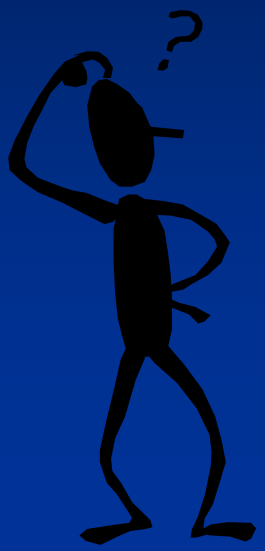
Randomized Algorithms

Bibhudatta Sahoo, PhD

National Institute of Technology Rourkela

Why use randomness?

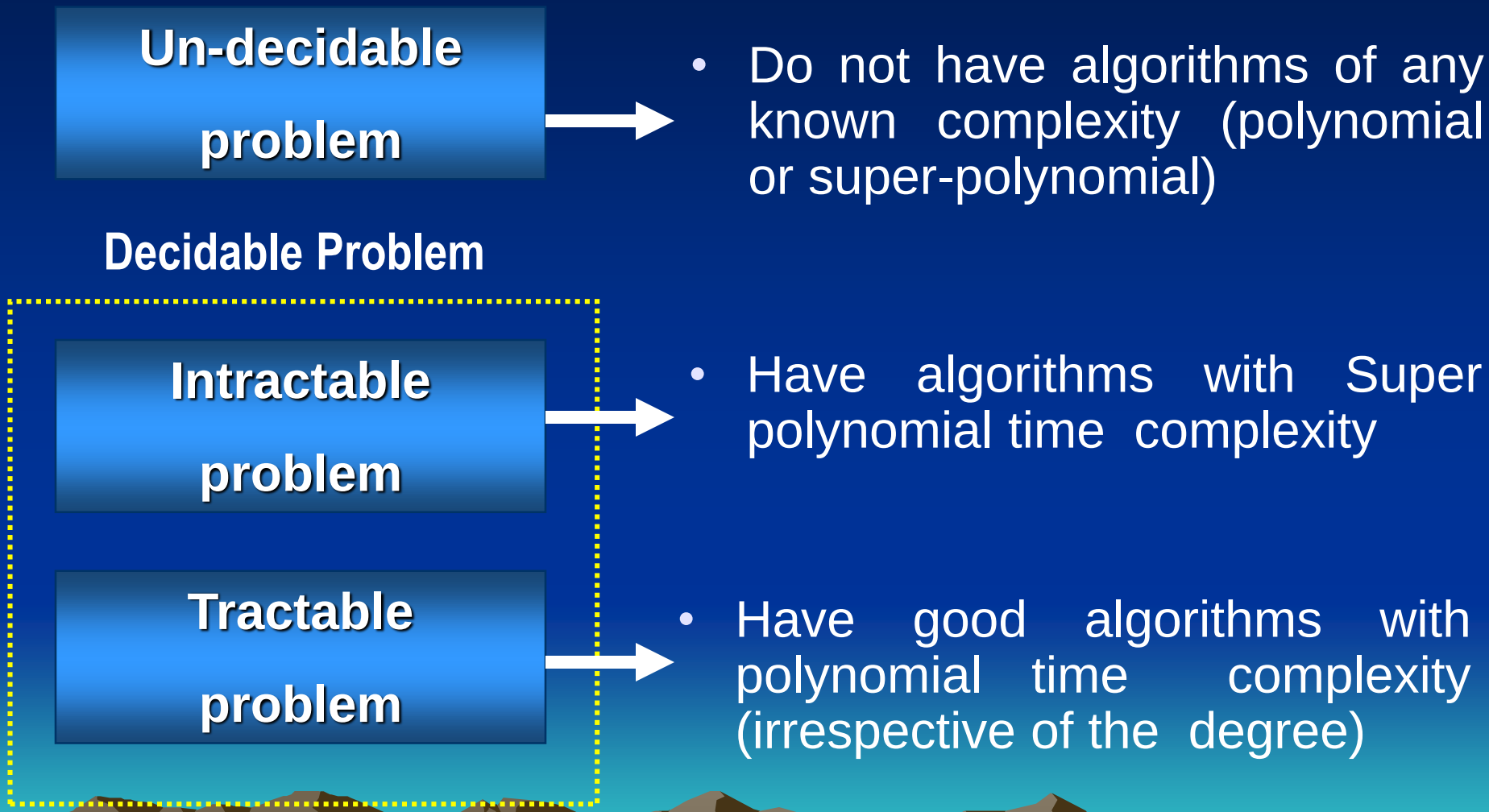
- Avoid worst-case behavior: randomness can (probabilistically) guarantee average case behavior
- Efficient approximate solutions to **intractable problems**
- "Randomized algorithm for a problem is usually **simpler** and **more efficient** than its deterministic counterpart."



Intractable Problems

?

Problem Taxonomy



- Do not have algorithms of any known complexity (polynomial or super-polynomial)
- Have algorithms with Super polynomial time complexity
- Have good algorithms with polynomial time complexity (irrespective of the degree)

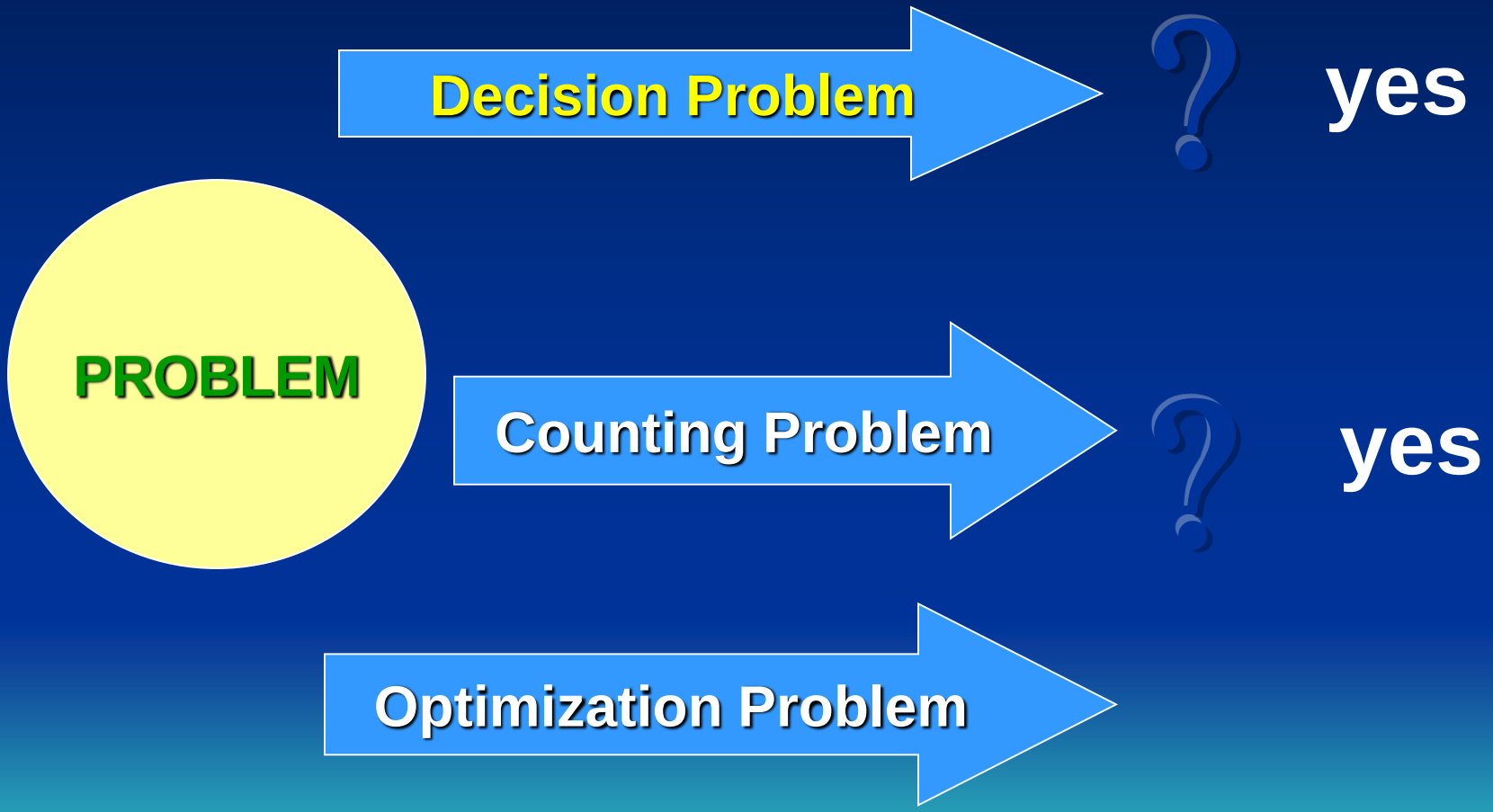
Different type of decidable Problem

- **Decision Problems:** The class of problems, the output is either yes or no
- ➲ Whether a given number is prime?
- **Counting Problem:** The class of problem, the output is a natural number
- ➲ How many distinct factor are there for a given number
- **Optimization Problem:** The class of problem with some objective function based on the problem instance
- ➲ Finding a minimal spanning tree for a weighted graph

Notes

- **A decision problem** is in the class NP if: there is no known algorithm for it that will execute in polynomial time on a conventional computer
- it can be solved in polynomial time using a non-deterministic computer (one with inherently unlimited parallelism).
- A **problem is intractable** if no polynomial time algorithm can possibly be devised for it.
- A problem in NP is **NP-complete** if every other problem in NP can be expressed in terms of it by means of a polynomial time algorithm.
- **NP-complete problems** are considered to be the **hardest problems**, since if any problem in NP is shown to be intractable then all NP-complete problems are intractable.
- However, if any NP-complete problem can be solved in polynomial time, all problems in NP become tractable.

Optimization Problem



Problem Taxonomy: examples

Un-decidable problem

- Problems about abstract machines)

1. The halting problem (determining whether a Turing machine halts).
2. The mortality problem

Decidable Problem

Intractable problem

- Brut force algorithm to solve TSP requires $O(2^n n!)$

Tractable problem

- Brut force algorithm (Insertion sort) requires $O(n^2)$ for internal sorting problem

Introducing two important classes of problems: P and NP

- P is the class of problems solvable in polynomial time by deterministic Turing Machines (TM).

Or

- Mmore formally, P is the class of problems that, given a language L, there is a polynomial T(n) such that L = L(M) for a deterministic TM M of T(n).
- T(n) is the time complexity. T(n) is defined as the running time of a TM M whenever M is given an input of w (with length n) and halts after making at most T(n) moves (even if M does not accept).
- An example of an algorithm that belongs to P is Kruskal's greedy algorithm to find a minimum-weight spanning tree of a graph.

Introducing two important classes of problems: P and NP

- NP is the class of problems solvable in polynomial time by nondeterministic TMs.

Or

- More formally, NP is the class of problems, given a language L, there is a nondeterministic TM M and a polynomial time complexity T(n) such that L = L(M) and when M is given an input of length n, there are no sequences of more than T(n) moves of M.

- The definitions of P and NP lead to one of the most studied and open-ended questions in mathematics (and computer science): Is P = NP?

NP-Complete Problems

- In computational complexity theory, the complexity class NP-complete is a subset of NP, the set of all decision problems whose solutions can be verified quickly. (More strictly speaking, it is the set of decision problems that can be solved in polynomial time on a nondeterministic Turing machine.)

- At present, all known algorithms for NP-complete problems require time that is superpolynomial in the input size, and it is unknown whether there are any faster algorithms.

NP-Complete problem

- NP-Complete is a problem that is NP, but not P. More formally, NP-Complete is the class of problems, given a language L, L is in NP and for every language L' in NP there is a polynomial-time reduction of L' to L.

Some well-known problems that are NP-complete when expressed as decision problems.

- Boolean satisfiability problem (SAT)
- N-puzzle
- Knapsack problem
- Hamiltonian path problem
- Travelling salesman problem
- Subgraph isomorphism problem
- Subset sum problem
- Clique problem
- Vertex cover problem
- Independent set problem
- Graph coloring problem

NP-Complete Problems & randomized algorithm

Hard problems (NP-complete)	Easy problems (in P)
3SAT	2SAT, HORN SAT
TRAVELING SALESMAN PROBLEM	MINIMUM SPANNING TREE
LONGEST PATH	SHORTEST PATH
3D MATCHING	BIPARTITE MATCHING
KNAPSACK	UNARY KNAPSACK
INDEPENDENT SET	INDEPENDENT SET on trees
INTEGER LINEAR PROGRAMMING	LINEAR PROGRAMMING
RUDRATA PATH	EULER PATH
BALANCED CUT	MINIMUM CUT

Karp's 21 NP-complete problems

- Richard M. Karp (1972). "Reducibility Among Combinatorial Problems" . In R. E. Miller and J. W. Thatcher (editors). *Complexity of Computer Computations*. New York: Plenum. pp. 85–103.

1. **Satisfiability**: the boolean satisfiability problem for formulas in conjunctive normal form(often referred to as SAT)
2. **0–1 integer programming** (A variation in which only the restrictions must be satisfied, with no optimization)
3. **Clique** (see also independent set problem)
4. **Set packing**
5. **Vertex cover**
6. **Set covering**
7. **Feedback node set**

Karp's 21 NP-complete problems

8. **Feedback arc set**
9. **Directed Hamilton circuit** (Karp's name, now usually called **Directed Hamiltonian cycle**)
10. **Undirected Hamilton circuit** (Karp's name, now usually called**Undirected Hamiltonian cycle**)
11. **Satisfiability with at most 3 literals per clause** (equivalent to 3-SAT)
12. **Chromatic number** (also called the Graph Coloring Problem)
13. **Clique cover**
14. **Exact cover**

Karp's 21 NP-complete problems

15. **Hitting set**
16. **Steiner tree**
17. **3-dimensional matching**
18. **Knapsack** (Karp's definition of Knapsack is closer to Subset sum)
19. **Job sequencing**
20. **Partition**
21. **Max cut**

Solving NP-complete problems in general, and they often give rise to substantially faster algorithms:

- Approximation: Instead of searching for an optimal solution, search for an "almost" optimal one.
- Randomization: Use randomness to get a faster average running time, and allow the algorithm to fail with some small probability.
- Restriction: By restricting the structure of the input (e.g., to planar graphs), faster algorithms are usually possible.
- Parameterization: Often there are fast algorithms if certain parameters of the input are fixed.
- Heuristic: An algorithm that works "reasonably well" on many cases, but for which there is no proof that it is both always fast and always produces a good result. Metaheuristic approaches are often used.

Solving NP-complete problems

- **One example** of a heuristic algorithm is a suboptimal O(*n* log *n*) greedy algorithm used for graph coloring during the register allocation phase of some compilers, a technique called graph-coloring global register allocation.
- Each vertex is a variable, edges are drawn between variables which are being used at the same time, and colors indicate the register assigned to each variable. Because most RISC machines have a fairly large number of general-purpose registers, even a heuristic approach is effective for this application.

Reference Books

- S. K. Basu, “ Design Methods and Analysis of Algorithms”, Prentice Hall Of India, 2007.
- Juraj Hromkovic, “ Design and Analysis of Randomized Algorithms”, Springer , 2005.
- R. Motwani and P. Raghavan, “ Randomized Algorithms”, Cambridge University Press, 1995.
- J. E. Hopcroft, (R. Motwani) and J. D. Ullman, Introduction to Automata Theory, Languages, and Computation. Addison Wesley, 2000.

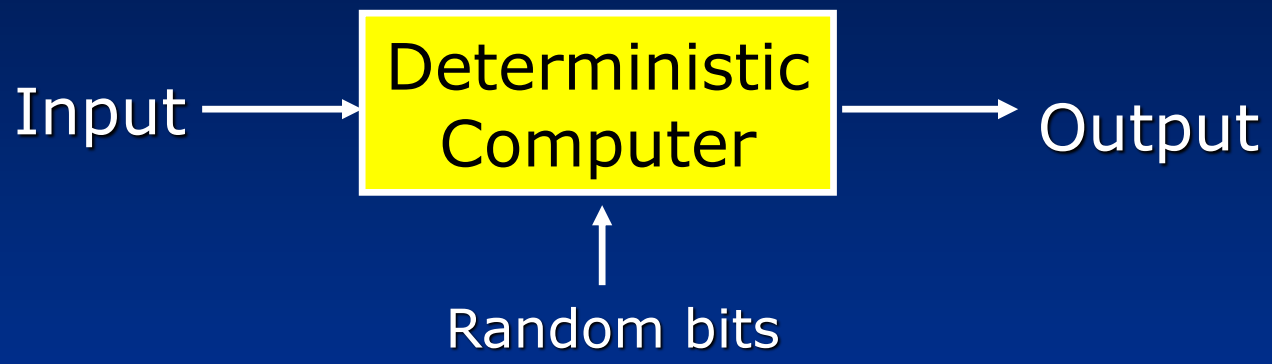
Advantages of Randomized Algorithm

- There are two main advantages of randomized algorithms:

- First, often the execution time or space requirement is smaller than that of the best deterministic algorithm.

- Second, randomized algorithms are extremely simple to comprehend and implement.

Randomized Algorithm



* A randomized algorithm is just one that depends on random numbers for its operation
* Randomized algorithm also known as Monte Carlo algorithms or stochastic methods

Classes of randomized algorithms ?

- Las Vegas algorithms

  always correct; expected running time (“probably fast”)

  Examples: randomized Quick sort,
  randomized algorithm for closest pair

- Monte Carlo algorithms (mostly correct):

  probably correct; guaranteed running time

  Example: randomized primality test

Randomized Algorithm

- 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 Algorithm

- Algorithm where some of the actions are dependent on chance are generally termed as probabilistic algorithms or randomized algorithms.
- The results of randomized algorithms may not be always 100% correct.
- Correctness of the output of these algorithm is characterized by some probability.
- The correctness can be increased by running the algorithms for a longer time.
- The output of randomize algorithm differs from run to run on the same input.

Advantages of Randomized Algorithm

- Randomized algorithms runs faster than the known best deterministic algorithm.
- The randomized algorithms are simple to describe and implement than the deterministic algorithms for comparable performance
- The randomized algorithms also yields better complexity bounds

Why Randomization ?

- 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.

Risk

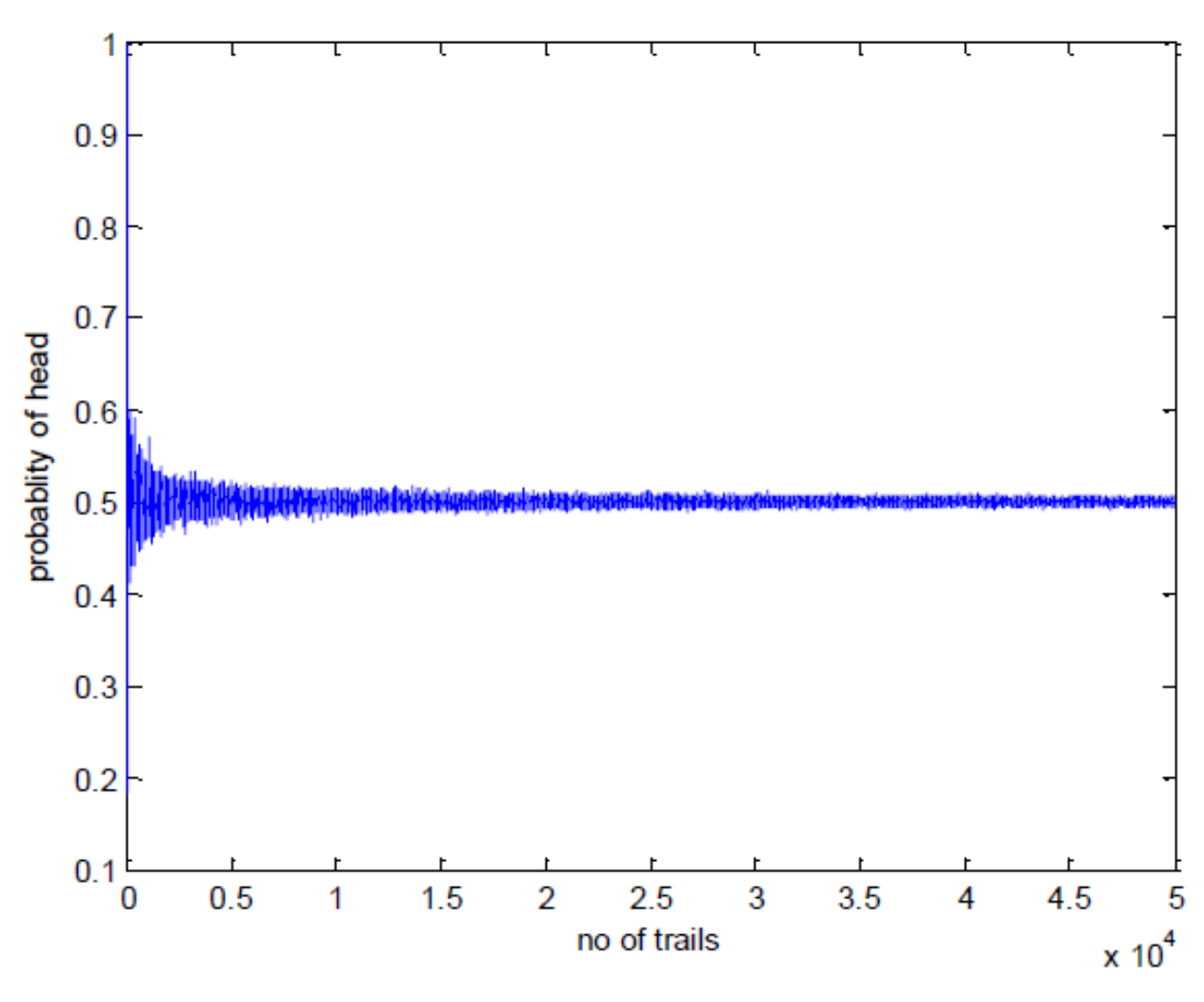
- Loss of confidence in the correctness. This loss can be made very small by repeated employment of randomness.

- Assumes the availability of truly unbiased random bits which are very expensive to generate in practice.

Coin Tossing

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

The probability of tossing the coin converges to 0.5



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.

Example: 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.

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.

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

Random search Algorithm

```
Algorithm Rseach (a, x, n)
// searches for x in a linear arry a[1:n].
// assume that x is in a[ ].
{
    while (true) do
          {
          i := Random ( ) mod n+1;
// i is random in the range[1:n]
          if ( a[i] = x) then return i;
          }
}
```

Randomized Quicksort

Given a set S of n numbers.

1. If $|S| \leq 1$, output the elements of S and stop.
2. Choose a pivot element y uniformly at random from S.
3. Determine the set S_1 of elements $\leq y$, and the set S_2 of elements $> y$.
4. Recursively apply to S_1, output the pivot element y, then recursively apply to S_2.

Randomized Quicksort

Algorithm RANDOMIZEDQUICKSORT
Input: An array $A[1..n]$ of n elements.
Output: The elements in A sorted in nondecreasing order.

1. *rquicksort*$(1, n)$

Procedure *rquicksort*$(low, high)$

1. **if** $low < high$ **then**
2. $v \leftarrow$ *random*$(low, high)$
3. interchange $A[low]$ and $A[v]$
4. SPLIT$(A[low..high], w)$ {w is the new position of the pivot}
5. *rquicksort*$(low, w-1)$
6. *rquicksort*$(w+1, high)$
7. **end if**

Informal analysis

Randomized quicksort has expected time (averaged over all choices of pivots) of $O(n\log n)$.

Assume we sort the set and divide into four parts, the middle two contain the best pivots.

Each is larger than at least $\frac{1}{4}$ of the elements and smaller than at least $\frac{1}{4}$ of the elements.

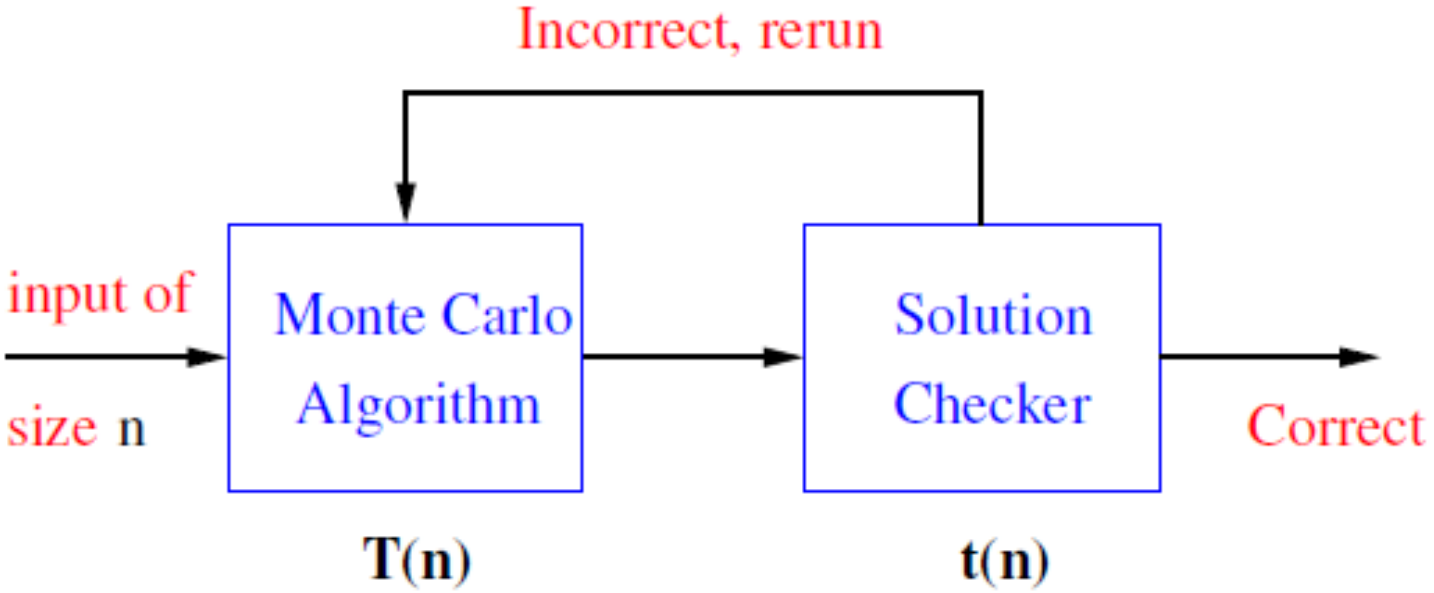
Choosing an element from the middle two means we split at most $2\log 2n$ times.

A random choice will only choose from these middle parts half the time, but this is good enough to give an average call depth of $2(2\log 2n)$.

And hence the expected time of $O(n\log n)$.

From Monte Carlo to Las Vegas!

Since (our) Las Vegas algorithms are Monte Carlo ones with an error probability of zero, we can construct Las Vegas ones from Monte Carlo.



If the success probability is $p(n)$, what's the expected run time?

Analysis ...

Let the number of iterations be denoted by the random variable X .

Independent choices are made on each iteration and X is said to be geometrically distributed.

Given success probability $p(n)$, the expectation $E[X]$ is:

$$\frac{1}{p(n)}$$

.

Hence, the expected run-time of the Las Vegas algorithm is:

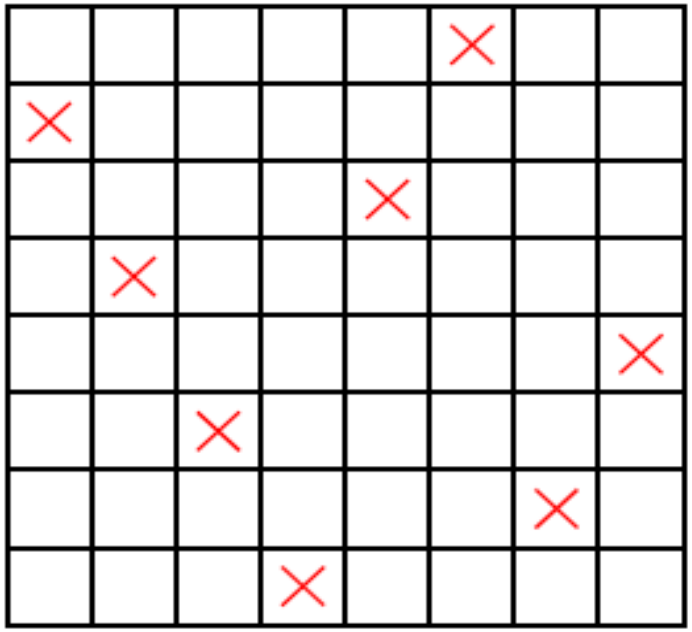
$$\frac{T(n) + t(n)}{p(n)}$$

Example: 8 queen problem

Place 8 queens on a chess board so that no one attacks another.

Remember: a queen attacks other pieces on the same row, same column and same diagonals.

A possible solution:



Backtracking algorithm

Place first queen in top-left corner.

Assume now the situation with some non-attacking queens in place.

If < 8 queens on board, place a queen in the next row — if it attacks a queen, move along the row one square at a time. If no position is possible, move the queen in the immediately preceeding row on one square. If no position is suitable, move the queen in the next row back, etc..

This algorthm actually returns a solution after examining 114 of the 2057 possible states.

A Las Vegas approach { non-backtracking}

Assume that k rows, $0 \leq k \leq 8$, are successfully occupied by queens.

If $k = 8$ then stop with success.
Otherwise, proceed to occupy row $k + 1$.

Calculate all positions on this row not attacked by existing queens.
If there are none, then fail.
Otherwise, pick one at random , and continue to next row.

Note, there is no backtracking, the algorithm simply fails if a queen can't be placed.

BUT this can be repeated, and will consider a probably different placement.

Random eight queens | cont.

The probability of success for the random placement is $0.1293\ldots$, i.e. approximately 1 in 8 times, just by guessing.

The expected number of states explored can be calculated to be around 56 , compared with 114 for the deterministic algorithm.

Further improvements can be made by combining random placement with some backtracking, first fixing some queens at random and then completing the arrangement via backtracking.

When the first two rows are occupied at random, then the expected number of states explored to find a solution becomes just 29.

Monte Carlo algorithms { characteristics}

Computational Complexity of A Randomized Algorithm

- If A is a randomized algorithm, then its running time on a fixed instance I of size n may vary from one execution to another.

- Therefore, a more natural measure of performance is the expected running time of A on a fixed instance I.

- This is the mean time taken by algorithm A to solve the instance I over and over. Thus, it is natural to talk about the worst case expected time and the average case expected time.

COMPLEXITY CLASSES

Complexity Classes

There are some interesting complexity classes involving randomized algorithms:

- **Randomized Polynomial time (*RP*)**
- **Zero-error Probabilistic Polynomial time (*ZPP*)**
- **Probabilistic Polynomial time (*PP*)**
- **Bounded-error Probabilistic Polynomial time (*BPP*)**

Randomized Polynomial time (*RP*)

Definition: The class *RP* consists of all languages *L* that have a randomized algorithm *A* running in worst-case polynomial time such that for any input *x* in $\sum^*$:

$$x \in L \Rightarrow \Pr[A(x) \text{ accepts}] \geq \frac{1}{2}$$

$$x \notin L \Rightarrow \Pr[A(x) \text{ accepts}] = 0$$

- Independent repetitions of the algorithms can be used to reduce the probability of error to exponentially small.
- Notice that the success probability can be changed to an inverse polynomial function of the input size without affecting the definition of *RP*. Why?

Zero-error Probabilistic Polynomial time (*ZPP*)

Definition: The class *ZPP* is the class of languages which have Las Vegas algorithms running in expected polynomial time.

ZPP = *RP* ∩ co-*RP*. Why?

(Note that a language *L* is in co-*X* where *X* is a complexity class if and only if it's complement $\sum^*$-*L* is in *X*.)

Probabilistic Polynomial time (*PP*)

Definition: The class *PP* consists of all languages *L* that have a randomized algorithm *A* running in worst-case polynomial time such that for any input *x* in $\sum^*$:

$$x \in L \Rightarrow \Pr[A(x) \text{ accepts}] \geq \frac{1}{2}$$

$$x \notin L \Rightarrow \Pr[A(x) \text{ accepts}] < \frac{1}{2}$$

Probabilistic Polynomial time (*PP*)

- To reduce the error probability, we can repeat the algorithm several times on the same input and produce the output which occurs in the majority of those trials.

- However, the definition of *PP* is quite weak since we have no bound on how far from ½ the probabilities are. It may not be possible to use a small number (e.g., polynomial no.) of repetitions to obtain a significantly small error probability.

Question

Consider a randomized algorithm with 2-sided error as in the definition of *PP*. Show that a polynomial no. of independent repetitions of this algorithm needs not suffice to reduce the error probability to ¼. (Hint: Consider the case where the error probability is $\frac{1}{2} - \frac{1}{2^n}$.)

Bounded-error Probabilistic Polynomial time (*BPP*)

Definition: The class *BPP* consists of all languages *L* that have a randomized algorithm *A* running in worst-case polynomial time such that for any input *x* in Σ^*:

$$x \in L \Rightarrow \Pr[A(x) \text{ accepts}] \geq \frac{3}{4}$$

$$x \notin L \Rightarrow \Pr[A(x) \text{ accepts}] \leq \frac{1}{4}$$

Bounded-error Probabilistic Polynomial time (*BPP*)

- For this class of algorithms, the error probability can be reduced to $\frac{1}{2^n}$ with only a polynomial number of iterations.
- In fact, the probability bounds $\frac{3}{4}$ and $\frac{1}{4}$ can be changed to $\frac{1}{2} + 1/p(n)$ and $\frac{1}{2} - 1/p(n)$ respectively where $p(n)$ is a polynomial function of the input size n without affecting the definition of *BPP*. Why?

APPLICATIONS

Randomized algorithms

- A randomized algorithm is just one that depends on random numbers for its operation
- These are randomized algorithms:
  - Using random numbers to help **find a solution** to a problem
  - Using random numbers to **improve a solution** to a problem
- These are related topics:
  - Getting or generating “random” numbers
  - Generating random data for testing (or other) purposes

Pseudorandom numbers

- The computer is *not capable* of generating truly random numbers
  - The computer can only generate pseudorandom numbers--numbers that are generated by a formula
  - Pseudorandom numbers *look* random, but are perfectly predictable if you know the formula
    - Pseudorandom numbers are good enough for most purposes, but not all--for example, not for serious security applications
  - Devices for generating truly random numbers do exist
    - They are based on radioactive decay, or on lava lamps
- "Anyone who attempts to generate random numbers by deterministic means is, of course, living in a state of sin."

  —John von Neumann

Generating random numbers

- Perhaps the best way of generating “random” numbers is by the linear congruential method:
  - r = (a * r + b) % m;
    where a and b are large prime numbers, and m is 2^{32} or 2^{64}
  - The initial value of r is called the seed
  - If you start over with the same seed, you get the same sequence of “random” numbers
- One advantage of the linear congruential method is that it will (eventually) cycle through all possible numbers
- Almost any “improvement” on this method turns out to be worse
  - “Home-grown” methods typically have much shorter cycles
- There are complex mathematical tests for “randomness” which you should know if you want to “roll your own”

Getting (pseudo)random numbers in Java

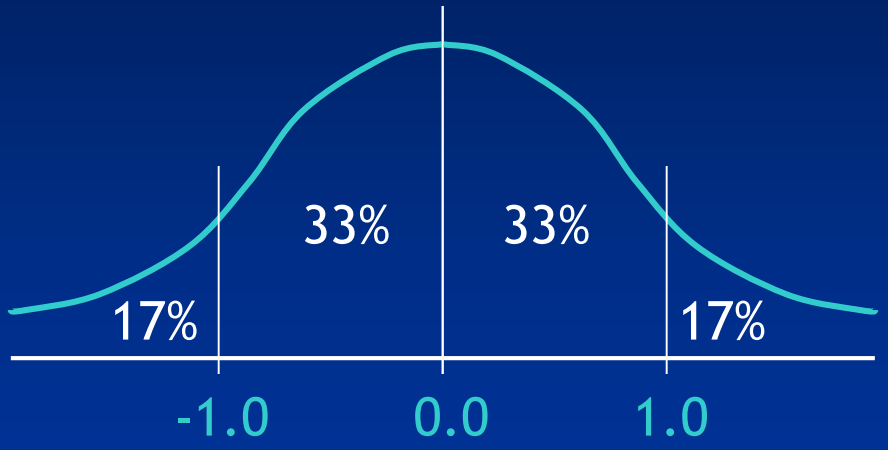
- import java.util.Random;

- new Random(long seed) // constructor
- new Random() // constructor, uses System.timeInMillis() as seed

- void setSeed(long seed)
- nextBoolean()
- nextFloat(), nextDouble() // 0.0 ≤ return value < 1.0
- nextInt(), nextLong() // all 2^{32} or 2^{64} possibilities
- nextInt(int n) // 0 ≤ return value < n
- nextGaussian()
  - Returns a double, normally distributed with a mean of 0.0 and a standard deviation of 1.0

Gaussian distributions

- The Gaussian distribution is a normal curve with mean (average) of 0.0 and standard deviation of 1.0
- It looks like this:



- Most real "natural" data—shoe sizes, quiz scores, amount of rainfall, length of life, etc.—fit a normal curve
- To generate realistic data, the Gaussian *may* be what you want:
  `r = desiredStandardDeviation * random.nextGaussian() + desiredMean;`
- "Unnatural data," such as income or taxes, may or may not be well described by a normal curve

Examples on randomization

Randomizing an array or Shuffling an array

- Good:

```
static void shuffle(int[] array) {
    for (int i = array.length; i > 0; i--) {
        int j = random.nextInt(i);
        swap(array, i - 1, j);
    }
} // all permutations are equally likely
```

- Bad:

```
static void shuffle(int[] array) {
    for (int i = 0; i < array.length; i++) {
        int j = random.nextInt(array.length);
        swap(array, i, j);
    }
} // all permutations are not equally likely
```

Checking randomness

- With a completely random shuffle, the probability that an element will end up in a given location is equal to the probability that it will end up in any other location
- I claim that the second algorithm on the preceding slide is *not* completely random
- I don't know how to prove this, but I can demonstrate it

  - Declare an array **counts** of 20 locations, initially all zero
  - Do the following 10000 times:
    - Fill an array of size 20 with the numbers 1, 2, 3, ..., 20
    - Shuffle the array
    - Find the location **loc** at which the 5 (or some other number) ends up
    - Add 1 to **counts[loc]**
  - Print the **counts** array

Randomly choosing N things

- Some students had an assignment in which they had to choose exactly N *distinct* random locations in an array
- Their algorithm was something like this:

```
for i = 1 to n {
    choose a random location x
    if x was previously chosen, start over
}
```

- Can you do better?

Generating random numbers in C

```
/*  * rand: Generates 5 numbers using standard "srand()/rand()" function
 * * SAMPLE OUTPUT:    rand[0]= 824522256,  rand[1]= 1360907941, rand[2]= 1513675795,
 rand[3]= 1046462087,  rand[4]= 253823980  */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
 int
main (int argc, char *argv[])
{
  /* Simple "srand()" seed: just use "time()" */
  unsigned int iseed = (unsigned int)time(NULL);
  srand (iseed);
   /* Now generate 5 pseudo-random numbers */
  int i;
  for (i=0; i<5; i++)
  {
    printf ("rand[%d]= %u\n",
      i, rand ());
  }
  return 0;
}
```

Random Numbers Using Computer

- Problem solving With the advent of computers, programmers recognized the need for a means of introducing randomness into a computer program.
- However, surprising as it may seem, it is difficult to get a computer to do something by chance.
- A computer follows its instructions blindly and is therefore completely predictable. (A computer that doesn't follow its instructions in this manner is broken.) There are two main approaches to generating random numbers using a computer:
  - **Pseudo-Random Number Generators (PRNGs)**
  - **True Random Number Generators (TRNGs).**

Pseudo-Random Number Generators (PRNGs)

- As the word ‘pseudo’ suggests, pseudo-random numbers are not random in the way you might expect, at least not if you're used to dice rolls or lottery tickets. Essentially,
- PRNGs are algorithms that use mathematical formulae or simply precalculated tables to produce sequences of numbers that appear random.
- A good example of a PRNG is the linear congruential method.
- A good deal of research has gone into pseudo-random number theory, and modern algorithms for generating pseudo-random numbers are so good that the numbers look exactly like they were really random.

The basic difference between PRNGs and TRNGs

- The basic difference between **PRNGs** and **TRNGs** is easy to understand if you compare computer-generated random numbers to rolls of a die.
- Because **PRNGs** generate random numbers by using mathematical formulae or pre-calculated lists, using one corresponds to someone rolling a die many times and writing down the results.
- Whenever you ask for a die roll, you get the next on the list. Effectively, the numbers appear random, but they are really predetermined.
- **TRNGs** work by getting a computer to actually roll the die — or, more commonly, use some other physical phenomenon that is easier to connect to a computer than a die is.

Pseudo-Random Number Generators (PRNGs)

- PRNGs are *efficient*, meaning they can produce many numbers in a short time, and *deterministic*, meaning that a given sequence of numbers can be reproduced at a later date if the starting point in the sequence is known.
- Efficiency is a nice characteristic if your application needs many numbers, and determinism is handy if you need to replay the same sequence of numbers again at a later stage.
- PRNGs are typically also *periodic*, which means that the sequence will eventually repeat itself. While periodicity is hardly ever a desirable characteristic, modern PRNGs have a period that is so long that it can be ignored for most practical purposes
- These characteristics make PRNGs suitable for applications where many numbers are required and where it is useful that the same sequence can be replayed easily.
- **Popular examples of such applications are simulation and modeling applications.**
- **PRNGs are not suitable for applications where it is important that the numbers are really unpredictable, such as data encryption and gambling.**

Pseudo-Random Number Generators (PRNGs)

- It should be noted that even though good PRNG algorithms exist, they aren't always used, and it's easy to get nasty surprises.
- Take the example of the popular web programming language PHP. If you use PHP for GNU/Linux, chances are you will be perfectly happy with your random numbers.
- However, if you use PHP for Microsoft Windows, you will probably find that your random numbers aren't quite up to scratch as shown in this visual analysis from 2008.
- Another example dates back to 2002 when one researcher reported that the PRNG on MacOS was not good enough for scientific simulation of virus infections. The bottom line is that even if a PRNG will serve your application's needs, you still need to be careful about which one you use.

True Random Number Generators (TRNGs)

- In comparison with PRNGs, TRNGs extract randomness from physical phenomena and introduce it into a computer. You can imagine this as a die connected to a computer, but typically people use a physical phenomenon that is easier to connect to a computer than a die is.
- The physical phenomenon can be very simple, like the little variations in somebody's mouse movements or in the amount of time between keystrokes.
- In practice, however, you have to be careful about which source you choose. For example, it can be tricky to use keystrokes in this fashion, because keystrokes are often buffered by the computer's operating system, meaning that several keystrokes are collected before they are sent to the program waiting for them. To a program waiting for the keystrokes, it will seem as though the keys were pressed almost simultaneously, and there may not be a lot of randomness there after all.

Ways to get true randomness into the computer.

- A really good physical phenomenon to use is a radioactive source.
- The points in time at which a radioactive source decays are completely unpredictable, and they can quite easily be detected and fed into a computer, avoiding any buffering mechanisms in the operating system.
- The HotBits service at Fourmilab in Switzerland is an excellent example of a random number generator that uses this technique.
- Another suitable physical phenomenon is atmospheric noise, which is quite easy to pick up with a normal radio. This is the approach used by RANDOM.ORG.
- You could also use background noise from an office or laboratory, but you'll have to watch out for patterns.
- The fan from your computer might contribute to the background noise, and since the fan is a rotating device, chances are the noise it produces won't be as random as atmospheric noise

Comparison of PRNGs and TRNGs

Characteristic	Pseudo-Random Number Generators	True Random Number Generators
Efficiency	Excellent	Poor
Determinism	Deterministic	Non- determinstic
Periodicity	Periodic	Aperiodic

These characteristics make TRNGs suitable for roughly the set of applications that PRNGs are unsuitable for, such as data encryption, games and gambling.

Conversely, the poor efficiency and nondeterministic nature of TRNGs make them less suitable for simulation and modeling applications, which often require more data than it's feasible to generate with a TRNG.

Applications vs. best type of generator

Application	Most Suitable Generator
Lotteries and Draws	TRNG
Games and Gambling	TRNG
Random Sampling (e.g., drug screening)	TRNG
Simulation and Modelling	PRNG
Security (e.g.,generation of data encryption keys)	TRNG
The Arts	Varies

Sources of random number

- RANDOM.ORG is a true random number service that generates randomness via atmospheric noise.

Sources of random number

- As long as you are careful, the possibilities are endless. Undoubtedly the visually coolest approach was the lavarand generator, which was built by Silicon Graphics and used snapshots of lava lamps to generate true random numbers. Unfortunately, lavarand is no longer operational, but one of its inventors is carrying on the work (without the lava lamps) at the LavaRnd web site. Yet another approach is the Java EntropyPool, which gathers random bits from a variety of sources including HotBits and RANDOM.ORG, but also from web page hits received by the EntropyPool's own web server.
- Regardless of which physical phenomenon is used, the process of generating true random numbers involves identifying little, unpredictable changes in the data. For example, HotBits uses little variations in the delay between occurrences of radioactive decay, and RANDOM.ORG uses little variations in the amplitude of atmospheric noise.

The characteristics of TRNGs

- The characteristics of TRNGs are quite different from PRNGs. First, TRNGs are generally rather *inefficient* compared to PRNGs, taking considerably longer time to produce numbers.
- They are also *nondeterministic*, meaning that a given sequence of numbers cannot be reproduced, although the same sequence may of course occur several times by chance. TRNGs have no period.

Use of Random Numbers

- Random numbers are useful for a variety of purposes, such as generating data encryption keys, simulating and modeling complex phenomena and for selecting random samples from larger data sets.
- They have also been used aesthetically, for example in literature and music, and are of course ever popular for games and gambling.
- When discussing single numbers, a random number is one that is drawn from a set of possible values, each of which is equally probable, i.e., a uniform distribution. When discussing a sequence of random numbers, each number drawn must be statistically independent of the others.

Application of Randomized algorithm

FINDING THE Π VALUE

Determining π

Square = 1

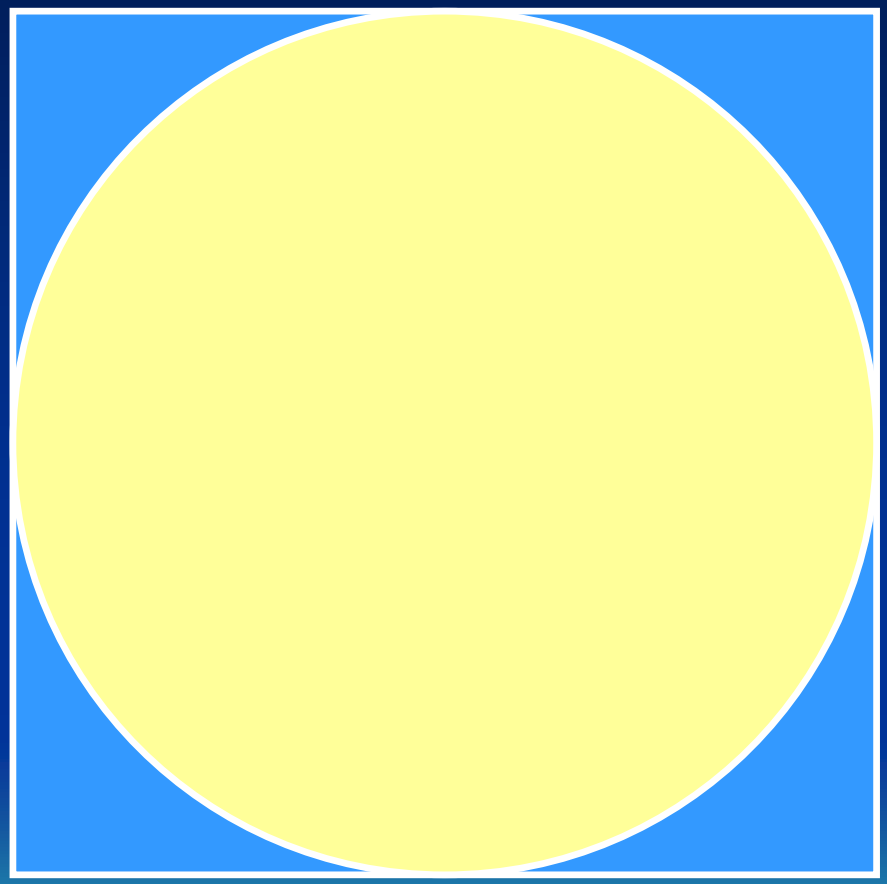
Circle = π/4

The probability

a random point

in square is in circle:

= π/4



π = 4 * points in circle/points

Determining π

```
def findPi (points):
    incircle = 0
    for i in range (points):
        x = random.random ()
        y = random.random ()
        if (square (x - 0.5) + square (y - 0.5) \
            < 0.25):  # 0.25 = r^2
            incircle = incircle + 1

    return 4.0 * incircle / points
```

Results

n: 1	4.0	4.0	0.0
n: 2	2.0	4.0	4.0
n: 4	3.0	4.0	3.0
...			
n: 64	3.0625	3.125	3.0625
...			
n: 1024	3.16796875	3.13671875	3.1640625
...			
n: 16384	3.12622070312	3.14038085938	3.1279296875
n: 131072	3.13494873047	3.14785766602	3.13766479492
n: 1048576	3.14015579224	3.14387893677	3.14112472534

If we wait long enough will it produce an arbitrarily accurate value?

Examples

The 0-1 knapsack problem

- Even if we don't use a randomized algorithm to *find* a solution, we might use one to *improve* a solution
- The 0-1 knapsack problem can be expressed as follows:
  - A thief breaks into a house, carrying a knapsack...
    - He can carry up to 25 pounds of loot
    - He has to choose which of N items to steal
      - Each item has some weight and some value
      - "0-1" because each item is stolen (1) or not stolen (0)
    - He has to select the items to steal in order to maximize the value of his loot, but cannot exceed 25 pounds
- A greedy algorithm does not find an optimal solution…but…
- We could use a greedy algorithm to find an initial solution, then use a randomized algorithm to try to improve that solution

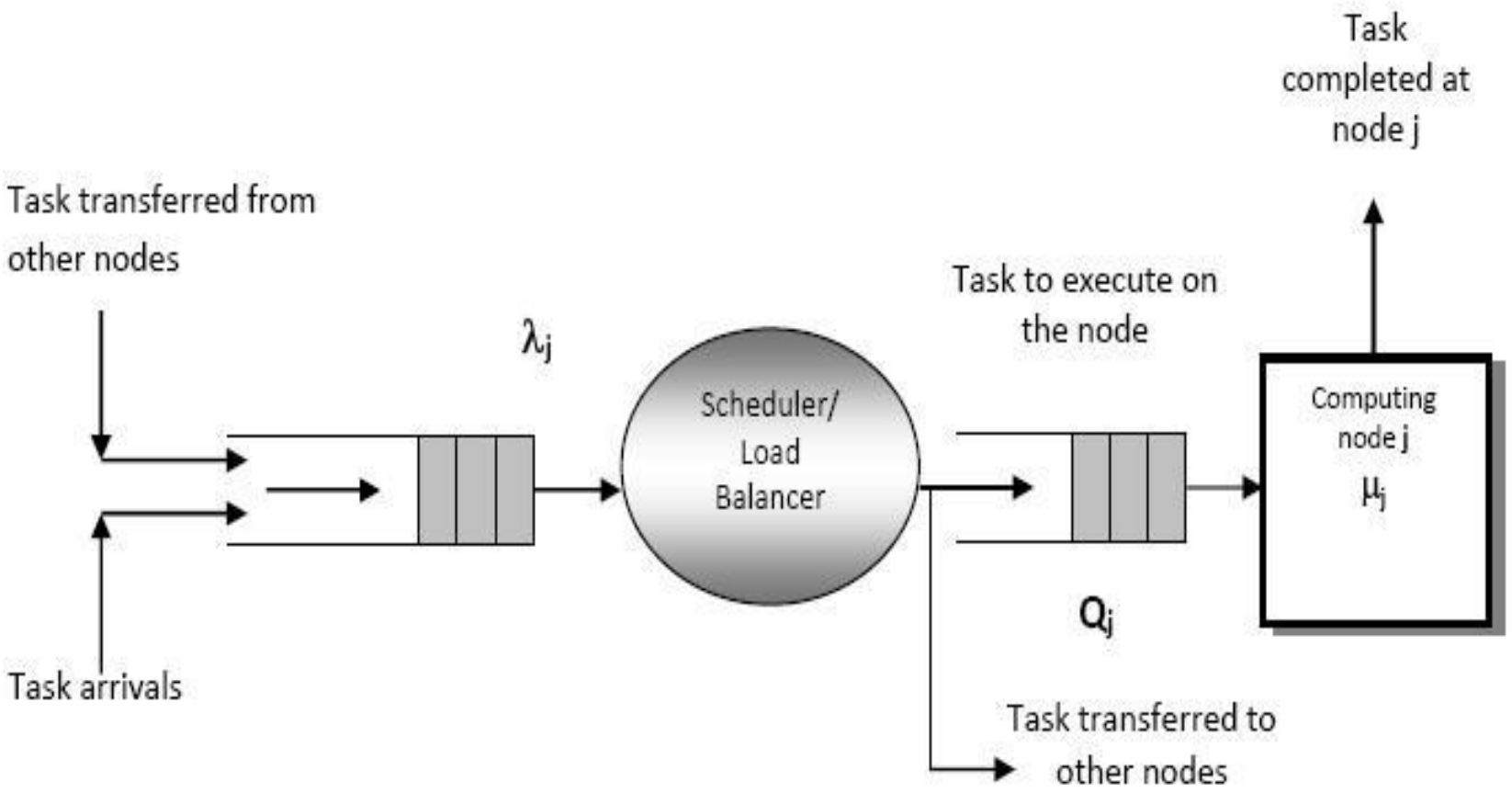
Improving the knapsack solution

- We can employ a greedy algorithm to fill the knapsack
- Then--
  - Remove a randomly chosen item from the knapsack
  - Replace it with other items (possibly using the same greedy algorithm to choose those items)
  - If the result is a better solution, keep it, else go back to the previous solution
  - Repeat this procedure as many times as desired
    - You might, for example, repeat it until there have been no improvements in the last k trials, for some value of k
- You probably won't get an optimal solution this way, but you might get a better one than you started with

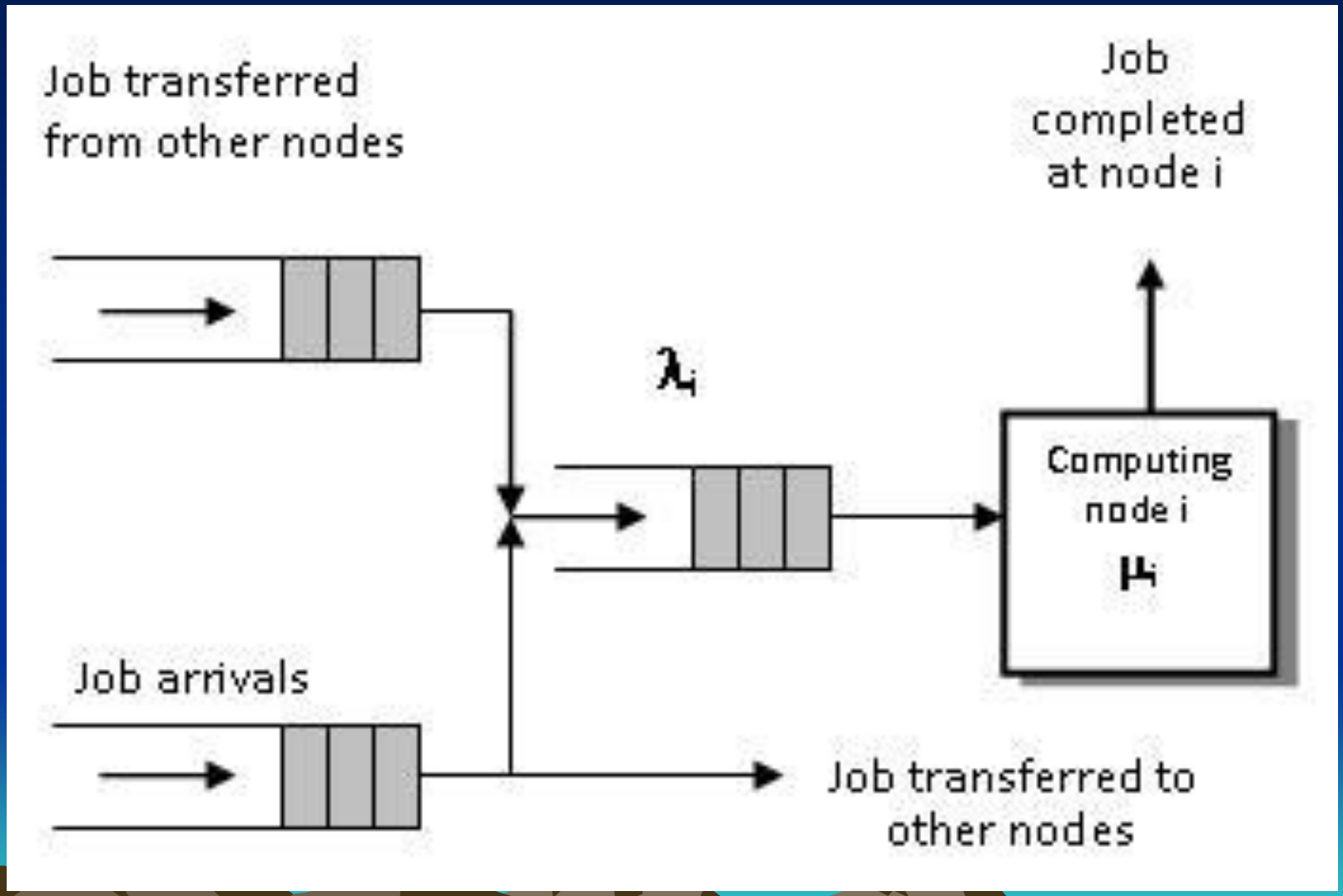
Queuing problems

- Suppose:
  - Customers arrive at a service desk at an *average* rate of 50 an hour
  - It takes one minute to serve each customer
- How long, on average, does a customer have to wait in line?
  - If you know **queuing theory**, you may be able to solve this
  - Otherwise, just write a program to try it out!
  - This kind of program is typically called a Monte Carlo method, and is a great way to avoid learning scads of math

Load balancer in HDCS



Queue at a Computing node



Notes

- Sometimes you want random numbers for their own sake—for example, you want to shuffle a virtual deck of cards to play a card game
- Sometimes you can use random numbers in a simulation (such as the queue example) to avoid difficult mathematical problems, or when there is no known feasible algorithm (such as the 0-1 knapsack problem)
- Randomized algorithms are basically *experimental*—you almost certainly won't get perfect or optimal results, but you can get "pretty good" results
- Typically, the longer you allow a randomized algorithm to run, the better your results
- A randomized algorithm is what you do when you don't know what else to do
  - **As such, it should be in every programmer's toolbox!**

Conclusions

- Employing randomness leads to improved simplicity and improved efficiency in solving the problem.
- However, assumes the availability of a perfect source of independent and unbiased random bits.
- Access to truly unbiased and independent sequence of random bits is expensive and should be considered as an expensive resource like time and space. One should aim to minimize the use of randomness to the extent possible.

Conclusions

- Assumes efficient realizability of any rational bias. However, this assumption introduces error and increases the work and the required number of random bits.

- There are ways to reduce the randomness from several algorithms while maintaining the efficiency nearly the same.


Thank you for your attention
Prof. Bibhudatta Sahoo 2016

Exercises

1. Finding MST in a graph with n vertices and m edges has a Las Vegas algorithm which has the expected running time O(n+m).
2. Devise an algorithm for the all-pairs shortest paths problem that does not use matrix multiplication and runs in time $O(n^{3-\varepsilon})$ for a positive constant ε.
3. Devise an algorithm for computing the diameter of an unweighted graph that does not use matrix multiplication and runs in time $O(n^{3-\varepsilon})$ for a positive constant ε.
4. Devise a Las Vegas or a deterministic algorithm for min-cuts with running time close to $O(n^2)$.
5. Is there a randomized algorithm for min-cuts with expected running time close to O(m)?

Exercises

- Given an undirected, connected multi-graph $G(V,E)$, we want to find a cut (V_1,V_2) such that the number of edges between V_1 and V_2 is minimum. This problem can be solved optimally by applying the max-flow min-cut algorithm $O(n^2)$ time by trying all pairs of source and destination. Write an randomized algorithm to find min-cut?
- What will happen if we apply a similar approach to find the max-cut instead? Will it be better or worse than the previous method of random assignment?

Exercises

- Show that in Randomized Selection, the expected number of elements comparisons performed by Algorithm RANDOMIZEDSELECT on input of size n is less than 4n. Its expected running time is Θ(n).

The Problem

Suppose we have n processes $P_1, P_2, \ldots, P_n$, each competing for access to a single shared database. We imagine time as being divided into discrete *rounds*. The database has the property that it can be accessed by at most one process in a single round; if two or more processes attempt to access it simultaneously, then all processes are "locked out" for the duration of that round. So, while each process wants to access the database as often as possible, it's pointless for all of them to try accessing it in every round; then everyone will be perpetually locked out. What's needed is a way to divide up the rounds among the processes in an equitable fashion, so that all processes get through to the database on a regular basis.

If it is easy for the processes to communicate with one another, then one can imagine all sorts of direct means for resolving the contention. But suppose that the processes can't communicate with one another at all; how then can they work out a protocol under which they manage to "take turns" in accessing the database?

Designing a Randomized Algorithm

Randomization provides a natural protocol for this problem, which we can specify simply as follows. For some number $p > 0$ that we'll determine shortly, each process will attempt to access the database in each round with probability p, independently of the decisions of the other processes. So, if exactly one process decides to make the attempt in a given round, it will succeed; if two or more try, then they will all be locked out; and if none try, then the round is in a sense "wasted." This type of strategy, in which each of a set of identical processes randomizes its behavior, is the core of the *symmetry-breaking* paradigm that we mentioned initially: If all the processes operated in lockstep, repeatedly trying to access the database at the same time, there'd be no progress; but by randomizing, they "smooth out" the contention.


Thank You!