptography](#). In cryptographic applications, pseudo-random numbers cannot be used, since the adversary can predict them, making the algorithm effectively deterministic. Therefore, either a source of truly random numbers or a [cryptographically secure pseudo-random number generator](#) is required. Another area in which randomness is inherent is [quantum computing](#).

In the example above, the Las Vegas algorithm always outputs the correct answer, but its running time is a random variable. The Monte Carlo algorithm (related to the [Monte Carlo method](#) for simulation) is guaranteed to complete in an amount of time that can be bounded by a function the input size and its parameter k , but allows a *small probability of error*. Observe that any Las Vegas algorithm can be converted into a Monte Carlo algorithm (via [Markov's inequality](#)), by having it output an arbitrary, possibly incorrect answer if it fails to complete within a specified time. Conversely, if an efficient verification procedure exists to check whether an answer is correct, then a Monte Carlo algorithm can be converted into a Las Vegas algorithm by running the Monte Carlo algorithm repeatedly till a correct answer is obtained.

Computational complexity[\[edit\]](#)

[Computational complexity theory](#) models randomized algorithms as [probabilistic Turing machines](#). Both [Las Vegas](#) and [Monte Carlo algorithms](#) are considered, and several [complexity classes](#) are studied. The most basic randomized complexity class is [RP](#), which is the class of [decision problems](#) for which there is an efficient (polynomial time) randomized algorithm (or probabilistic Turing machine) which recognizes NO-instances with absolute certainty and recognizes YES-instances with a probability of at least 1/2. The complement class for RP is co-RP. Problem classes having (possibly nonterminating) algorithms with [polynomial time](#) average case running time whose output is always correct are said to be in [ZPP](#).

The class of problems for which both YES and NO-instances are allowed to be identified with some error is called [BPP](#). This class acts as the randomized equivalent of [P](#), i.e. BPP represents the class of efficient randomized algorithms.

History[\[edit\]](#)

Historically, the first randomized algorithm was a method developed by [Michael O. Rabin](#) for the [closest pair problem](#) in [computational geometry](#).^[5] The study of randomized algorithms was spurred by the 1977 discovery of a [randomized primality test](#) (i.e., determining the [primality](#) of a number) by [Robert M. Solovay](#) and [Volker Strassen](#). Soon afterwards Michael O. Rabin demonstrated that the 1976 [Miller's primality test](#) can be turned into a randomized algorithm. At that time, no practical [deterministic algorithm](#) for primality was known.

The Miller-Rabin primality test relies on a binary relation between two positive integers k and n that can be expressed by saying that k "is a witness to the compositeness of" n . It can be shown that

- If there is a witness to the compositeness of n , then n is composite (i.e., n is not [prime](#)), and
- If n is composite then at least three-fourths of the natural numbers less than n are witnesses to its compositeness, and
- There is a fast algorithm that, given k and n , ascertains whether k is a witness to the compositeness of n .

Observe that this implies that the primality problem is in Co-[RP](#).

If one [randomly](#) chooses 100 numbers less than a composite number n , then the probability of failing to find such a "witness" is $(1/4)^{100}$ so that for most practical purposes, this is a good primality test. If n is big, there may be no other test that is practical. The probability of error can be reduced to an arbitrary degree by performing enough independent tests.

Therefore, in practice, there is no penalty associated with accepting a small probability of error, since with a little care the probability of error can be made astronomically small. Indeed, even though a deterministic polynomial-time primality test has since been found (see [AKS primality test](#)), it has not replaced the older probabilistic tests in [cryptographic software](#) nor is it expected to do so for the foreseeable future.

Examples[\[edit\]](#)

Quicksort[\[edit\]](#)

[Quicksort](#) is a familiar, commonly used algorithm in which randomness can be useful. Any deterministic version of this algorithm requires $\mathcal{O}(n^2)$ time to sort n numbers for some well-defined class of degenerate inputs (such as an already sorted array), with the specific class of inputs that generate this behavior defined by the protocol for pivot selection. However, if the algorithm selects pivot elements uniformly at random, it has a provably high probability of finishing in $\mathcal{O}(n \log n)$ time regardless of the characteristics of the input.

Randomized incremental constructions in geometry[\[edit\]](#)

In [computational geometry](#), a standard technique to build a structure like a [convex hull](#) or [Delaunay triangulation](#) is to randomly permute the input points and then insert them one by one into the existing structure. The randomization ensures that the expected number of changes to the structure caused by an insertion is small, and so the expected running time of the algorithm can be upper bounded. This technique is known as [randomized incremental construction](#).^[6]

Min cut[\[edit\]](#)

Main article: [Karger's algorithm](#)

Input: A [graph](#) $G(V,E)$

Output: A [cut](#) partitioning the vertices into L and R , with the minimum number of edges between L and R .

Recall that the [contraction](#) of two nodes, u and v , in a (multi-)graph yields a new node u' with edges that are the union of the edges incident on either u or v , except from any edge(s) connecting u and v . Figure 1 gives an example of contraction of vertex A and B . After contraction, the resulting graph may have parallel edges, but contains no self loops.



Figure 2: Successful run of Karger's algorithm on a 10-vertex graph. The minimum cut has size 3 and is indicated by the vertex colours.

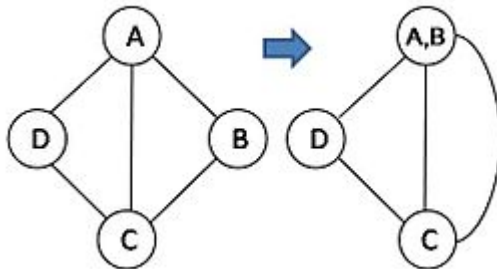


Figure 1: Contraction of vertex A and B

Karger's^[7] basic algorithm:

```

begin
   $i=1$ 
  repeat
    repeat
      Take a random edge  $(u,v) \in E$  in  $G$ 
      replace  $u$  and  $v$  with the contraction  $u'$ 
    until only 2 nodes remain
    obtain the corresponding cut result  $C_i$ 
     $i=i+1$ 
  until  $i=m$ 
  output the minimum cut among  $C_1, C_2, \dots, C_m$ .
end

```

In each execution of the outer loop, the algorithm repeats the inner loop until only 2 nodes remain, the corresponding cut is obtained. The run time of one execution is $O(n)$, and n denotes the number of vertices. After m times executions of the outer loop, we output the minimum cut among all the results. The figure 2 gives an example of one execution of the algorithm. After execution, we get a cut of size 3.

Lemma 1: Let k be the min cut size, and let $C = \{e_1, e_2, \dots, e_k\}$ be the min cut. If, during iteration i , no edge $e \in C$ is selected for contraction, then $C_i = C$.

Proof: If G is not connected, then G can be partitioned into L and R without any edge between them. So the min cut in a disconnected graph is 0. Now, assume G is connected. Let $V=L \cup R$ be the partition of V induced by C : $C = \{ \{u,v\} \in E : u \in L, v \in R \}$ (well-defined since G is connected). Consider an edge $\{u,v\}$ of C . Initially, u,v are distinct vertices. *As long as we pick an edge $\{u,v\}$, u and v do not get merged.* Thus, at the end of the algorithm, we have two compound nodes covering the entire

graph, one consisting of the vertices of L and the other consisting of the vertices of R . As in figure 2, the size of min cut is 1, and $C = \{(A,B)\}$. If we don't select (A,B) for contraction, we can get the min cut.

Lemma 2: If G is a multigraph with p vertices and whose min cut has size k , then G has at least $pk/2$ edges.

Proof: Because the min cut is k , every vertex v must satisfy $\text{degree}(v) \geq k$. Therefore, the sum of the degree is at least pk . But it is well known that the sum of vertex degrees equals $2|E|$. The lemma follows.

Analysis of algorithm

The probability that the algorithm succeeds is $1 -$ the probability that all attempts fail. By independence, the probability that all attempts fail is

$$\prod_{i=1}^m \Pr(C_i \neq C) = \prod_{i=1}^m (1 - \Pr(C_i = C)).$$

By lemma 1, the probability that $C_i = C$ is the probability that no edge of C is selected during iteration i . Consider the inner loop and let G_j denote the graph after j edge contractions, where $j \in \{0, 1, \dots, n-3\}$. G_j has $n-j$ vertices. We use the chain rule of [conditional possibilities](#). The probability that the edge chosen at iteration j is not in C , given that no edge of C has been chosen

before, is $\frac{1 - \frac{|C|}{|E(G_j)|}}{1 - \frac{|C|}{|E(G_j)|}}$. Note that G_j still has min cut of size k , so by Lemma 2, it still has at least

$$\frac{k}{2} \text{ edges.}$$

Thus, $\frac{1 - \frac{|C|}{|E(G_j)|}}{1 - \frac{|C|}{|E(G_j)|}} \geq \frac{1 - \frac{k}{2(n-j)}}{1 - \frac{k}{2(n-j)}}$.

So by the chain rule, the probability of finding the min cut C is

$$\Pr[C_i = C] \geq \left(\frac{n-2}{n}\right) \left(\frac{n-3}{n-1}\right) \left(\frac{n-4}{n-2}\right) \cdots \left(\frac{3}{5}\right) \left(\frac{2}{4}\right) \left(\frac{1}{3}\right).$$

Cancellation gives $\Pr[C_i = C] \geq \frac{2}{n(n-1)}$. Thus the probability that the algorithm succeeds is at

least $1 - \left(1 - \frac{2}{n(n-1)}\right)^m$. For $m = \frac{n(n-1)}{2 \ln n}$, this is equivalent to $1 - \frac{1}{e}$. The algorithm

finds the min cut with probability $1 - \frac{1}{e}$, in time $O(mn) = O(n^3 \log n)$.

Derandomization[[edit](#)]

Randomness can be viewed as a resource, like space and time. Derandomization is then the process of *removing* randomness (or using as little of it as possible). From the viewpoint of [computational](#)

[complexity](#), derandomizing an efficient randomized algorithm is the question "is $P = BPP$?"^{[*[citation needed](#)*]}

There are specific methods that can be employed to derandomize particular randomized algorithms:

- the [method of conditional probabilities](#), and its generalization, [pessimistic estimators](#)
- [discrepancy theory](#) (which is used to derandomize geometric algorithms)
- the exploitation of limited independence in the random variables used by the algorithm, such as the [pairwise independence](#) used in [universal hashing](#)
- the use of [expander graphs](#) (or [dispersers](#) in general) to *amplify* a limited amount of initial randomness (this last approach is also referred to as generating [pseudorandom](#) bits from a random source, and leads to the related topic of pseudorandomness)

Where randomness helps^{[[edit](#)]}

When the model of computation is restricted to [Turing machines](#), it is currently an open question whether the ability to make random choices allows some problems to be solved in polynomial time that cannot be solved in polynomial time without this ability; this is the question of whether $P = BPP$. However, in other contexts, there are specific examples of problems where randomization yields strict improvements.

- Based on the initial motivating example: given an exponentially long string of 2^k characters, half a's and half b's, a [random access machine](#) requires at least 2^{k-1} lookups in the worst-case to find the index of an *a*; if it is permitted to make random choices, it can solve this problem in an expected polynomial number of lookups.
- The natural way of carrying out a numerical computation in [embedded systems](#) or [cyber-physical systems](#) is to provide a result that approximates the correct one with high probability (or Probably Approximately Correct Computation (PACC)). The hard problem associated with the evaluation of the discrepancy loss between the approximated and the correct computation can be effectively addressed by resorting to randomization^[8]
- In [communication complexity](#), the equality of two strings can be verified to some reliability using $\Theta(n)$ bits of communication with a randomized protocol. Any deterministic protocol requires $\Theta(n)$ bits if defending against a strong opponent.^[9]
- The volume of a convex body can be estimated by a randomized algorithm to arbitrary precision in polynomial time.^[10] [Bárány](#) and [Füredi](#) showed that no deterministic algorithm can do the same.^[11] This is true unconditionally, i.e. without relying on any complexity-theoretic assumptions, assuming the convex body can be queried only as a black box.
- A more complexity-theoretic example of a place where randomness appears to help is the class [IP](#). IP consists of all languages that can be accepted (with high probability) by a polynomially long interaction between an all-powerful prover and a verifier that implements a BPP algorithm. $IP = PSPACE$.^[12] However, if it is required that the verifier be deterministic, then $IP = NP$.
- In a [chemical reaction network](#) (a finite set of reactions like $A+B \rightarrow 2C + D$ operating on a finite number of molecules), the ability to ever reach a given target state from an initial state is decidable, while even approximating the probability of ever reaching a given target state (using the standard concentration-based probability for which reaction will occur next) is undecidable. More specifically, a limited Turing machine can be simulated with arbitrarily high probability of running correctly for all time, only if a random chemical reaction network is used. With a

simple nondeterministic chemical reaction network (any possible reaction can happen next), the computational power is limited to [primitive recursive functions](#).^[13]

See also^{[[edit](#)]}

- [Probabilistic analysis of algorithms](#)
- [Atlantic City algorithm](#)
- [Monte Carlo algorithm](#)
- [Las Vegas algorithm](#)
- [Principle of deferred decision](#)

Notes^{[[edit](#)]}

- ↑ Hoare, C. A. R. (July 1961). "[Algorithm 64: Quicksort](#)". *Commun. ACM*. **4** (7): 321–. [doi:10.1145/366622.366644](#). [ISSN 0001-0782](#).
- ↑ Kudelić, Robert (2016-04-01). "[Monte-Carlo randomized algorithm for minimal feedback arc set problem](#)". *Applied Soft Computing*. **41**: 235–246. [doi:10.1016/j.asoc.2015.12.018](#).
- ↑ "Probabilistic algorithms should not be mistaken with methods (which I refuse to call algorithms), which produce a result which has a high probability of being correct. It is essential that an algorithm produces correct results (discounting human or computer errors), even if this happens after a very long time." Henri Cohen (2000). *A Course in Computational Algebraic Number Theory*. Springer-Verlag, p. 2.
- ↑ "In [testing primality](#) of very large numbers chosen at random, the chance of stumbling upon a value that fools the [Fermat test](#) is less than the chance that [cosmic radiation](#) will cause the computer to make an error in carrying out a 'correct' algorithm. Considering an algorithm to be inadequate for the first reason but not for the second illustrates the difference between mathematics and engineering." Hal Abelson and Gerald J. Sussman (1996). *Structure and Interpretation of Computer Programs*. MIT Press, [section 1.2](#).
- ↑ Smid, Michiel. Closest point problems in computational geometry. Max-Planck-Institut für Informatik|year=1995
- ↑ Seidel R. [Backwards Analysis of Randomized Geometric Algorithms](#).
- ↑ A. A. Tsay, W. S. Lovejoy, David R. Karger, *Random Sampling in Cut, Flow, and Network Design Problems*, Mathematics of Operations Research, 24(2):383–413, 1999.
- ↑ Alippi, Cesare (2014), *Intelligence for Embedded Systems*, Springer, [ISBN 978-3-319-05278-6](#).
- ↑ Kushilevitz, Eyal; Nisan, Noam (2006), *Communication Complexity*, Cambridge University Press, [ISBN 9780521029834](#). For the deterministic lower bound see p. 11; for the logarithmic randomized upper bound see pp. 31–32.
- ↑ Dyer, M.; Frieze, A.; Kannan, R. (1991), "[A random polynomial-time algorithm for approximating the volume of convex bodies](#)", *Journal of the ACM*, **38** (1): 1–17, [doi:10.1145/102782.102783](#)
- ↑ Füredi, Z.; Bárány, I. (1986), "Computing the volume is difficult", *Proc. 18th ACM Symposium on Theory of Computing* (Berkeley, California, May 28–30, 1986), New York, NY: ACM, pp. 442–447, [doi:10.1145/12130.12176](#), [ISBN 0-89791-193-8](#)
- ↑ Shamir, A. (1992), "IP = PSPACE", *Journal of the ACM*, **39** (4): 869–877, [doi:10.1145/146585.146609](#)
- ↑ Cook, Matthew; Soloveichik, David; Winfree, Erik; Bruck, Jehoshua (2009), "Programmability of chemical reaction networks", in Condon, Anne; Harel, David; Kok, Joost

N.; *Salomaa, Arto*; *Winfree, Erik*, *Algorithmic Bioprocesses*, *Natural Computing Series*, Springer-Verlag, pp. 543–584, doi:[10.1007/978-3-540-88869-7_27](https://doi.org/10.1007/978-3-540-88869-7_27).

References[[edit](#)]

- [Thomas H. Cormen](#), [Charles E. Leiserson](#), [Ronald L. Rivest](#), and [Clifford Stein](#). *Introduction to Algorithms*, Second Edition. MIT Press and McGraw–Hill, 1990. ISBN 0-262-03293-7. Chapter 5: Probabilistic Analysis and Randomized Algorithms, pp. 91–122.
- Dirk Draheim. "*Semantics of the Probabilistic Typed Lambda Calculus (Markov Chain Semantics, Termination Behavior, and Denotational Semantics)*." Springer, 2017.
- [Jon Kleinberg](#) and [Éva Tardos](#). *Algorithm Design*. Chapter 13: "Randomized algorithms".
- Don Fallis. 2000. "[The Reliability of Randomized Algorithms](#)." *British Journal for the Philosophy of Science* 51:255–71.
- [M. Mitzenmacher](#) and [E. Upfal](#). *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, New York (NY), 2005.
- [Rajeev Motwani](#) and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, New York (NY), 1995.
- Rajeev Motwani and P. Raghavan. [Randomized Algorithms](#). A survey on Randomized Algorithms.
- Christos Papadimitriou* (1993), *Computational Complexity (1st ed.)*, Addison Wesley, ISBN 0-201-53082-1 Chapter 11: Randomized computation, pp. 241–278.
- M. O. Rabin (1980), *Probabilistic Algorithm for Testing Primality*, *Journal of Number Theory*, 12:128–38.
- A. A. Tsay, W. S. Lovejoy, David R. Karger, *Random Sampling in Cut, Flow, and Network Design Problems*, *Mathematics of Operations Research*, 24(2):383–413, 1999.

Retrieved from "https://en.wikipedia.org/w/index.php?title=Randomized_algorithm&oldid=822901723"

Categories:

- [Randomized algorithms](#)
- [Analysis of algorithms](#)

Hidden categories:

- [All articles with unsourced statements](#)
- [Articles with unsourced statements from October 2017](#)

Navigation menu

Personal tools

- Not logged in
- [Talk](#)
- [Contributions](#)
- [Create account](#)
- [Log in](#)

Namespaces

- [Article](#)
- [Talk](#)



Variants

Views

- [Read](#)
- [Edit](#)
- [View history](#)



More

Search

<input type="text" value="Search Wikipedia"/>	<input type="button" value="Go"/>
---	-----------------------------------

Navigation

- [Main page](#)
- [Contents](#)
- [Featured content](#)
- [Current events](#)
- [Random article](#)
- [Donate to Wikipedia](#)
- [Wikipedia store](#)

Interaction

- [Help](#)
- [About Wikipedia](#)
- [Community portal](#)
- [Recent changes](#)
- [Contact page](#)

Tools

- [What links here](#)
- [Related changes](#)
- [Upload file](#)
- [Special pages](#)

- [Permanent link](#)
- [Page information](#)
- [Wikidata item](#)
- [Cite this page](#)

Print/export

- [Create a book](#)
- [Download as PDF](#)
- [Printable version](#)



Languages

- [العربية](#)
- [বাংলা](#)
- [Deutsch](#)
- [Español](#)
- [Français](#)
- [한국어](#)
- [Português](#)
- [Русский](#)
- [中文](#)

10 more

[Edit links](#)

- This page was last edited on 29 January 2018, at 04:09.
- Text is available under the [Creative Commons Attribution-ShareAlike License](#); additional terms may apply. By using this site, you agree to the [Terms of Use](#) and [Privacy Policy](#). Wikipedia® is a registered trademark of the [Wikimedia Foundation, Inc.](#), a non-profit organization.

- [Privacy policy](#)
- [About Wikipedia](#)
- [Disclaimers](#)
- [Contact Wikipedia](#)
- [Developers](#)
- [Cookie statement](#)
- [Mobile view](#)
- [Enable previews](#)

- 
- 