

# Randomized Algorithms

**Dr. Bibhudatta Sahoo**

Communication & Computing Group

**Department of CSE, NIT Rourkela**

Email: [bdsahu@nitrkl.ac.in](mailto:bdsahu@nitrkl.ac.in), 9937324437, 2462358

# Problem Taxonomy

1. **Un-decidable Problem:** Do not have algorithms of any known complexity (polynomial or super-polynomial)
2. **Intractable problem:** Have algorithms with super polynomial time complexity.
3. **Tractable problem:** Have good algorithms with polynomial time complexity (irrespective of the degree)

## Different type of decidable Problem

[1] **Decision Problems:** The class of problems, the output is either yes or no.

**Example :** Whether a given number is prime?

[2] **Counting Problem:** The class of problem, the output is a natural number

**Example :** How many distinct factor are there for a given number

[3] **Optimization Problem:** The class of problem with some objective function based on the problem instance

**Example :** Finding a minimal spanning tree for a weighted graph

# Three general categories of intractable problems

1. Problems for which **polynomial-time algorithms** have been found
2. Problems that have been proven to be **intractable**
3. Problems that have not been proven to be intractable, but for which polynomial-time algorithms have never been found .

It is a surprising phenomenon that most problems in computer science seem to fall into either the **first** or **third** category

- Easy = best known solution is efficient
- Hard = best known solution is NOT efficient
- Efficient algorithm: a reasonably large problem can be solved in a reasonable amount of time

## Methods adopted to cope with NP-complete Problem

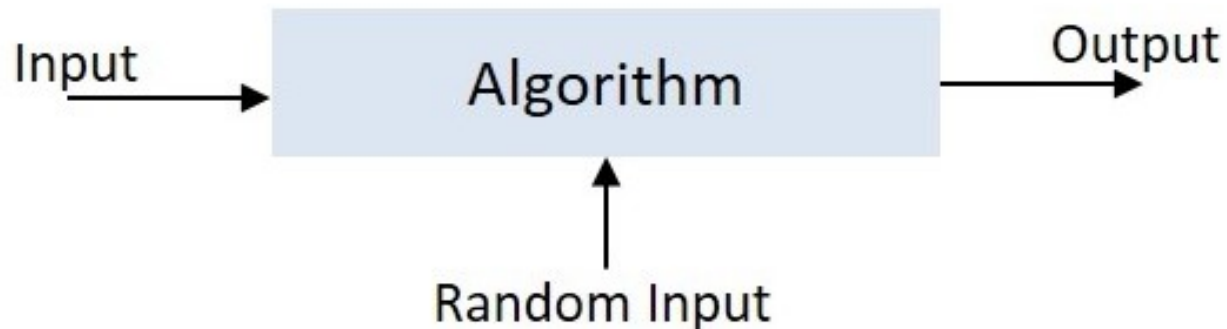
- Use dynamic programming, backtracking or branch-and-bound technique to reduce the computational cost. This might work if the problem size is not too big.
- Find the sub-problem of the original problem that have polynomial time solution.
- Use **approximation algorithms** to find approximate solution in polynomial time.
- Use **randomized algorithm** to find solutions is **affordable time** with a high probability of correctness of the solution.
- Use **heuristic** like greedy method, simulated annealing or genetic algorithms etc. However, the solutions produced cannot be guaranteed to be within a certain distance from the optimal solution.

# Introduction

- An algorithm that uses random numbers to decide what to do next anywhere in its logic is called a Randomized Algorithm.

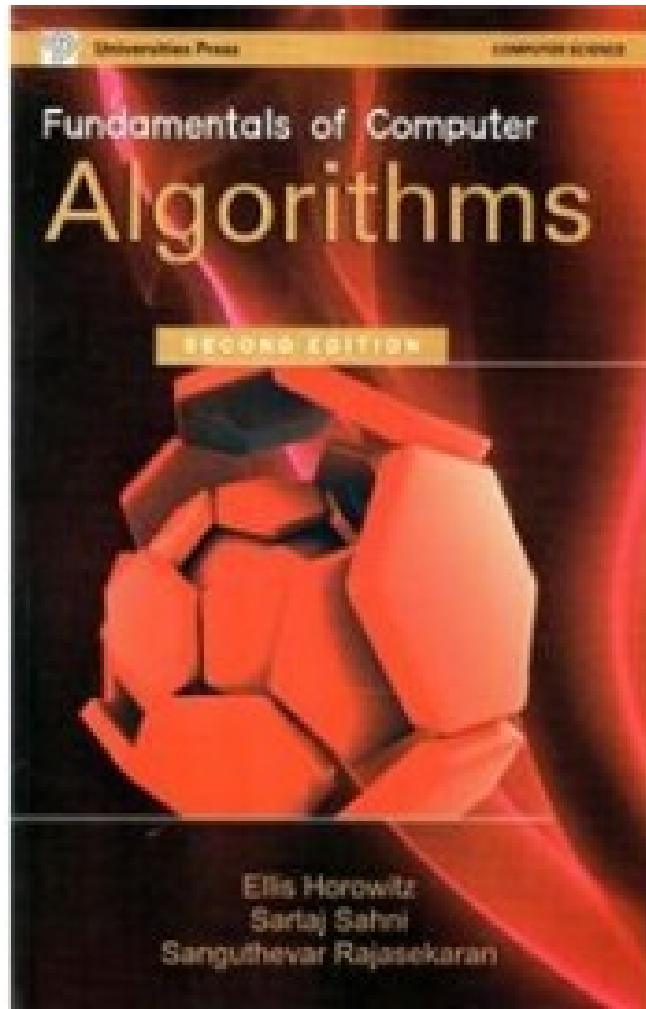


(a) Deterministic algorithm structure



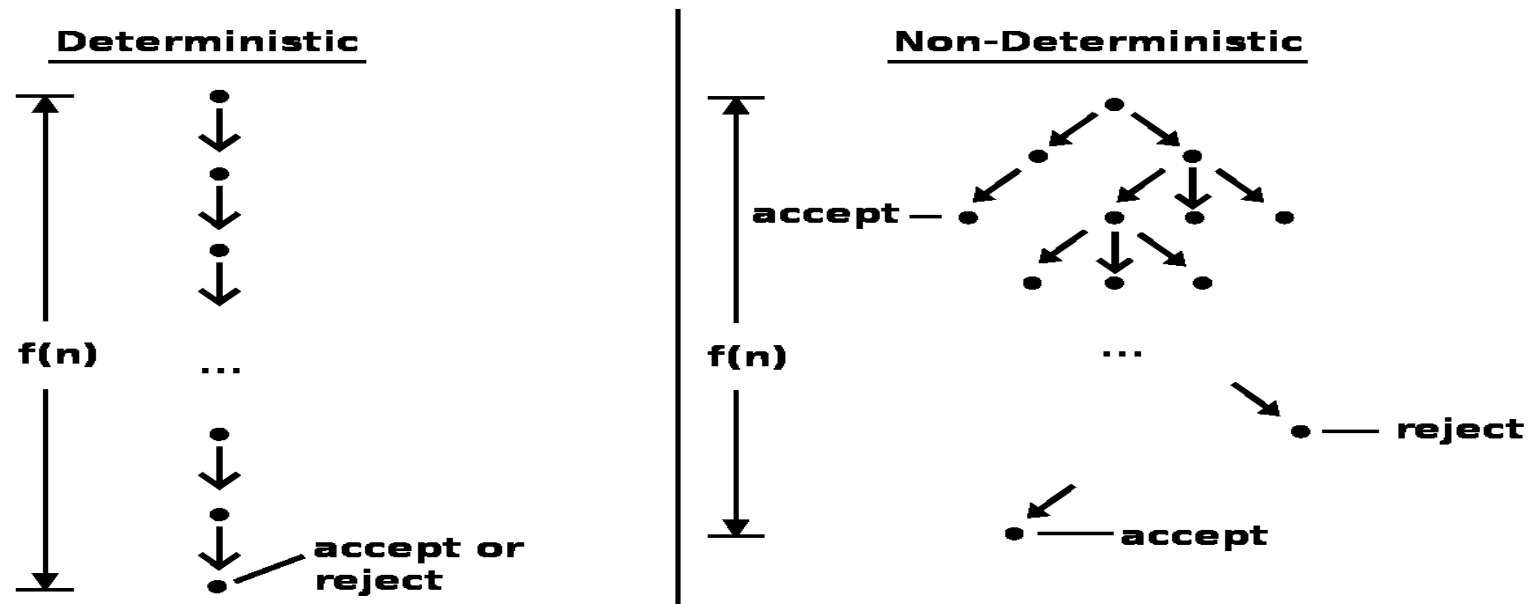
(b) Randomized algorithm structure

# Chapter : 11.1



# Deterministic vs non-deterministic algorithm

- A **deterministic algorithm** represents a single path from an input to an outcome.
- A **nondeterministic algorithm** represents a single path stemming into many paths, some of which may arrive at the same output and some of which may arrive at unique outputs.





# What is a non-deterministic algorithm?

- In computer programming, a nondeterministic algorithm is an algorithm that, even for the same input, can exhibit **different behaviors** on different runs, as opposed to a deterministic algorithm.
- A probabilistic algorithm's behaviors depends on a random number generator.
- An algorithm that solves a problem in nondeterministic polynomial time can run in polynomial time or exponential time depending on the choices it makes during execution.
- The nondeterministic algorithms are often used to find an **approximation** to a solution, when the exact solution would be too costly to obtain using a deterministic one.
- Non-deterministic algorithms are useful for finding **approximate solutions**, when an exact solution is difficult or expensive to derive using a deterministic algorithm.

# What is a non-deterministic algorithm?

- A non-deterministic algorithm is capable of execution on a deterministic computer which has an unlimited number of parallel processors.
- There are many problems which can be conceptualized with help of non-deterministic algorithms including the unresolved problem of P vs NP in computing theory.
- Non-deterministic algorithms are used in solving problems which allow multiple outcomes. Every outcome the non-deterministic algorithm produces is valid, regardless of the choices made by the algorithm during execution.

# What is a non-deterministic algorithm?

- There are many problems which can be conceptualized with help of non-deterministic algorithms including the unresolved problem of P vs NP in computing theory.
- Non-deterministic algorithms are used in solving problems which **allow multiple outcomes**. Every outcome the non-deterministic algorithm produces is valid, regardless of the choices made by the algorithm during execution.
- Non-deterministic algorithms are useful for finding **approximate solutions**, when an exact solution is difficult or expensive to derive using a deterministic algorithm.

# Non-deterministic algorithm.

- A non-deterministic algorithm usually has two phases and output steps. The first phase is the **guessing phase**, which makes use of arbitrary characters to run the problem.
- The second phase is the **verifying phase**, which returns true or false for the chosen string.
- If the **checking stage or verifying phase** of a nondeterministic algorithm is of polynomial time-complexity, then this algorithm is called an NP (nondeterministic polynomial) algorithm
- There are many problems which can be conceptualized with help of non-deterministic algorithms including the unresolved problem of P vs NP in computing theory.

# Non-deterministic operations and functions

- There are three new functions which specify such types of algorithms are:
  - ❑ **Choice(S)** arbitrarily chooses one of the elements of the set **S**.
  - ❑ **Failure()** signals an unsuccessful completion.
  - ❑ **Success()** signals a successful completion.
- The assignments statement **x: Choice (1, n)** could result in **x** being assigned any one of the integers in the range **[1, n]**. There is no rule specifying how this choice is to be made.
- The **Failure()** and the **Success()** signals are used to define a computation of the algorithm. These statements cannot be used to effect a return.

# Non-deterministic algorithm

- Whenever there is a set of the choices that lead to a successful completion, then one such set of the choices is always made and the algorithm terminates successfully.
- A **non - deterministic algorithm** terminates unsuccessfully if and only if there exists no set of the choices leading to a success signal.
- The computing times for the Choices, the Success, and the Failure are taken to be  $O(1)$ .
- A machine capable of executing a **non - deterministic algorithm** in this way is called a **non – deterministic** machine.
- A deterministic interpretation of a non-deterministic algorithm can be made by allowing **unbounded parallelism** in computation

# Nondeterministic Search

**Example 11.1** Consider the problem of searching for an element  $x$  in a given set of elements  $A[1 : n]$ ,  $n \geq 1$ . We are required to determine an index  $j$  such that  $A[j] = x$  or  $j = 0$  if  $x$  is not in  $A$ . A nondeterministic algorithm for this is Algorithm 11.1.

---


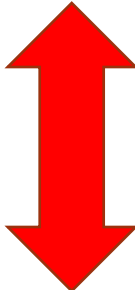
```
1   $j := \text{Choice}(1, n);$   
2  if  $A[j] = x$  then {write  $(j)$ ; Success();}  
3  write  $(0)$ ; Failure();
```

---

**Algorithm 11.1** Nondeterministic search

# Non-deterministic Sorting

```
1  Algorithm NSort( $A$ ,  $n$ )
2  // Sort  $n$  positive integers.
3  {
4      for  $i := 1$  to  $n$  do  $B[i] := 0$ ; // Initialize  $B[ ]$ .
5      for  $i := 1$  to  $n$  do
6      {
7           $j := \text{Choice}(1, n)$ ;
8          if  $B[j] \neq 0$  then Failure();
9           $B[j] := A[i]$ ;
10     }
11     for  $i := 1$  to  $n - 1$  do // Verify order.
12         if  $B[i] > B[i + 1]$  then Failure();
13     write ( $B[1 : n]$ );
14     Success();
15 }
```



---

**Algorithm 11.2** Nondeterministic sorting



# Nondeterministic 0/1 knapsack algorithm

---

```
1  Algorithm DKP( $p, w, n, m, r, x$ )
2  {
3       $W := 0; P := 0;$ 
4      for  $i := 1$  to  $n$  do
5          {
6               $x[i] := \text{Choice}(0, 1);$ 
7               $W := W + x[i] * w[i]; P := P + x[i] * p[i];$ 
8          }
9      if  $((W > m) \text{ or } (P < r))$  then Failure();
10     else Success();
11 }
```

---

**Algorithm 11.4** Nondeterministic knapsack algorithm

# Nondeterministic clique algorithm

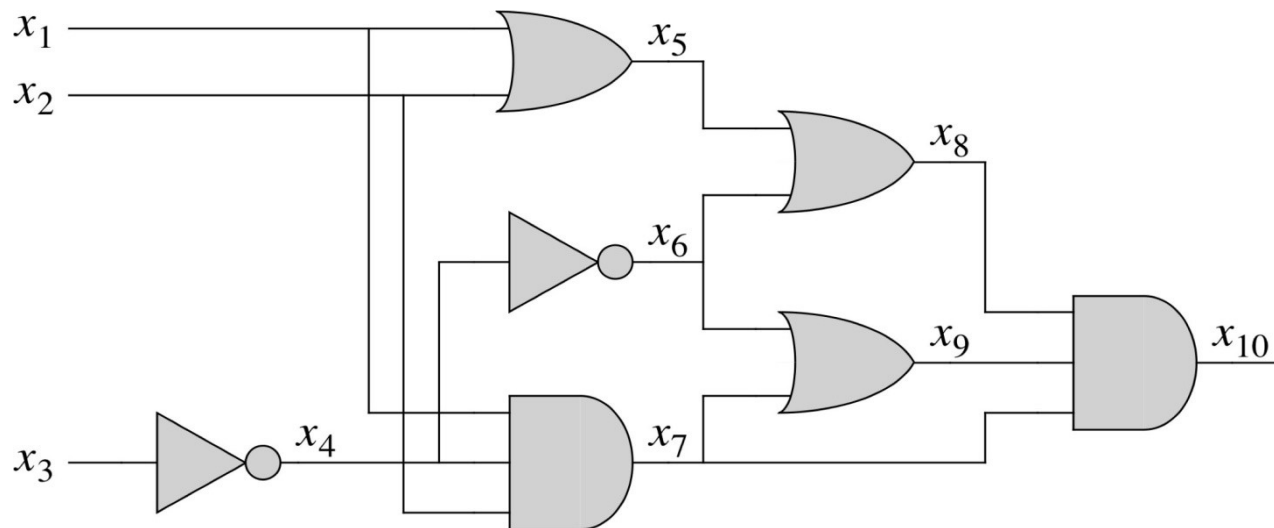
```
1  Algorithm DCK( $G, n, k$ )
2  {
3       $S := \emptyset$ ; //  $S$  is an initially empty set.
4      for  $i := 1$  to  $k$  do
5          {
6               $t := \text{Choice}(1, n)$ ;
7              if  $t \in S$  then Failure();
8               $S := S \cup \{t\}$  // Add  $t$  to set  $S$ .
9          }
10     // At this point  $S$  contains  $k$  distinct vertex indices.
11     for all pairs  $(i, j)$  such that  $i \in S, j \in S$ , and  $i \neq j$  do
12         if  $(i, j)$  is not an edge of  $G$  then Failure();
13     Success();
14 }
```

---

**Algorithm 11.5** Nondeterministic clique pseudocode

# Circuit-SAT (Satisfiability)

- Take a Boolean circuit with a single output node and ask whether there is an assignment of values to the circuit's inputs so that the output is “1”



- Seems simple enough, but no known deterministic polynomial time algorithm exists
- Easy to verify in **polynomial time**!
- This problem is in *NP*.

# Nondeterministic satisfiability

## Nondeterministic algorithm:

1. Guess truth assignment
2. Check assignment to see if it satisfies

```
1  Algorithm Eval( $E$ ,  $n$ )
2  // Determine whether the propositional formula  $E$  is
3  // satisfiable. The variables are  $x_1, x_2, \dots, x_n$ .
4  {
5      for  $i := 1$  to  $n$  do // Choose a truth value assignment.
6           $x_i := \text{Choice}(\text{false}, \text{true});$ 
7          if  $E(x_1, \dots, x_n)$  then Success();
8          else Failure();
9  }
```

---

**Algorithm 11.6** Nondeterministic satisfiability

# Randomization vs. Determinism

## ■ Deterministic algorithms

- **Non-adaptive**: always queries the same set of indices
- **Adaptive**: choice of  $i_t$  deterministically depends on answers to first  $t-1$  queries

## ■ Randomized algorithms

- **Non-adaptive**: indices are chosen randomly according to some distribution (e.g., uniform)
- **Adaptive**:  $i_t$  is chosen randomly according to a distribution, which depends on the answers to previous queries

## ■ Our focus: randomized algorithms

# Non-deterministic Sorting

```
1  Algorithm NSort( $A, n$ )
2  // Sort  $n$  positive integers.
3  {
4      for  $i := 1$  to  $n$  do  $B[i] := 0$ ; // Initialize  $B[ ]$ .
5      for  $i := 1$  to  $n$  do
6      {
7           $j := \text{Choice}(1, n)$ ;
8          if  $B[j] \neq 0$  then Failure();
9           $B[j] := A[i]$ ;
10     }
11     for  $i := 1$  to  $n - 1$  do // Verify order.
12         if  $B[i] > B[i + 1]$  then Failure();
13     write ( $B[1 : n]$ );
14     Success();
15 }
```

Uniform distribution

randomized algorithm

---

**Algorithm 11.2** Nondeterministic sorting

# Why use randomness?

- Randomness often helps in significantly reducing the work involved in determining a correct choice when there are several but finding one is very time consuming.
- Reduction of work (and time) can be significant on the average or in the worst case.
- Randomness often leads to very simple and elegant approaches to solve a problem or it can improve the performance of the same algorithm.
- **Avoid worst-case behavior:** randomness can (probabilistically) guarantee average case behavior
- Efficient approximate solutions to **intractable problems**
- “Randomized algorithm for a problem is usually **simpler** and **more efficient** than its deterministic counterpart.”

# Randomized algorithm

- A randomized algorithm is just one that depends on random numbers for its operation.
- A Randomized algorithm can be defined as one that receives, in addition to its input, a stream of random bits that it can use in the course of its action for the purpose of making random choices.
- A randomized algorithm may give different results when applied to the same input in different runs.
- Randomized algorithms are often easier to design than deterministic algorithms, though often the analysis requires some manipulations of random events or random variables.
- Often the **execution time** or **space requirement** of a randomized algorithm is smaller than that of the best deterministic algorithm that we know of for the same problem.

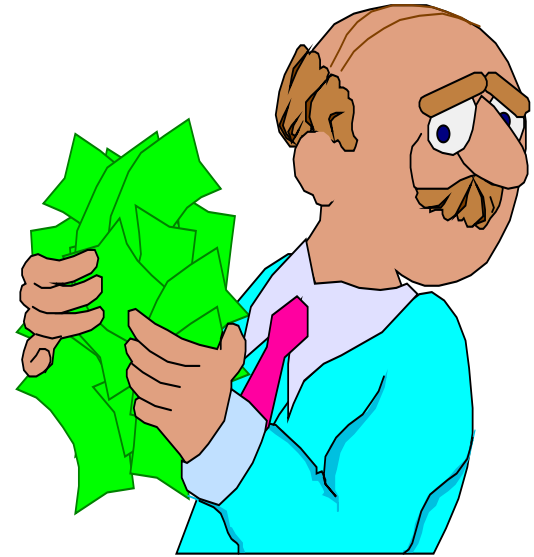


# Classification of Randomized Algorithm

- **Las Vegas algorithms:** It constitutes those randomized algorithms that always give a correct answer, or do not give an answer at all.
- **Monte Carlo Algorithms:** always gives an answer, but may occasionally produce an answer that is incorrect.
- The probability of producing an incorrect answer can be made arbitrarily small by running the algorithm repeatedly with independent random choices in each run.

# Classification of Randomized Algorithm

1. Numerical Probabilistic Algorithm
2. Monte-carlo Algorithm
3. Las-Vegas Algorithm
4. Sherwood Algorithm



# Numerical Probabilistic Algorithm

- Applied to the problems, when it is not possible to arrive at closed form solutions because of the uncertainties involved in the data or because of the limitations of a digital computer in representing irrationals.
- Approximation of such numerical values are to be computed (estimated) through simulation using randomness
- Quality of the solution produced by these algorithms can be improved by running the algorithms for a longer time.

❑ **Computing  $\Pi$**

❑ **Numerical Integration**

❑ **Average no of packets in router**

# Monte-Carlo Algorithms

- Always gives an answer but the answer is not necessarily correct
- Probability of correctness can be improved by running the algorithms for a longer time.
- A Monte-carlo Algorithm is said to have an **one-sided error** if the probability that its output is wrong for at least one of the possible outputs (yes/no) that it produce.
- A Monte-carlo Algorithm is said to have an **two-sided error** if there is non-zero probability that it errors when it outputs yes or no.

❑ **Randomized Min-Cut**

❑ **Primality Testing**

❑ **Finding Majority Element**

# Las-Vegas Algorithms

- If any answer is to be found out using Las-Vegas Algorithm, then the **answer is correct**.
- These are randomized algorithms which never produce incorrect results, but whose execution time may vary from one run to another.
- Random choices made within the algorithm are used to establish an expected running time for the algorithm that is, essentially, independent of the input.
- The probability of finding answer increases if the algorithm is repeated good number of times.

❑ **Identifying the Repeated Element**

❑ **8 Queen Problem**

# Sherwood Algorithms

- This algorithm always gives the answer and answer is always correct
- This algorithms are used, when a deterministic algorithm behave inconsistently
- Randomness tends to make best case's behaviour look like that of the average case behaviour

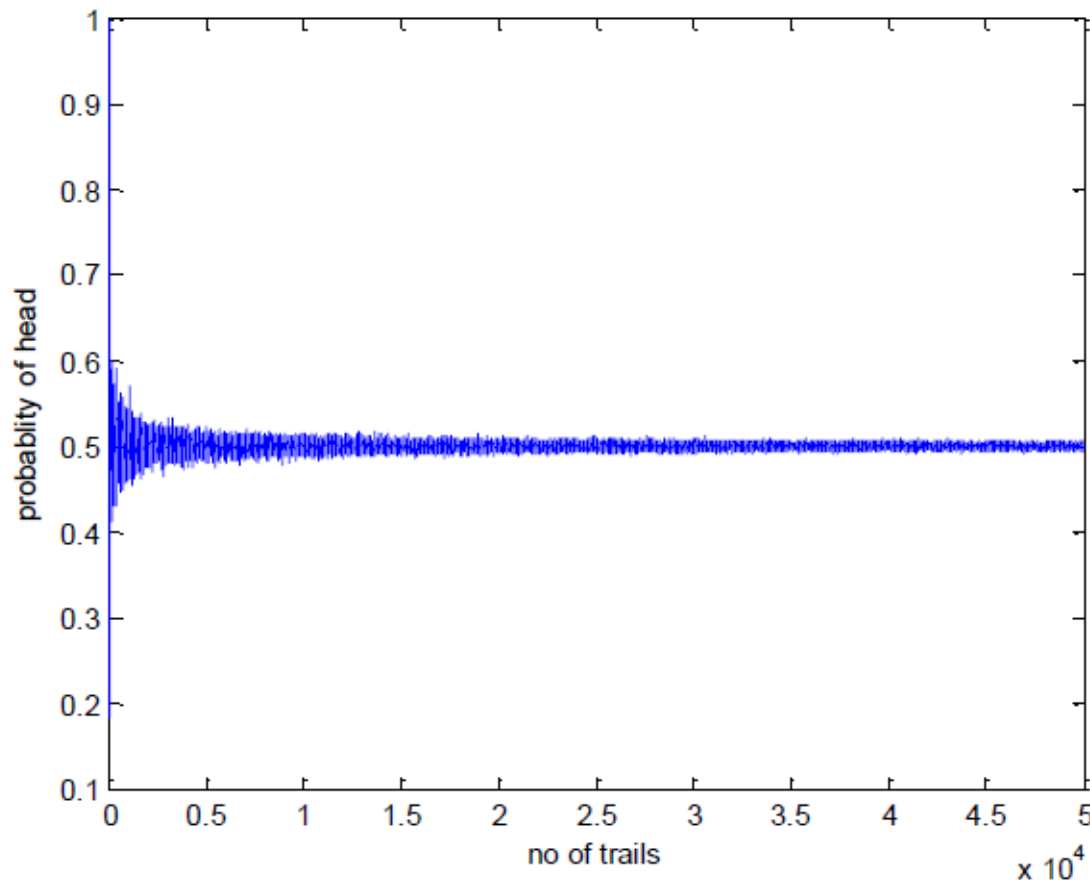
## ❑ Randomized Quick Sort

# Numerical Probabilistic Algorithm

Here the random element allows us to get approximate numerical results, often much faster than direct methods, and multiple runs will provide increasing approximation.

# Coin Tossing

Through the simulation, show that probability of getting HEAD by tossing a fair coin is about 0.5.





# Compute 'pi' using randomized algorithm

- Main program

```
close all;
clear all;

min_size = 10;
max_size = 1000;
step = 1;

numSimulations = 0;
for i = min_size:step:max_size
    numSimulations = numSimulations + 1;
end

arr_iter = zeros(1, numSimulations);
arr_pival = zeros(1, numSimulations);

k = 1;
for i = min_size:step:max_size
    arr_iter(k) = i;
    arr_pival(k) = findPi(i);
    k = k + 1;
end

arr_ref = 3.141 * ones(1, numSimulations);

figure;
plot(arr_iter, arr_pival, arr_iter, arr_ref);
xlabel('No. of Iterations');
ylabel('Value of Pi');
title('Computing Value of Pi');
```

# Compute 'pi' using randomized algorithm

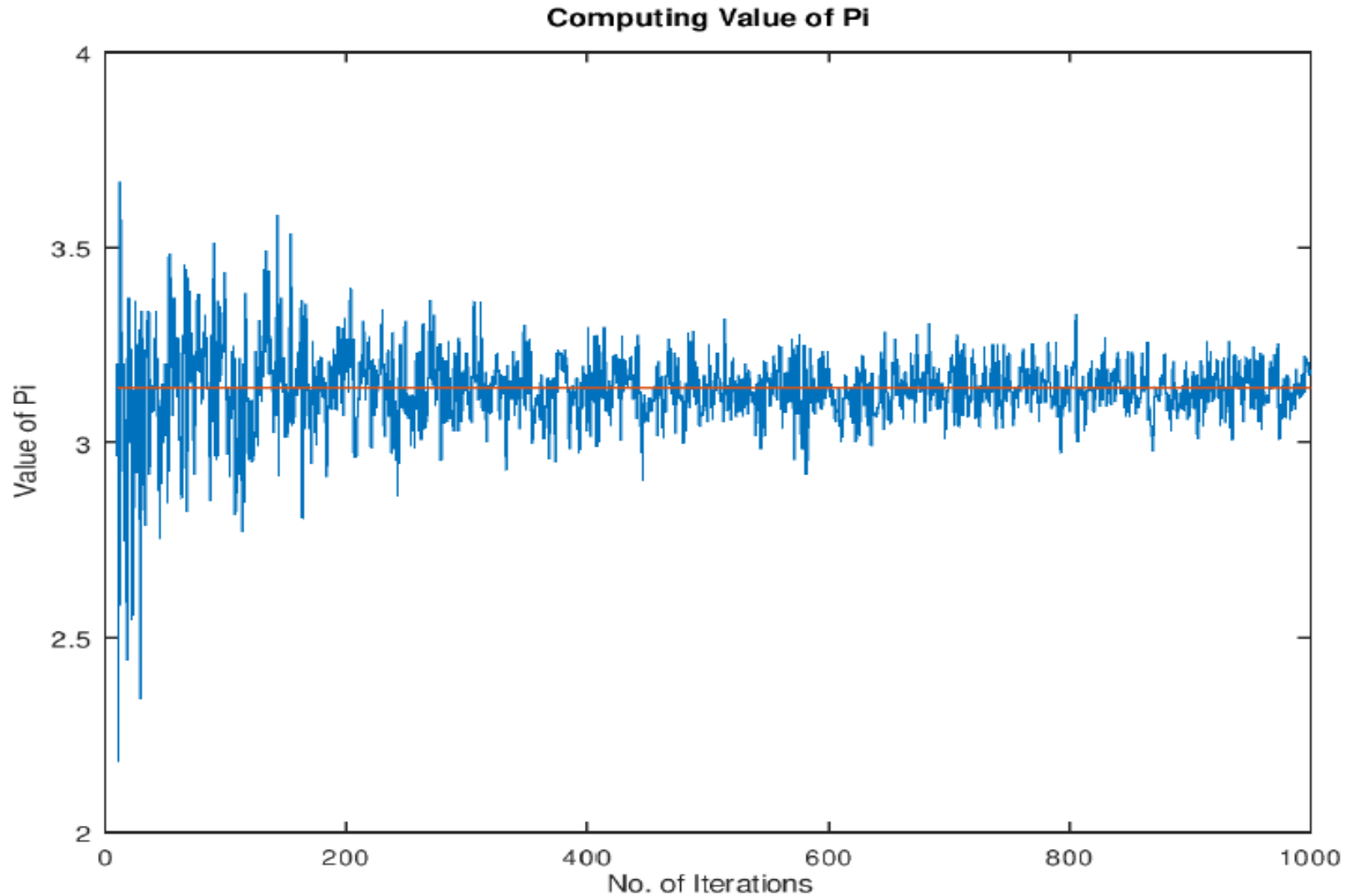
Function **findPi**

```
function [ pi_val ] = findPi( num_points )

    incircle = 0;
    for i = 1:num_points
        x = rand(1,1);
        y = rand(1,1);
        if ((x - 0.5)^2 + (y - 0.5)^2) < 0.25
            incircle = incircle + 1;
        end
    end

    pi_val = 4.0 * incircle / num_points;
end
```

# Compute 'pi' using randomized algorithm



## Identifying the repeated array number

- Let an array  $a[ ]$  of  $n$  numbers that has  $n/2$  distinct elements and  $n/2$  copies of another element. The problem is to identify the repeated element.

## Algorithm: Identifying the repeated array number

```
1  RepeatedElement( $a, n$ )
2  // Finds the repeated element from  $a[1 : n]$ .
3  {
4      while (true) do
5      {
6           $i := \text{Random}() \bmod n + 1$ ;  $j := \text{Random}() \bmod n + 1$ ;
7          //  $i$  and  $j$  are random numbers in the range  $[1, n]$ .
8          if  $((i \neq j) \text{ and } (a[i] = a[j]))$  then return  $i$ ;
9      }
10 }
```

Elegant randomized Las Vegas algorithm that takes only  $O(\log n)$  time

# Primality testing

A primality test is an algorithm for determining whether an input number is prime.

# Primality testing

- A fundamental problem in mathematics is: given a number  $N$ , how do we know whether it is prime or composite.
- A natural number  $N$  is said to be a prime number if it can be divided only by 1 and itself. Primality Testing is done to check if a number is a prime or not.

## Algorithm : Primality testing : first attempt

```
1  Prime0( $n, \alpha$ )
2  // Returns true if  $n$  is a prime and false otherwise.
3  //  $\alpha$  is the probability parameter.
4  {
5       $q := n - 1$ ;
6      for  $i := 1$  to large do // Specify large.
7      {
8           $m := q$ ;  $y := 1$ ;
9           $a := \text{Random}() \bmod q + 1$ ;
10         // Choose a random number in the range  $[1, n - 1]$ .
11          $z := a$ ;
12         // Compute  $a^{n-1} \bmod n$ .
13         while ( $m > 0$ ) do
14         {
15             while ( $m \bmod 2 = 0$ ) do
16             {
17                  $z := z^2 \bmod n$ ;  $m := \lfloor m/2 \rfloor$ ;
18             }
19              $m := m - 1$ ;  $y := (y * z) \bmod n$ ;
20         }
21         if ( $y \neq 1$ ) then return false;
22         // If  $a^{n-1} \bmod n$  is not 1,  $n$  is not a prime.
23     }
24     return true;
25 }
```





## Another primality testing algorithm

```
1  Prime1( $n$ )
2  {
3      // Specify  $t$ .
4      for  $i := 1$  to  $t$  do
5          {
6               $m := \text{Power}(n, 0.5)$ ;
7               $j := \text{Random}() \bmod m + 2$ ;
8              if  $((n \bmod j) = 0)$  then return false;
9              // If  $j$  divides  $n$ ,  $n$  is not prime.
10         }
11     return true;
12 }
```

# Randomized search

```
1  Algorithm RSearch( $a, x, n$ )
2  // Searches for  $x$  in  $a[1 : n]$ . Assume that  $x$  is in  $a[ ]$ .
3  {
4      while (true) do
5      {
6           $i := \text{Random}() \bmod n + 1$ ;
7          //  $i$  is random in the range  $[1, n]$ .
8          if ( $a[i] = x$ ) then return  $i$ ;
9      }
10 }
```

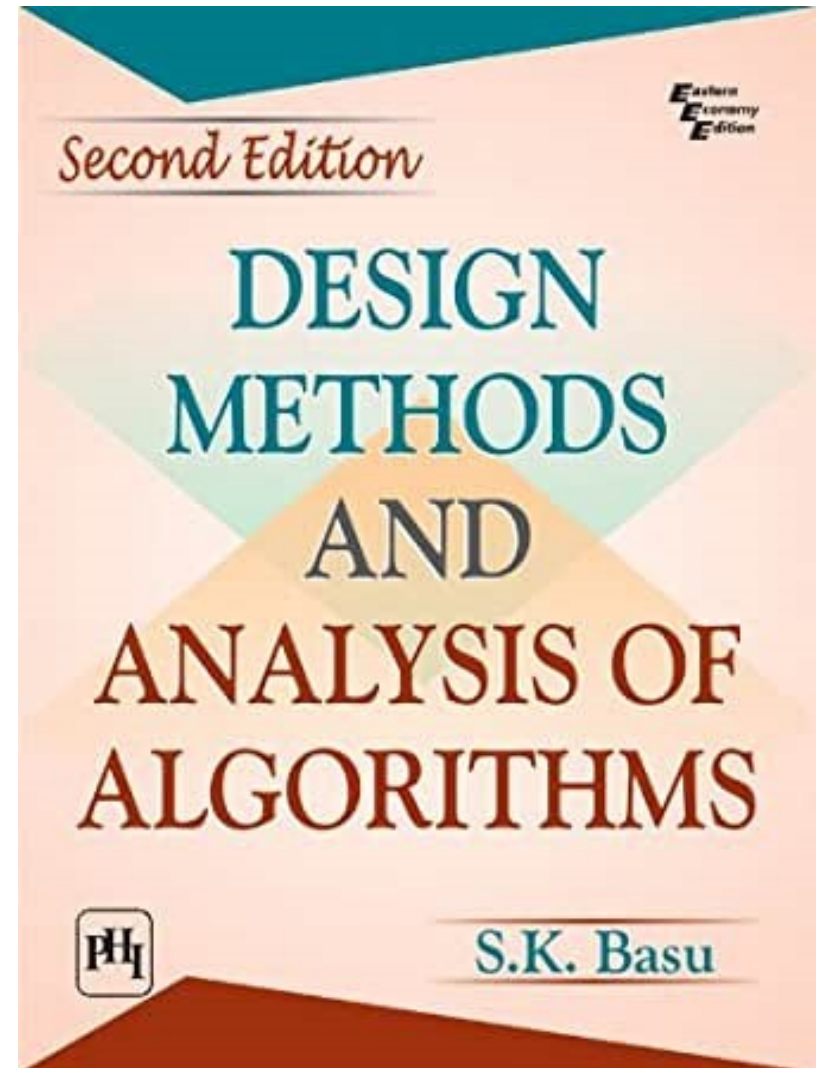
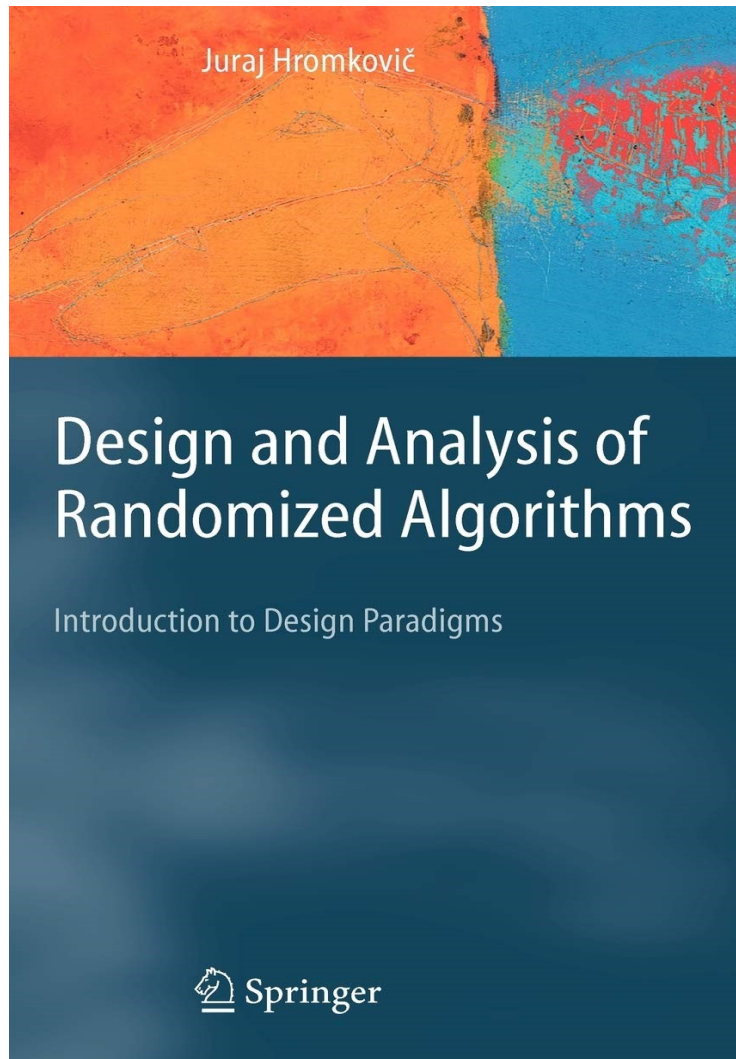
## Sample problems

1. Write the most efficient algorithm you can think of for the following: Find the  $k$ -th item in an  $n$ -node doubly-linked list. What is the running time in terms of big-theta?
2. Write the most efficient algorithm you can think of for the following: Given a set of  $p$  points, find the pair closest to each other. Be sure and try a divide-and-conquer approach, as well as others. What is the running time in terms of big-theta?
3. Design and implement an algorithm that finds all the duplicates in a random sequence of integers.
4. Reconsider the three algorithms you just designed given the following change: the input has increased by 1,000,000,000 times.

## Sample problems

5. You have two arrays of integers. Within each array, there are no duplicate values but there may be values in common with the other array. Assume the arrays represent two different sets. Define an  $O(n \log n)$  algorithm that outputs a third array representing the union of the two sets. The value " $n$ " in  $O(n \log n)$  is the sum of the sizes of the two input arrays.
6. You are given an increasing sequence of numbers  $u_1, u_2, \dots, u_m$ , and a decreasing series of numbers  $d_1, d_2, \dots, d_n$ . You are given one more number  $C$  and asked to determine if  $C$  can be written as the sum of one  $u_i$  and one  $d_j$ . There is an obvious brute force method, just comparing all the sums to  $C$ , but there is a much more efficient solution. Define an algorithm that works in linear time.
7. Design and implement a dynamic programming algorithm to solve the **change counting problem**. Your algorithm should always find the optimal solution--even when the greedy algorithm fails.

# Suggested Reading



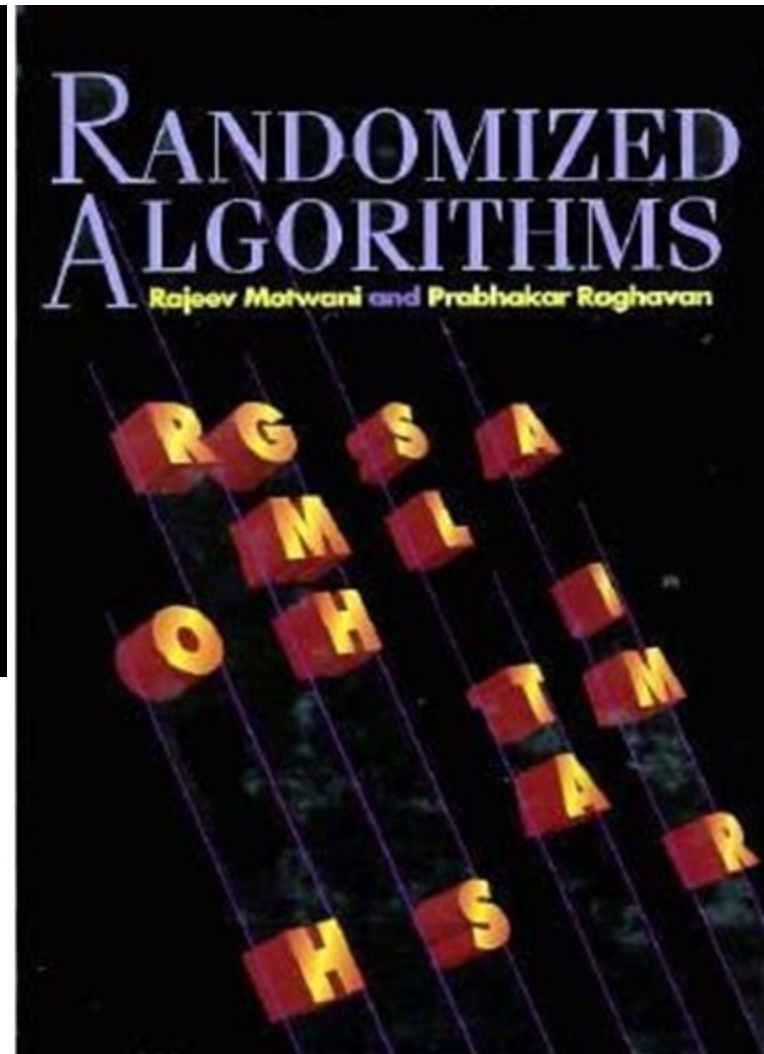
# Suggested Reading



A mathematically deep computer scientist, a catalyst of Silicon valley innovation and a luminary in many academic disciplines.



**Late Prof. Rajeev Motwani**  
(BT/CSE/1983)  
(1962-2009)





# Rajeev Motwani



“Today, whenever you use a piece of technology, there is a good chance a little bit of Rajeev Motwani is behind it,” was the tribute paid by **Sergey Brin** after Rajeev Motwani’s death in 2009.

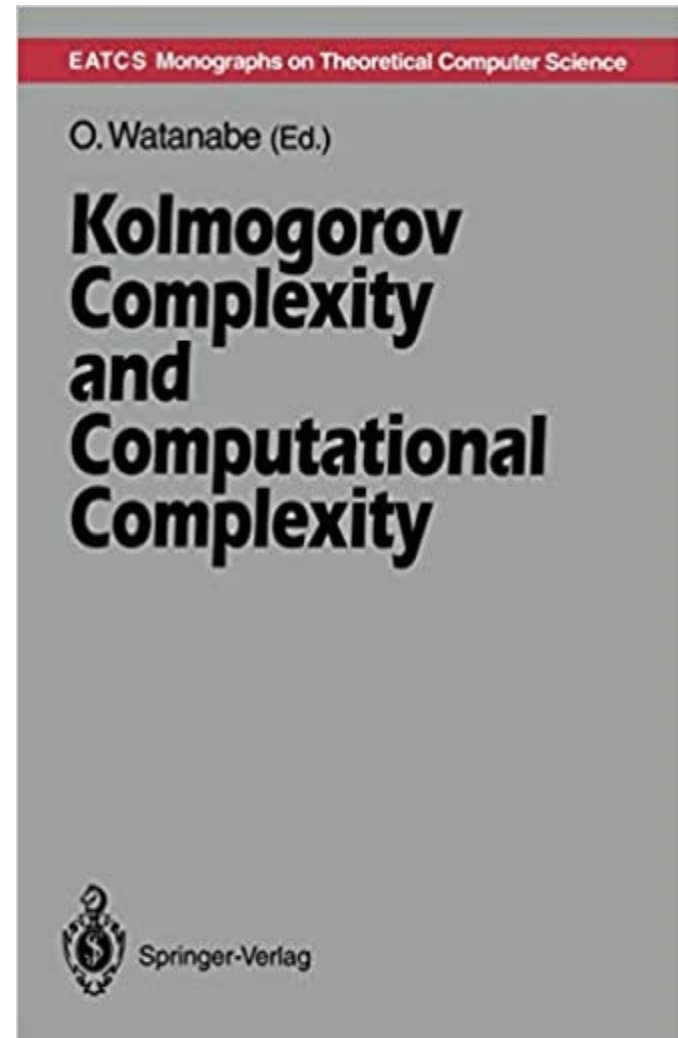
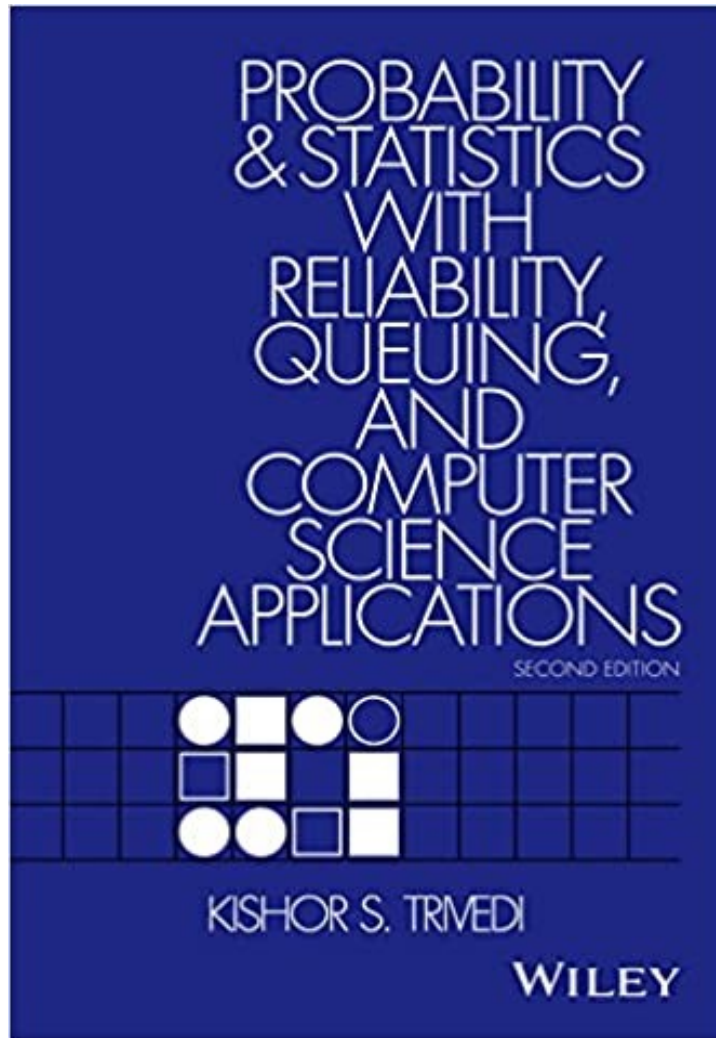
**Professor Motwani** closely collaborated with **Brin** and **Page** while writing an influential early paper on Page Rank algorithm, which formed the basis for Google’s search techniques. After Google was founded, he became a member of Google’s technical advisory council. Professor Motwani’s research spanned various fields such as databases, data mining, data privacy, web search and robotics.



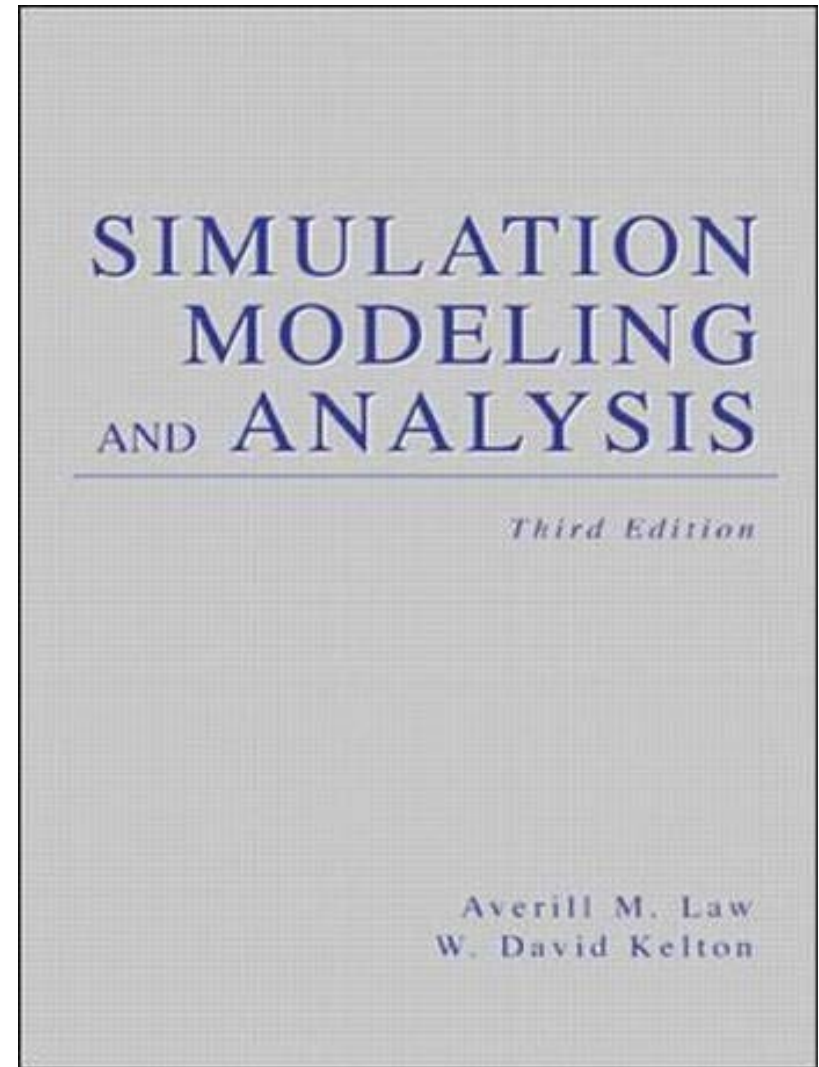
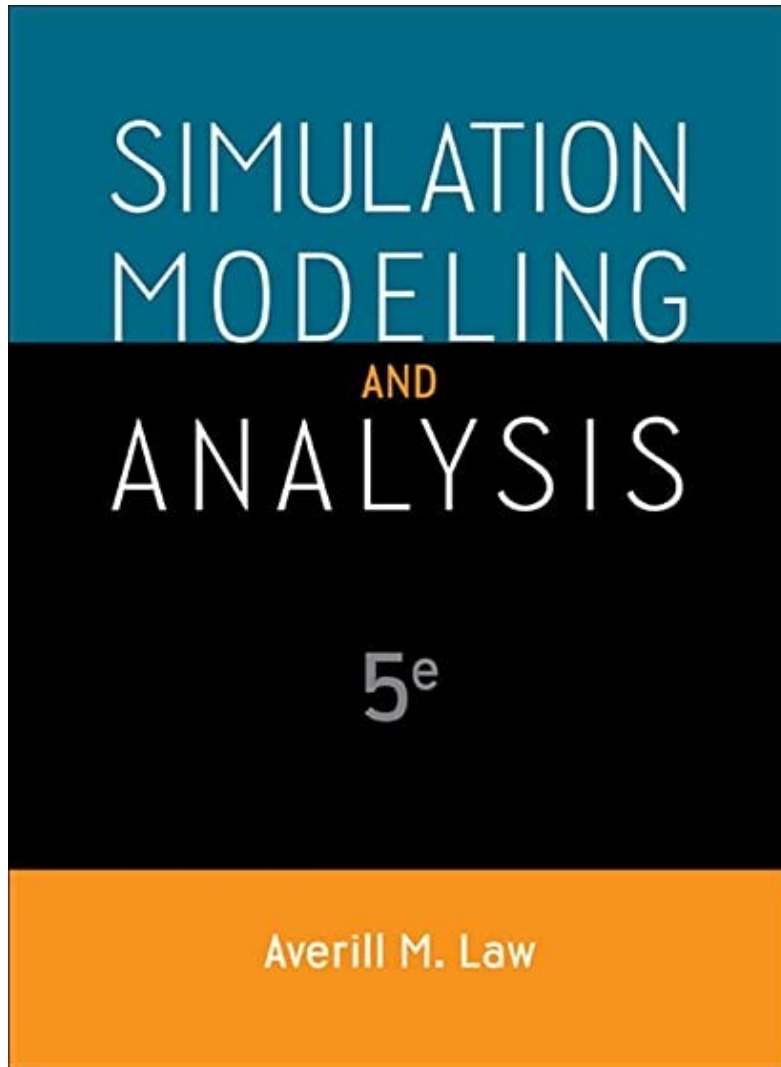
# Kolmogorov Complexity

- The goal of Kolmogorov (or Kolmogorov-Chaitin or descriptive) complexity is to develop a measure of the complexity or “randomness” of an object.
- Kolmogorov complexity of an object or algorithm is the length of its optimal specification. In some sense, it could be thought of as algorithmic **entropy**, in the sense that it is the amount of information contained in the object.

# Suggested Reading



## Suggested Reading: Simulation & Modeling



Thanks for Your Attention!



# Primes is in P



Nitin Saxena(left), Neeraj Kayal and Manindra Agrawal(right) , have discovered a "**polynomial time**" **algorithm for primality testing**.

Agrawal and company also give a certificate or proof that the algorithm works in polynomial time and always returns correct answers. The work has been hailed as the most outstanding work in theoretical computer science to emerge from India.

# Primes is in P

- <https://annals.math.princeton.edu/2004/160-2/p12>
- [https://www.cse.iitk.ac.in/users/manindra/algebra/primality\\_v6.pdf](https://www.cse.iitk.ac.in/users/manindra/algebra/primality_v6.pdf)
- <https://www.microsoft.com/en-us/research/people/neeraka/>



# Primes is in P

- Algorithmic complexity theory classifies problems depending on how difficult they are to solve.
- A problem is assigned to P class (where P stands for 'polynomial time') if the number of steps needed to solve it is at most some power of the problem size. That is, the problem is easy, feasible and tractable.
- A problem is assigned to NP class (where NP stands for 'non-deterministic polynomial time') if it permits a non-deterministic solution but the proof of a given solution is bounded by some power of the problem's size. That is, verification of the solution is easy, feasible and tractable.
- Primality testing, as mentioned before, is much easier than the problem of factorisation and has long been believed to be in P. Only now with AKS' work, it has actually been shown to belong to P. Indeed, the **AKS paper** is titled "Primes is in P".
- The big (philosophical) question in theoretical computer science is whether all problems are easy, feasible and tractable; that is whether  $P=NP$ ? Researchers, however, would like to believe that there will always be some unsolvable problems and the conjecture P is not equal to NP. This has been posed as the millennium problem with a big award attached to it.

# PRIMES is in P

- The **AKS primality test** (also known as **Agrawal–Kayal–Saxena primality test** and **cyclotomic AKS test**) is a deterministic primality-proving algorithm created and published by Manindra Agrawal, Neeraj Kayal, and Nitin Saxena, computer scientists at the Indian Institute of Technology Kanpur, on August 6, 2002, in an article titled "PRIMES is in P".
- The algorithm was the first that can provably determine whether any given number is prime or composite in **polynomial time**, without relying on mathematical conjectures such as the generalized Riemann hypothesis.