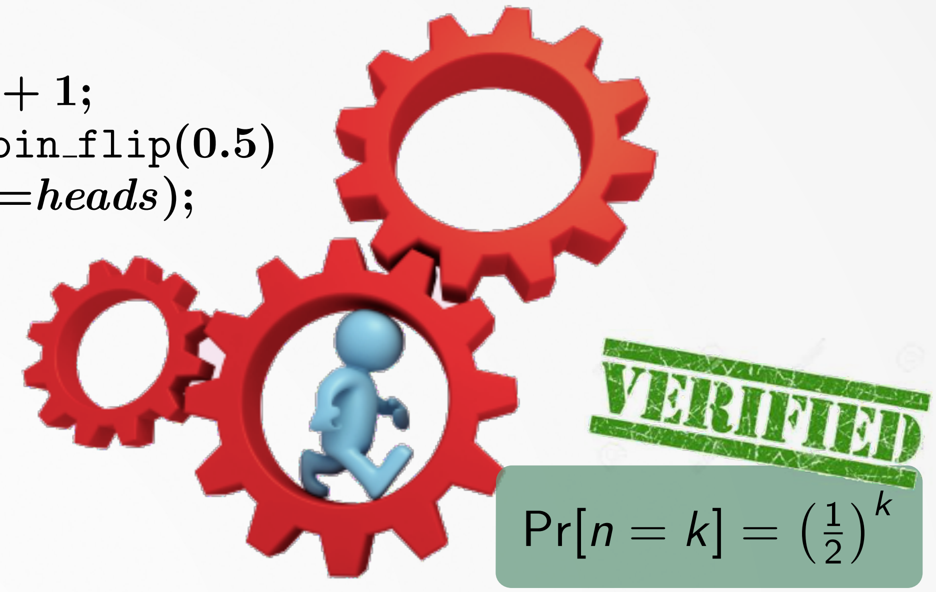
Seminar on

“Verification of Probabilistic Programs”

```
n := 0;
repeat
  n := n + 1;
  c := coin_flip(0.5)
until (c=heads);
return n
```



$\Pr[n = k] = \left(\frac{1}{2}\right)^k$

Lehrstuhl für Informatik 2
Softwaremodellierung und Verifikation

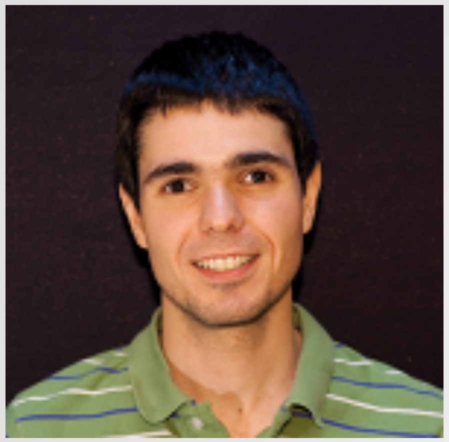
RWTH AACHEN UNIVERSITY

Federico Olmedo

federico.olmedo@cs.rwth-aachen.de

Seminar Details

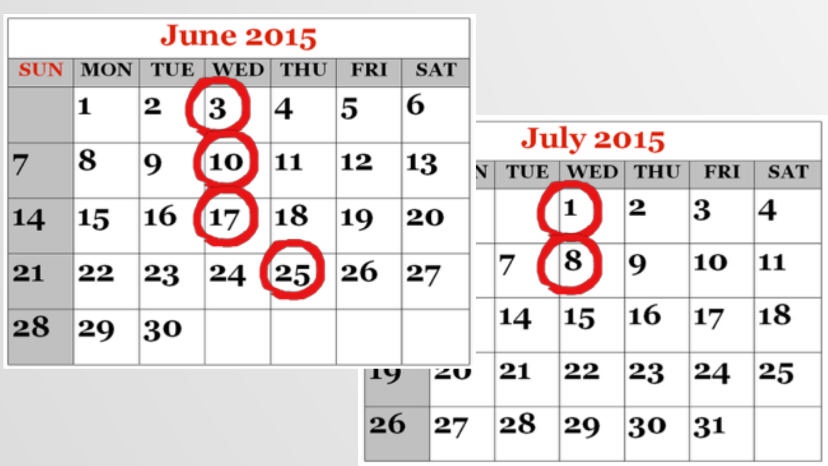
Speaker:



FEDERICO OLMEDO

Software Modelling and Verification Group
RWTH Aachen University

Structure & Schedule:



6 Weekly Presentations
16:30-17:45 Room 9U10 E3

Language:

Pre-requisites:

Previous knowledge on program logics and semantics is ONLY advised.

Webpage:

http://moves.rwth-aachen.de/teaching/ss-15/vpp/

Agenda

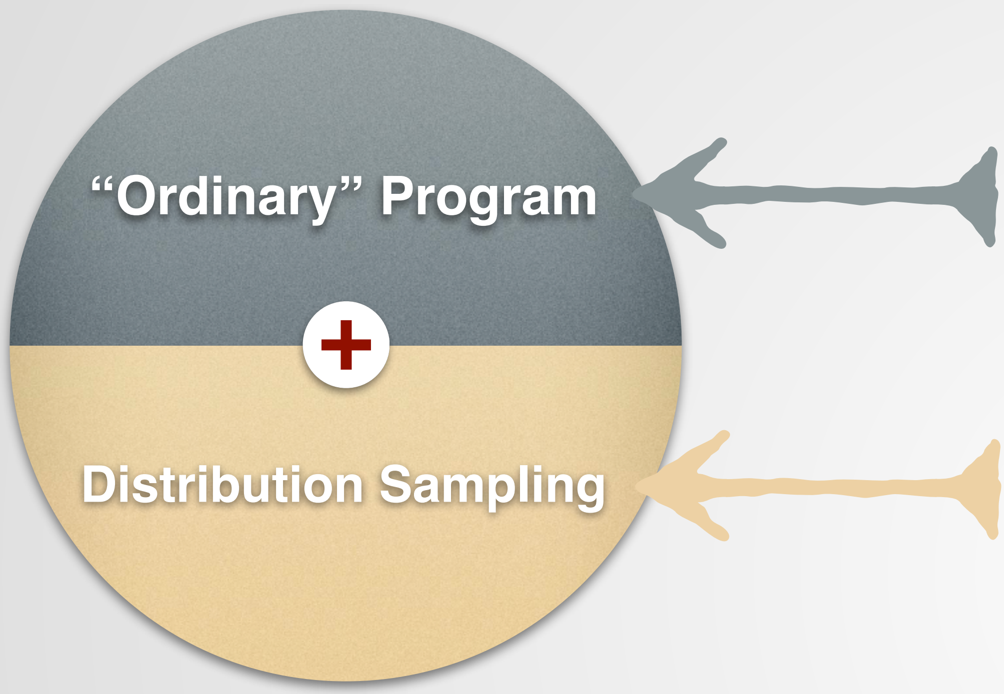
- Introduction to probabilistic programs
- The problem of probabilistic program verification
- Seminar content
- Summary

Agenda

- Introduction to probabilistic programs
- The problem of probabilistic program verification
- Seminar content
- Summary

Probabilistic Programs — Basics

What is a probabilistic program?

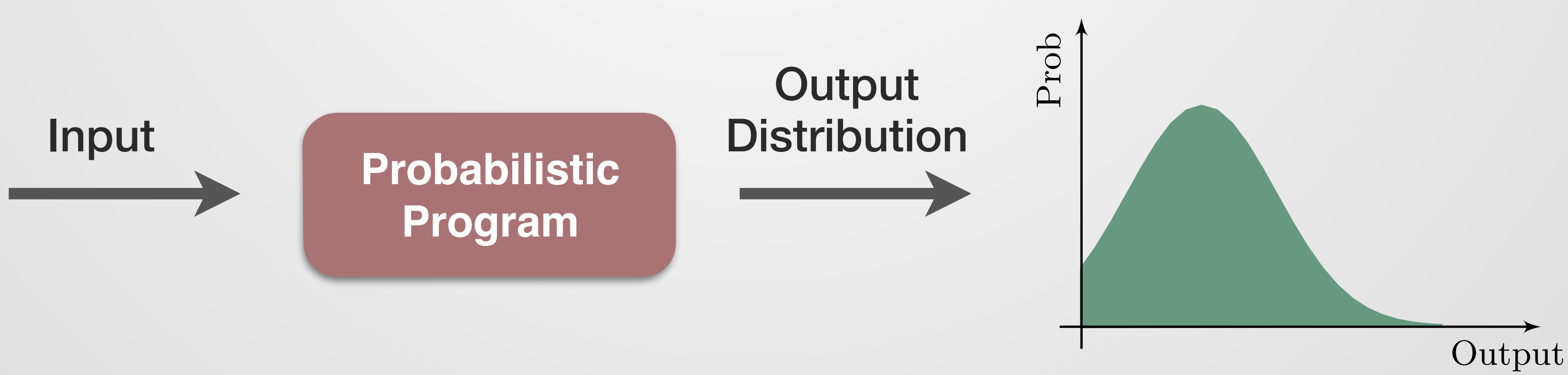


In this seminar

- **imperative** (eg. Probabilistic C)
- functional (eg. Church)
- logical (eg. CHRISM)
- …

- randomly choose a process with which communicate
- select a random prime in interval *[1, n^2]*
- flip a (fair/biased) coin;

Input/Output behaviour:



Probabilistic Programs — Examples

```
c1 := coin_flip(0.5);
c2 := coin_flip(0.5);
return (c1, c2)
```

```
n := 0;
repeat
   n := n + 1;
   c := coin_flip(0.5)
until (c=heads);
return n
```

Probabilistic Programs — Examples

c_1 := `coin_flip`(0.5);
c_2 := `coin_flip`(0.5);
`return` (c_1, c_2)

Output Distribution →

		c_1	
		h	t
c_2	h	1/4	1/4
	t	1/4	1/4

n := 0;
`repeat`
  n := $n + 1$;
  c := `coin_flip`(0.5)
`until` (c=*heads*);
`return` n

Probabilistic Programs — Examples

$c_1 := \texttt{coin\_flip}(0.5);$
$c_2 := \texttt{coin\_flip}(0.5);$
$\texttt{return}\ (c_1, c_2)$

Output Distribution

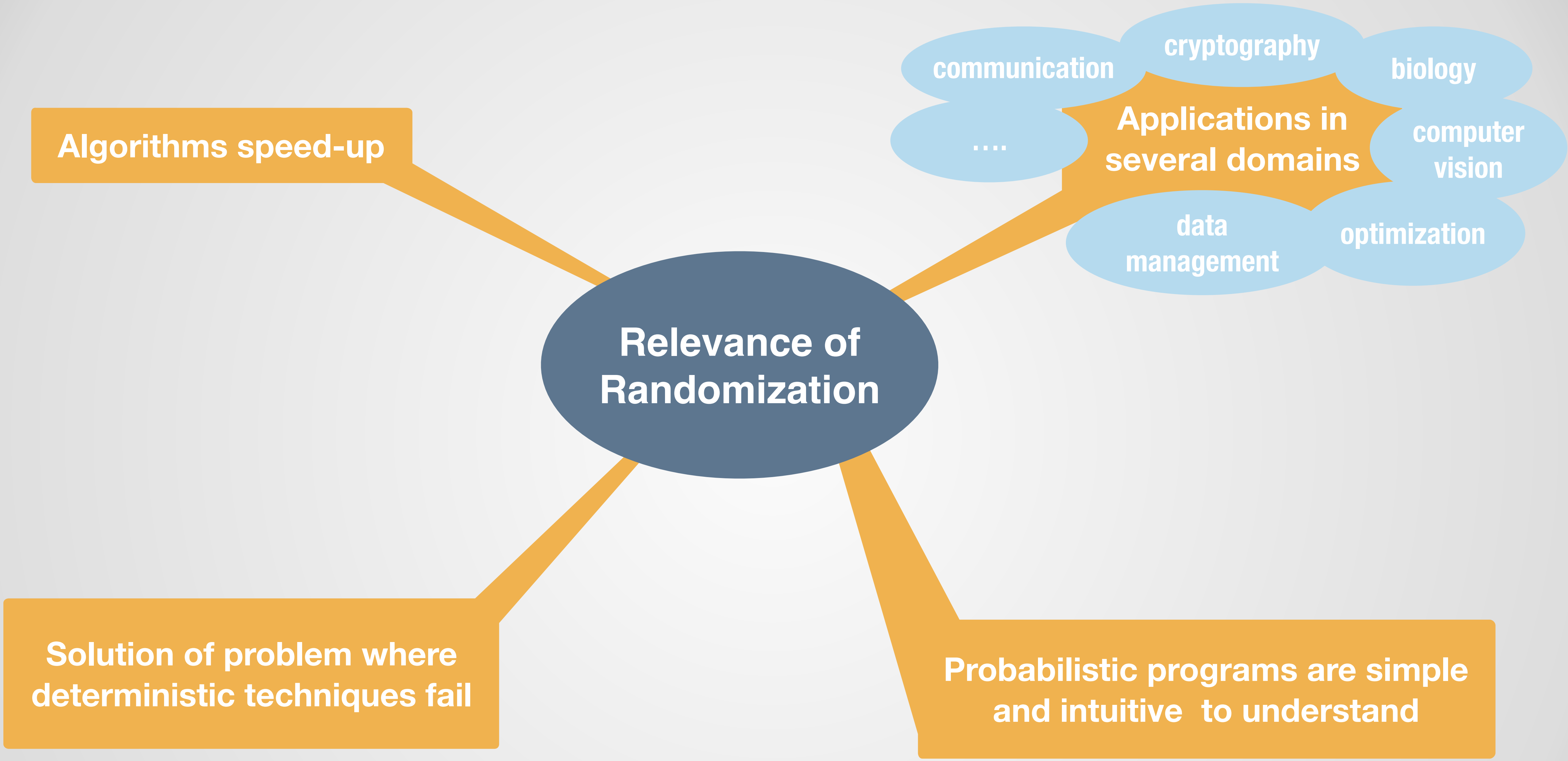
		c_1	
		h	t
c_2	h	1/4	1/4
	t	1/4	1/4

$n := 0;$
$\texttt{repeat}$
$\quad n := n + 1;$
$\quad c := \texttt{coin\_flip}(0.5)$
$\texttt{until}\ (c{=}\mathit{heads});$
$\texttt{return}\ n$

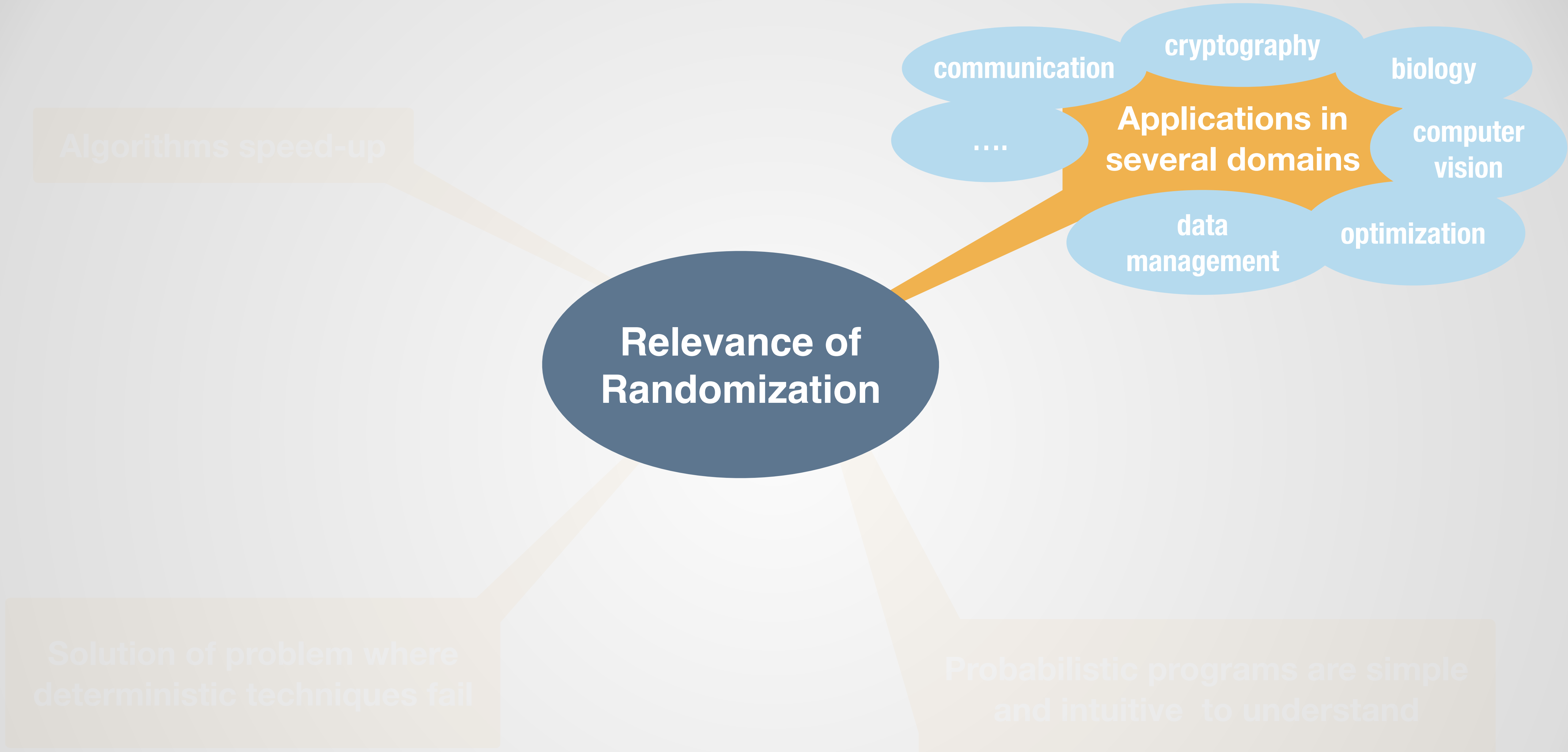
Output Distribution

n				
1	2	…	k	…
1/2	1/4		$1/2^k$	

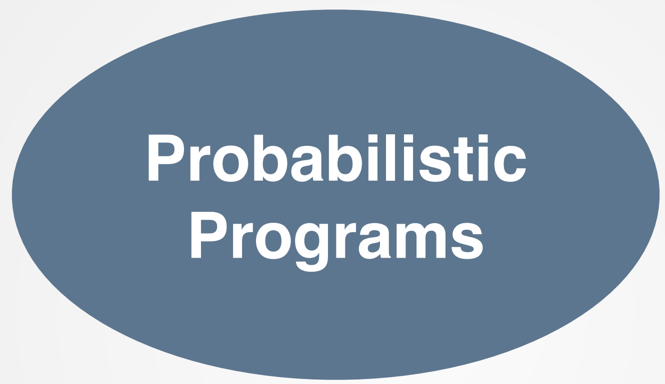
Probabilistic Programs — Relevance



Probabilistic Programs — Relevance



Probabilistic Programs — Application Domain



Probabilistic Programs — Application Domain

Skill Ranking System

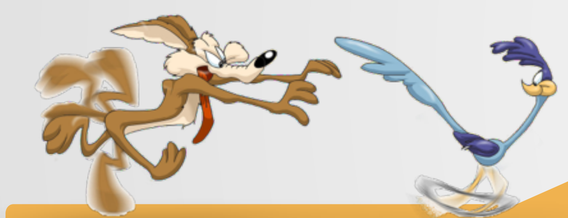
Probabilistic Programs

```
float skillA, skillB,skillC;
float perfA1,perfB1,perfB2,
      perfC2,perfA3,perfC3;
skillA = Gaussian(100,10);
skillB = Gaussian(100,10);
skillC = Gaussian(100,10);

// first game:A vs B, A won
perfA1 = Gaussian(skillA,15);
perfB1 = Gaussian(skillB,15);
observe(perfA1 > perfB1);
// second game:B vs C, B won
perfB2 = Gaussian(skillB,15);
perfC2 = Gaussian(skillC,15);
observe(perfB2 > perfC2);

// third game:A vs C, A won
perfA3 = Gaussian(skillA,15);
perfC3 = Gaussian(skillC,15);
observe(perfA3 > perfC3);
```

Probabilistic Programs — Application Domain

Skill Ranking System

Probabilistic Programs



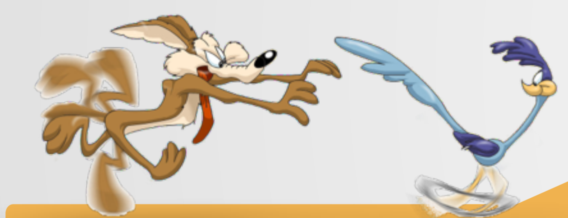
Predator—Prey Population Model

```
int goats, tigers;
double c1, c2, c3, curTime;
// initialize populations
goats = 100; tigers = 4;
// initialize reaction rates
c1 = 1; c2 = 5; c3 = 1;
//initialize time
curTime = 0;

while (curTime < TIMELIMIT)
{
  if (goats > 0 && tigers > 0)
  {
    double rate1, rate2, rate3,
           rate;
    rate1 = c1 * goats;
    rate2 = c2 * goats * tigers;
    rate3 = c3 * tigers;
    rate = rate1 + rate2 + rate3;

    double dwellTime =
               Exponential(rate);
    int discrete =
     Disc3(rate1/rate,rate2/rate);
    curTime += dwellTime;
    switch (discrete)
    {
      case 0: goats++; break;
      case 1: goats--; tigers++;
              break;
      case 2: tigers--; break;
    }
   }
  else if (goats > 0)
  {
    double rate;
    rate = c1 * goats;
    double dwellTime =
         Exponential(rate);
    curTime += dwellTime;
    goats++;
  }
   else if (tigers > 0)
  {
     double rate;
     rate = c3 * tigers;
     double dwellTime =
         Exponential(rate);
     curTime += dwellTime;
     tigers--;
  }
}//end while loop
return(goats,tigers);
}
```

Probabilistic Programs — Application Domain

Skill Ranking System

Probabilistic Programs



Predator—Prey Population Model

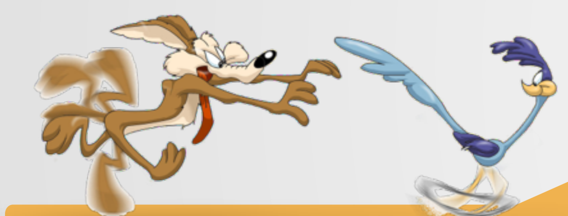
Dining Philosophers

```
DinPhil {(* algorithm for P_i *)
    WHILE TRUE DO{
        (* thinking section *)
        trying := true
        WHILE trying DO{
            choose s randomly and uniformly from {0, 1}
            wait until TEST & UPDATE(fork-available [i − s], FALSE, FALSE)
            IF TEST & UPDATE(fork-available[i − s̄], FALSE, FALSE) THEN
                trying := FALSE (* s̄ = complement of s *)
            ELSE fork-available[i − s] := TRUE
        }
        (* eating section *)
        fork-available[i − 1], fork-available[i] := TRUE
    }
}
```

Probabilistic Programs — Application Domain

Probabilistic Programs

Skill Ranking System



Predator—Prey Population Model

Dining Philosophers

Miller—Rabin Primality Test

MILLERRABIN(n)

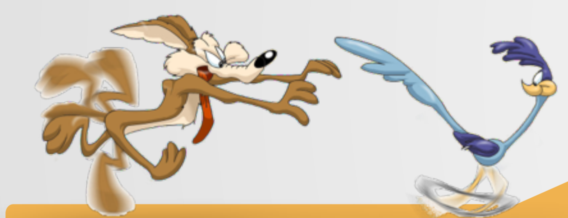
If $n > 2$ and n is even, return **composite.**
/* Factor $n-1$ as $2^s t$ where t is odd. */
$s \leftarrow 0$
$t \leftarrow n-1$
while t is even
    $s \leftarrow s+1$
    $t \leftarrow t/2$
end /* Done. $n-1 = 2^s t$. */
Choose $x \in \{1, 2, \ldots, n-1\}$ uniformly at random.
Compute each of the numbers $x^t, x^{2t}, x^{4t}, \ldots, x^{2^s t} = x^{n-1} \bmod n$.
If $x^{n-1} \not\equiv 1 \pmod{n}$, return **composite.**
for $i = 1, 2, \ldots, s$
    If $x^{2^i t} \equiv 1 \pmod{n}$ and $x^{2^{i-1} t} \not\equiv \pm 1 \pmod{n}$, return **composite.**
end /* Done checking for fake square roots. */
Return **probably prime.**

Probabilistic Programs — Application Domain

Probabilistic Programs

....

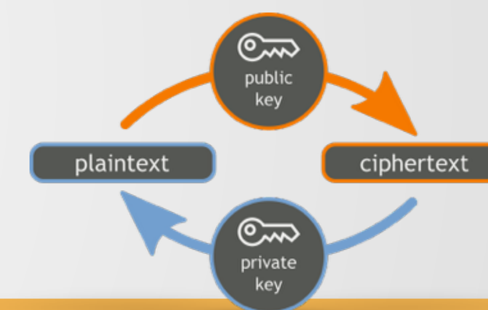
Skill Ranking System



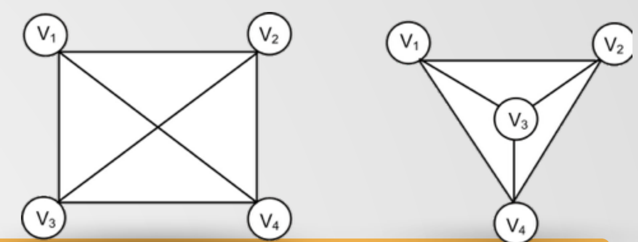
Predator—Prey Population Model

Dining Philosophers

Miller—Rabin Primality Test

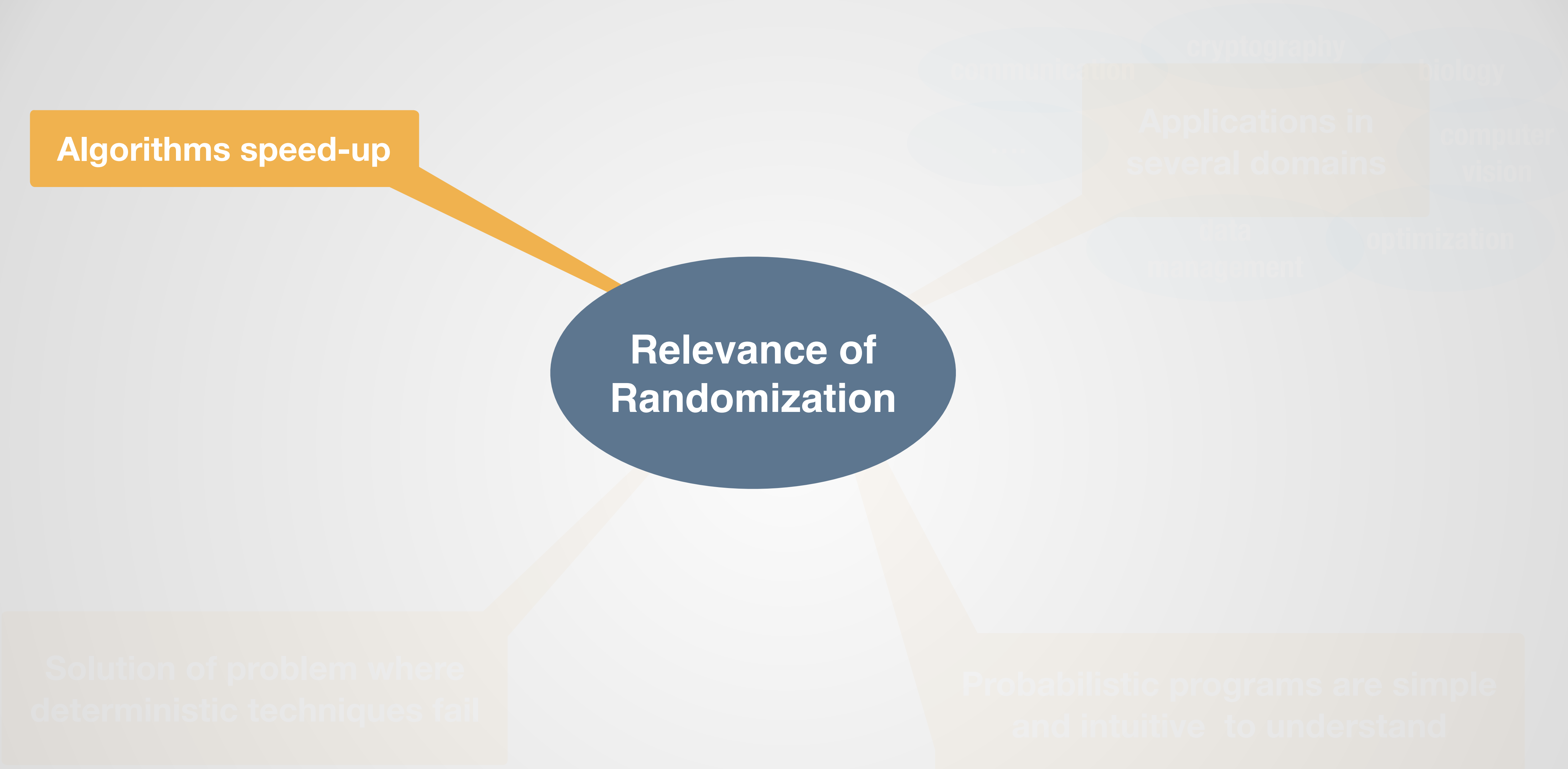


Public-Key Encryption



Graph Isomorphism

Probabilistic Programs — Relevance



Randomization allows speeding up algorithms

Quicksort:

$\mathsf{QS}(A) \triangleq$
    $\texttt{if}\,(|A| \leq 1)\,\texttt{then return}\,(A);$
    $i := \lfloor |A|/2 \rfloor;$
    $A_< := \{a' \in A \mid a' < A[i]\};$
    $A_> := \{a' \in A \mid a' > A[i]\};$
    $\texttt{return}\,\big(\mathsf{QS}(A_<) +\!\!+\; A[i] +\!\!+\; \mathsf{QS}(A_>)\big)$

Randomization allows speeding up algorithms

Quicksort:

$\mathrm{QS}(A) \triangleq$

$\quad$ `if` $(|A| \leq 1)$ `then return` (A);

$\quad i := \lfloor |A|/2 \rfloor$;

$\quad A_< := \{a' \in A \mid a' < A[i]\}$;

$\quad A_> := \{a' \in A \mid a' > A[i]\}$;

$\quad$ `return` $\big(\mathrm{QS}(A_<) +\!\!+\ A[i] +\!\!+\ \mathrm{QS}(A_>)\big)$

Problem of Quicksort:

In the average case, it performs fairly well:

On a random input of size *n*, it requires on average ***O****(n log(n))* comparisons (which matches information theory lower bound).

But in the worst case, it does not:

There exist “ill-behaved” inputs of size *n* which require ***O****(n²)* comparisons.

How to narrow the gap between the worst and average case performance?

Randomization allows speeding up algorithms

Quicksort:

$$
\begin{aligned}
&\mathsf{rQS}(A) \triangleq \\
&\quad \texttt{if}\ (|A| \leq 1)\ \texttt{then return}\ (A); \\
&\quad i := \mathsf{rand}[1 \ldots |A|]; \\
&\quad A_< := \{a' \in A \mid a' < A[i]\}; \\
&\quad A_> := \{a' \in A \mid a' > A[i]\}; \\
&\quad \texttt{return}\ \big(\mathsf{QS}(A_<) +\!\!+\ A[i] +\!\!+\ \mathsf{QS}(A_>)\big)
\end{aligned}
$$

Problem of Quicksort:

In the average case, it performs fairly well:

On a random input of size *n*, it requires on average ***O****(n log(n))* comparisons (which matches information theory lower bound).

But in the worst case, it does not:

There exist “ill-behaved” inputs of size *n* which require ***O****(n*^{2}*)* comparisons.

How to narrow the gap between the worst and average case performance?

Choose the pivot at random!

For any input, the expected number of comparisons matches the average case.

No “ill-behaved” input.

Randomization allows speeding up algorithms

Computing the cardinality of the union of sets:

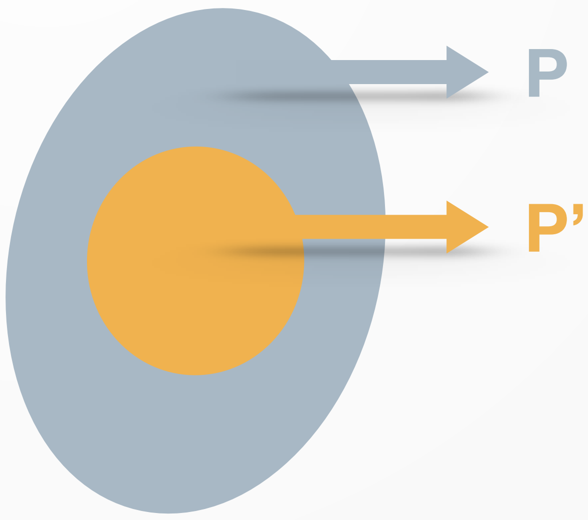
$$|S_1 \cup S_2 \cup \ldots \cup S_n| \;=\; \sum_i |S_i| - \sum_{i<j} |S_i \cap S_j| + \sum_{i<j<k} |S_i \cap S_j \cap S_k| - \ldots \qquad \begin{pmatrix}\text{Incl-Excl}\\ \text{Principle}\end{pmatrix}$$

Problem: Incl-Excl Principle yields an expensive solution, the RHS has $2^n - 1$ terms.

Solution based on randomization:

Random Sampling Technique

We can approximate some properties of a set from a randomly chosen subset.



P' estimates P

Randomization allows speeding up algorithms

Computing the cardinality $|S_1 \cup S_2 \cup \ldots \cup S_n|$

Solution based on randomization: sample an element $x^* \in S_1 \cup S_2 \cup \ldots \cup S_n$ and use $cov(x^*) = |\{i \mid x^* \in S_i\}|$ to estimate $|S_1 \cup S_2 \cup \ldots \cup S_n|$.

$m := |S_1| + \ldots + |S_n|$;
Draw a set S^* from $S_1, \ldots, S_n$ with probability $\Pr[S_i] = \frac{|S_i|}{m}$;
Draw an element x^* from S^* with uniform distribution;
$r := \frac{m}{cov(x^*)}$;
return (r)

It can be shown that r is an unbiased estimator of $|S_1 \cup S_2 \cup \ldots \cup S_n|$, *ie*

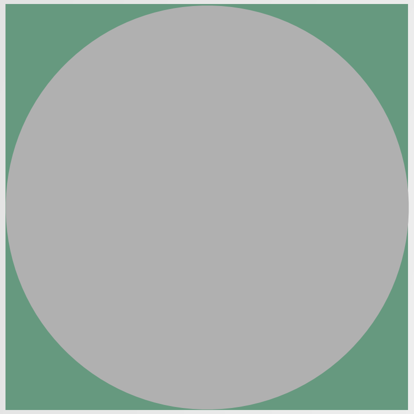
$$\mathbb{E}[r] = |S_1 \cup S_2 \cup \ldots \cup S_n|$$

Expected value of r

Randomization allows speeding up algorithms

Another application of the Random Sampling technique

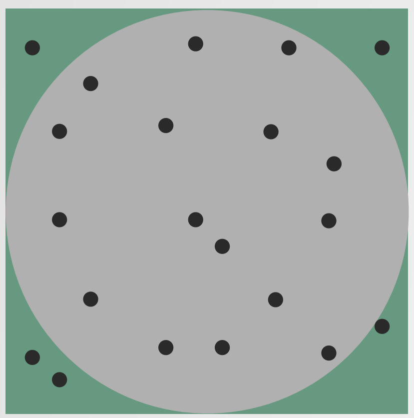
- Approximate the area of a circle



Randomization allows speeding up algorithms

Another application of the Random Sampling technique

- Approximate the area of a circle



Sample random points in the enclosing square. The fraction of points lying in the circle approximates its area.

$$Area(\bullet) \approx Area(\blacksquare) \cdot \frac{N_{\bullet}}{N_{\blacksquare}}$$

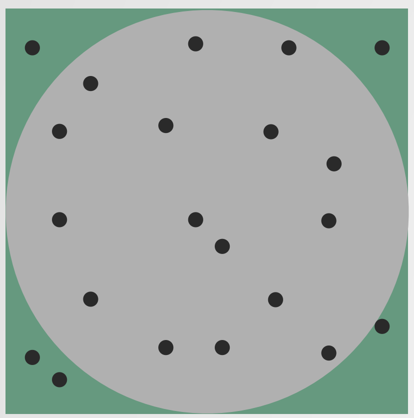
Nr of points hitting the circle

Total nr of random points

(The approximation improves as $N_{\blacksquare}$ grows larger.)

Another application of the Random Sampling technique

- Approximate the area of a circle



Sample random points in the enclosing square. The fraction of points lying in the circle approximates its area.

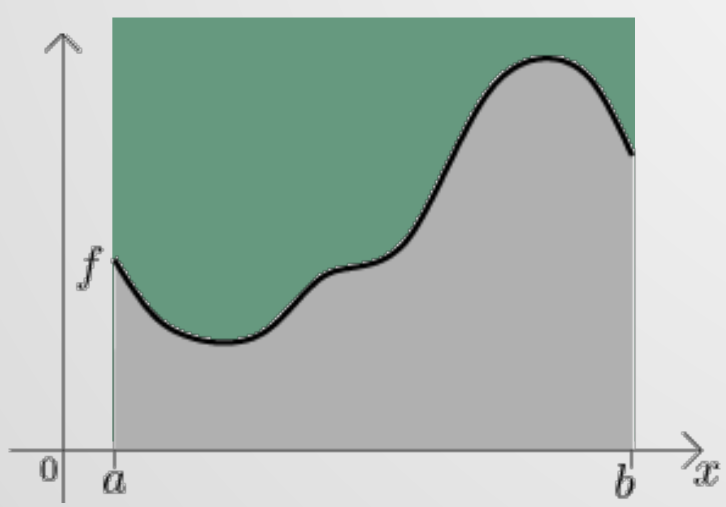
$$Area(\bullet) \approx Area(\blacksquare) \cdot \frac{N_{\bullet}}{N_{\blacksquare}}$$

Nr of points hitting the circle

Total nr of random points

(The approximation improves as $N_{\blacksquare}$ grows larger.)

- Approximate a definite integral



$$\int_a^b f(x)\,\mathrm{d}x \approx Area(\blacksquare) \cdot \frac{N_{\blacktriangle}}{N_{\blacksquare}}$$

Randomization allows speeding up algorithms

Testing against the null polynomial

Assume we have oracle access $\mathcal{O}(\cdot, \ldots, \cdot)$ to a multivariate polynomial p in $\mathbb{R}[x_1, \ldots, x_n]$. Determine whether p is identically zero.

Solution based on randomization:

Exploit the fact if $p \not\equiv 0$ and $\bar{a} = (a_1, \ldots, a_n)$ is chosen at random, $\Pr[p(\bar{a}) = 0]$ is small.

Theorem (Schwartz-Zippel): let $S \subseteq \mathbb{R}$ with $|S| = k \cdot deg(p)$. If each component of $\bar{a} = (a_1, \ldots, a_n)$ is chosen independently and uniformly from S, then

$$\Pr[p(\bar{a}) = 0 \mid p \not\equiv 0] \leq 1/k \ .$$

Let $S \subseteq \mathbb{R}$ with $|S| = k \cdot deg(p)$
Draw $a_1, \ldots, a_n$ independently and uniformly from S;
`if` $\mathcal{O}(a_1, \ldots, a_n) = 0$ `then return` ("$p \equiv 0$")
`else return` ("$p \not\equiv 0$")

Alg. outputs "$p \not\equiv 0$" $\Rightarrow$ $p \not\equiv 0$

Alg. outputs "$p \equiv 0$" $\not\Rightarrow$ $p \equiv 0$

$\Pr[p \not\equiv 0 \mid \text{output "}p \equiv 0\text{"}] \leq \frac{1}{k}$

Randomization allows speeding up algorithms

Testing against the null polynomial

Assume we have oracle access $\mathcal{O}(\cdot, \ldots, \cdot)$ to a multivariate polynomial p in $\mathbb{R}[x_1, \ldots, x_n]$. Determine whether p is identically zero.

Solution based on randomization:

Exploit the fact if $p \not\equiv 0$ and $\bar{a} = (a_1, \ldots, a_n)$ is chosen at random, $\Pr[p(\bar{a}) = 0]$ is small.

Theorem (Schwartz-Zippel): let $S \subseteq \mathbb{R}$ with $|S| = k \cdot deg(p)$. If each component of $\bar{a} = (a_1, \ldots, a_n)$ is chosen independently and uniformly from S, then

$$\Pr[p(\bar{a}) = 0 \mid p \not\equiv 0] \leq 1/k \, .$$

Let $S \subseteq \mathbb{R}$ with $|S| = k \cdot deg(p)$
`For` $i = 1 \ldots m$ `do`
   Draw $a_1, \ldots, a_n$ independently and uniformly from S;
   `if` $\mathcal{O}(a_1, \ldots, a_n) \neq 0$ `then return` (“$p \not\equiv 0$”)
`return` (“$p \equiv 0$”)

Alg. outputs “$p \not\equiv 0$” $\Rightarrow$ $p \not\equiv 0$

Alg. outputs “$p \equiv 0$” $\not\Rightarrow$ $p \equiv 0$

$$\Pr[p \not\equiv 0 \mid \text{output “}p \equiv 0\text{”}] \leq \left(\frac{1}{k}\right)^m$$

Randomization allows speeding up Algorithms

Underlying techniques

Abundance of Witnesses

- Decision problem whose output depends on the presence (resp. absence) of a witness to prove (resp. disprove) a property.
- Witnesses abound in a given search space.
- Given a witness, the property is “efficiently” verified.

Amplification by Independent Trials

- Used in conjunction with the “abundance of witnesses” technique to reduce the error probability.
- Given an algorithm with error probability ε, run it n independent times to reduce the error probability to ε^n.

Randomization allows speeding up Algorithms

Underlying techniques

Abundance of Witnesses

Any $\bar{a}$ such that $p(\bar{a}) \neq 0$

- Decision problem whose output depends on the presence (resp. absence) of a witness to prove (resp. disprove) a property.

$p \neq 0$

- Witnesses abound in a given search space.

Any $S \subseteq \mathbb{R}$ with $|S| \gg deg(p)$

- Given a witness, the property is "efficiently" verified.

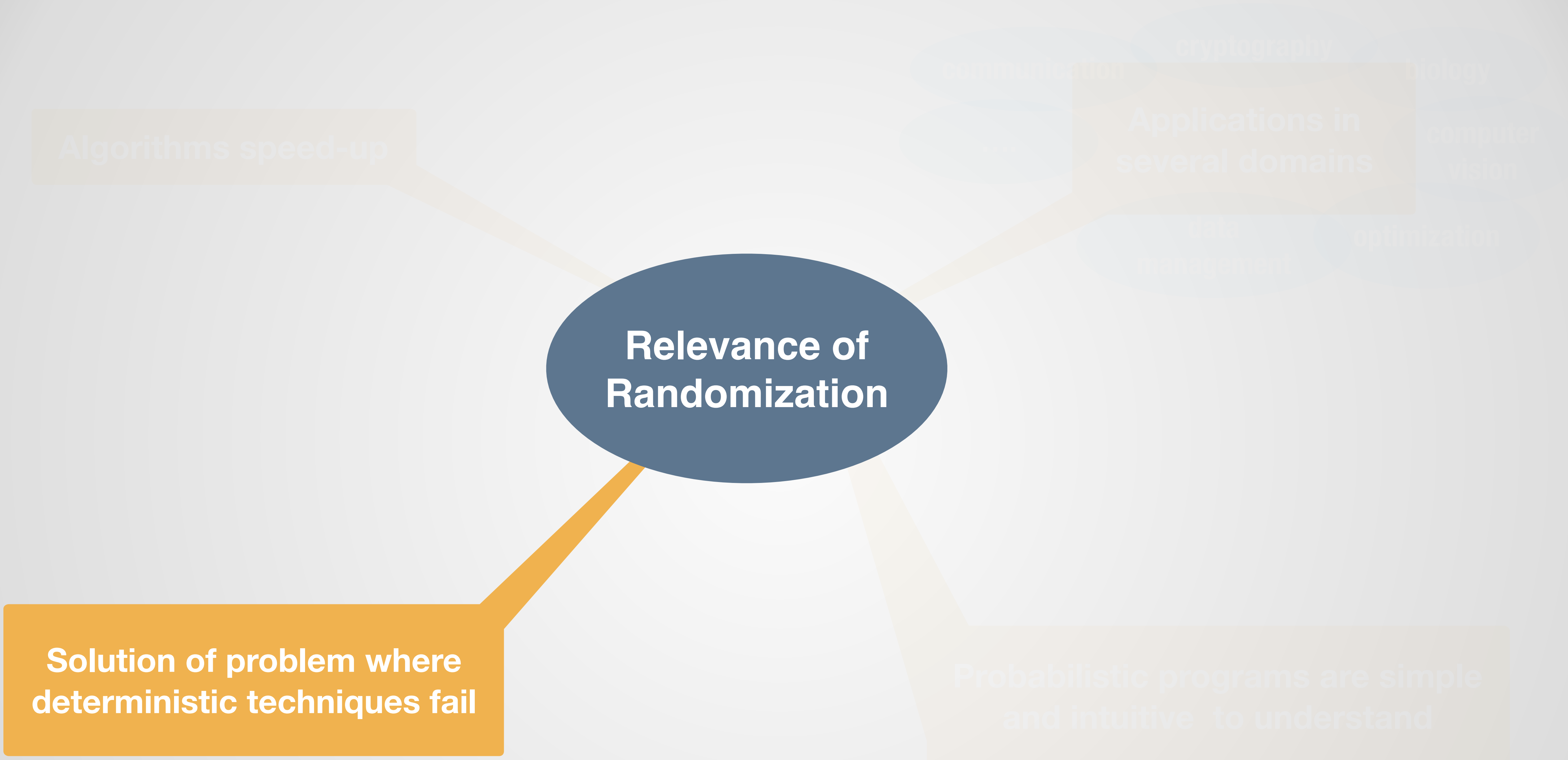
Schwartz-Zippel theorem

Other example: primality testing [Rabin '76].

Amplification by Independent Trials

- Used in conjunction with the "abundance of witnesses" technique to reduce the error probability.
- Given an algorithm with error probability ε, run it n **independent** times to reduce the error probability to ε^n.

Probabilistic Programs — Relevance



Randomization circumvents the Limitations of Determinism

The Dinning Philosopher Problem

Theorem (Lehmann & Rabin ’81) there exists no fully distributed and symmetric deterministic algorithm for the dining philosopher problem.

Randomized Algorithm

```
while (true) do
  (* Thinking Time *)
  trying := true
  while (trying) do
    s := rand{left, right}
    Wait until fork[s] is available and take it
    If fork[¬s] is available
          then take it and set trying to false
          else drop fork[s]
  (* Eating Time *)
  Drop both forks
```

Idea:

- Do not pick always the same fork first. Flip a coin to choose.
- If the second fork is not available, release the first and flip again the coin.

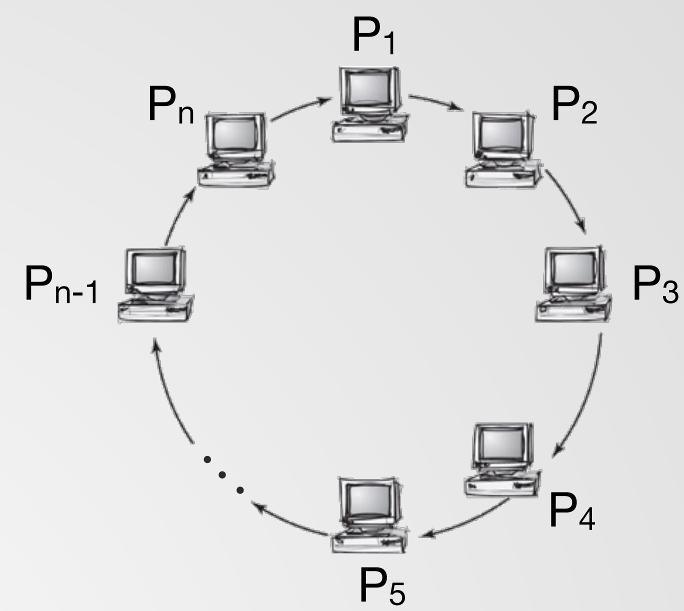
Algorithm is deadlock-free:

At any time, if there is a hungry philosopher, with probability one some philosopher will eventually eat. Algorithm can also be adapted to prevent starvation (ie the hungry philosopher will eventually eat).

Randomization circumvents the Limitations of Determinism

Leader Election

- Aim: to choose a leader node in a network.
- Network consists of *n* identical nodes $P_1,\ldots,P_n$ connected in a ring fashion.
- Transmission of messages is allowed between consecutive nodes in the ring.
- At the end of the process all nodes must agree on the election of the leader.



Theorem (Angluin '80) there exists **no deterministic algorithm** for carrying out the election in a ring of **identical** processes.

Randomized Algorithm

```
repeat
  s := empty list;
  name := rand{1, ..., K};
  For i = 1 ... n do
    s := s ++ [name];
    send(name) to next node;
    receive(name) from previous node
until (at least one name in s is unique)
return max{n ∈ s | n occus only once in s}
```

Idea:

- Each nodes chooses a random name from {1, …,K} and propagates it around the ring.
- At the end of the propagation each process has a list of the names of all the nodes.
- If there is a name that belongs to only one node, then this is the leader (in case of several, choose eg the largest)
- If there is no unique name, repeat the process.

Randomization circumvents the Limitations of Determinism

Underlying technique

Symmetry Breaking in Distributed Systems

- For many problems on distributed systems, deterministic solutions do not exists when objects are to be treaded identically.
- Using randomization to choose among identical objects may help solving the symmetry problem.

Probabilistic Programs — Tradeoffs

Advantages

- Reduction of time/space complexity
- Reduction of communication complexity in the distributed setting
- Allows tackling problems that have no deterministic solution
- Probabilistic programs are simple and easy to understand
- Wide range of application domains

Probabilistic Programs — Tradeoffs

Advantages

- Reduction of time/space complexity
- Reduction of communication complexity in the distributed setting
- Allows tackling problems that have no deterministic solution
- Probabilistic programs are simple and easy to understand
- Wide range of application domains

Disadvantages

- Absolute correctness is sometimes sacrificed: probabilistic programs are "correct with probability $1-\epsilon$ "
  - Quicksort, dining philosopher, leader election $\longrightarrow$ $\epsilon = 0$
  - Definite integral, testing against null polynomial $\longrightarrow$ $\epsilon > 0$

Probabilistic Programs — Tradeoffs

Advantages

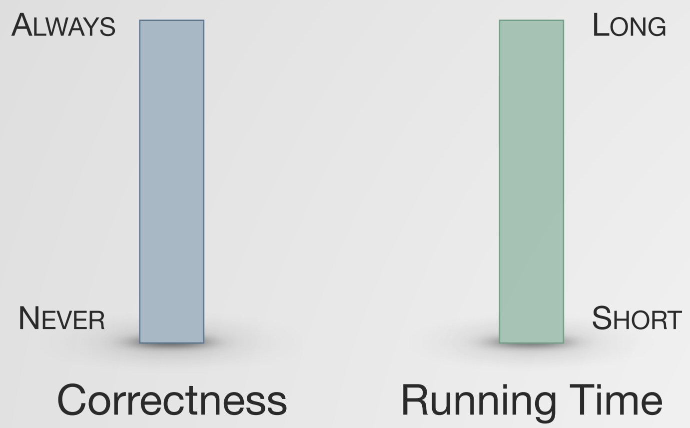
- Reduction of time/space complexity
- Reduction of communication complexity in the distributed setting
- Allows tackling problems that have no deterministic solution
- Probabilistic programs are simple and easy to understand
- Wide range of application domains

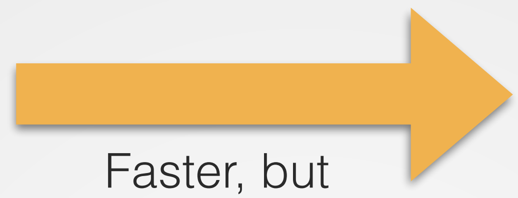
Disadvantages

- Absolute correctness is sometimes sacrificed: probabilistic programs are **"correct with probability $1-\epsilon$ "**
  - Quicksort, dining philosopher, leader election $\longrightarrow$ $\epsilon = 0$ ⟵ **LAS VEGAS ALGORITHM** — Running time is a random variable
  - Definite integral, testing against null polynomial $\longrightarrow$ $\epsilon > 0$ ⟵ **MONTE CARLO ALGORITHM** — Constant running time (always fast)

Probabilistic Programs — Reliability

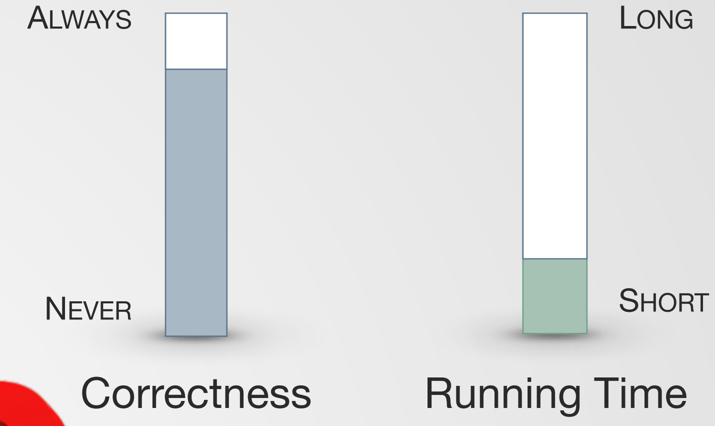
Deterministic Algorithm





Faster, but
“less”correct

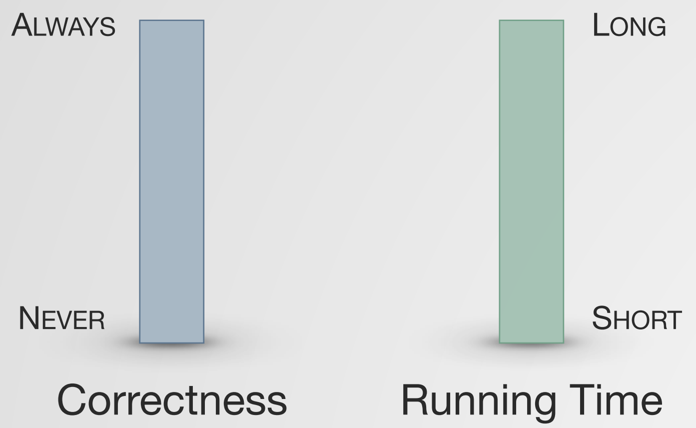
Probabilistic Algorithm



**Is the probabi-
listic algorithm
still reliable?**

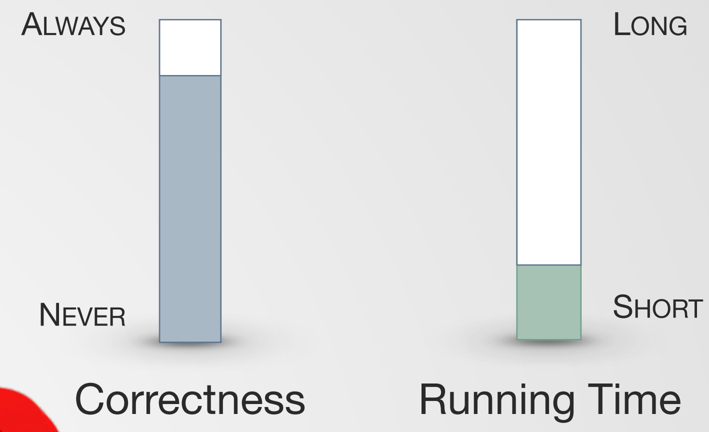
Probabilistic Programs — Reliability

Deterministic Algorithm



Faster, but
"less"correct

Probabilistic Algorithm



**Is the probabi-
listic algorithm
still reliable?**

Yes, absolutely!

Probabilistic Programs remain Reliable

Verifying Data Consistency

- Goal: R_I and R_{II} must communicate to verify whether $x{=}y$ $(x, y \in \{0,1\}^n)$.
- Requirement: minimize the # of bits exchanged.

$$\underset{R_I}{\boxed{x{=}x_1 \ldots x_n}} \xleftrightarrow{\text{communication}} \underset{R_{II}}{\boxed{y{=}y_1 \ldots y_n}}$$

Theorem: any deterministic protocol requires the exchange of (at least) n bits.

Randomized Algorithm

Idea: use random fingerprints of x and y.

Routine of $\mathrm{R_I} \triangleq$
$p := \mathsf{rand}\{i \in [2, n^2] \mid \mathit{prime}(i)\}$;
$s := x \bmod p$;
$\texttt{send}(p, s)$ $\texttt{to}$ $\mathrm{R_{II}}$;

Routine of $\mathrm{R_{II}} \triangleq$
$\texttt{receive}(p, s)$ $\texttt{from}$ $\mathrm{R_I}$;
$t := y \bmod p$;
$\texttt{if}$ $(s{=}t)$ $\texttt{then return}$ (“$x{=}y$”)
$\texttt{else return}$ (“$x{\neq}y$”)

$$\begin{array}{c}\#\text{bits}\\ \text{exchanged}\end{array} = \#\text{bits}(\underbrace{p}_{\leq n^2}) + \#\text{bits}(\underbrace{s}_{\leq p \leq n^2}) \leq 2\log_2(n^2)$$

Prot. outputs “$x{\neq}y$” $\Rightarrow$ $x{\neq}y$

$$\Pr[x \neq y \mid \text{output “}x{=}y\text{”}] \leq \frac{\ln(n^2)}{n}$$

Probabilistic Programs remain Reliable

Verifying Data Consistency

- Goal: R_I and R_{II} must communicate to verify whether $x=y$ $(x, y \in \{0,1\}^n)$.
- Requirement: minimize the # of bits exchanged.

$x=x_1 \ldots x_n$ $\longleftarrow$ communication $\longrightarrow$ $y=y_1 \ldots y_n$

R_I **R_{II}**

Theorem: any deterministic protocol requires the exchange of (at least) n bits.

Randomized Algorithm

Idea: use random fingerprints of x and y.

Routine of $R_I \triangleq$
  $p := \text{rand}\{i \in [2, n^2] \mid \mathit{prime}(i)\};$
  $s := x \bmod p;$
  $\texttt{send}(p, s)$ $\texttt{to}$ $R_{II};$

Routine of $R_{II} \triangleq$
  $\texttt{receive}(p, s)$ $\texttt{from}$ $R_I;$
  $t := y \bmod p;$
  $\texttt{if}$ $(s=t)$ $\texttt{then return}$ $(\text{“}x=y\text{”})$
  $\texttt{else return}$ $(\text{“}x\neq y\text{”})$

$$\#\text{bits exchanged} = \#\text{bits}(\underbrace{p}_{\leq n^2}) + \#\text{bits}(\underbrace{s}_{\leq p \leq n^2}) \leq 2\log_2(n^2)$$

Prot. outputs “$x \neq y$” $\Rightarrow$ $x \neq y$

$$\Pr[x \neq y \mid \text{output “}x=y\text{”}] \leq \frac{\ln(n^2)}{n}$$

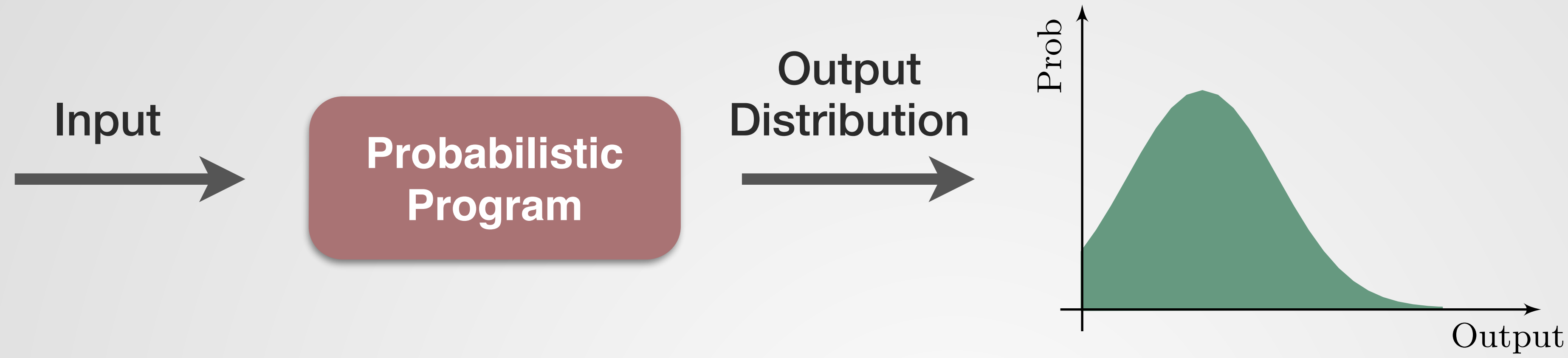
	# bits exch.	prob. error
$n=10^{10}$	133	4.60×10^{-09}
$n=10^{20}$	266	9.21×10^{-19}
$n=10^{30}$	398	1.38×10^{-28}
$n=10^{40}$	532	1.84×10^{-38}
$n=10^{50}$	664	2.30×10^{-48}

Agenda

- Introduction to probabilistic programs
- The problem of probabilistic program verification
- Seminar content
- Summary

Probabilistic Programs — The Verification Problem

Input/Output behaviour of probabilistic programs:



Verification of probabilistic programs:

- Property holds with a given probability

$$\Pr[error] \leq 0.02$$

- Assertion about the expected (ie average) value of a variable

$$\mathbb{E}[r] \leq 10$$

Examples of Probabilistic Assertions

- Cardinality of sets union

$m := |S_1| + \ldots + |S_n|$;
Draw a set S^* from $S_1, \ldots, S_n$ with probability $\Pr[S_i] = \frac{|S_i|}{m}$;
Draw an element x^* from S^* with uniform distribution;
$r := \frac{m}{cov(x)}$;
$\texttt{return}\ (r)$

$$\mathbb{E}[r] = |S_1 \cup S_2 \cup \ldots \cup S_n|$$

- Leader Election

$\texttt{repeat}$
$\quad s := \textit{empty list}$;
$\quad \textit{name} := \mathsf{rand}\{1, \ldots, K\}$;
$\quad \texttt{For}\ i = 1 \ldots n\ \texttt{do}$
$\quad\quad s := s \mathbin{++} [\textit{name}]$;
$\quad\quad \texttt{send}(\textit{name})\ \texttt{to}$ next node;
$\quad\quad \texttt{receive}(\textit{name})\ \texttt{from}$ previous node
$\texttt{until}$ (at least one name in s is unique)
$\texttt{return}\ \max\{n \in s \mid n$ occus only once in $s\}$

$$\Pr[\textit{termination}] = 1\ ?$$

Probabilistic Programs — The Verification Problem

Let p be probability that after the random choices of the node names, at least one name is unique. We know that $0 < p < 1$.

$$\begin{aligned}
\Pr[term] &= 1 - \Pr[non{-}term] \\
&= 1 - \Pr\left[\begin{array}{c}\text{in all rounds, there} \\ \text{is no unique name}\end{array}\right] \\
&= 1 - \lim_{n\to\infty} (1-p)^n \\
&= 1
\end{aligned}$$

Probabilistic Programs — The Verification Problem

Let p be probability that after the random choices of the node names, at least one name is unique. We know that $0 < p < 1$.

$$\begin{aligned}\Pr[term] &= 1 - \Pr[non\!-\!term] \\ &= 1 - \Pr\left[\begin{array}{c}\text{in all rounds, there}\\ \text{is no unique name}\end{array}\right] \\ &= 1 - \lim_{n\to\infty}(1-p)^n \\ &= 1\end{aligned}$$

Intricacy

Probabilistic programs may terminate with probability 1, and still admit diverging executions.

Insight: (set of) diverging executions have probability 0

Probabilistic Programs — The Verification Problem

Let p be probability that after the random choices of the node names, at least one name is unique. We know that $0 < p < 1$.

$$
\begin{aligned}
\Pr[\mathit{term}] &= 1 - \Pr[\mathit{non\!-\!term}] \\
&= 1 - \Pr\left[\begin{array}{c}\text{in all rounds, there} \\ \text{is no unique name}\end{array}\right] \\
&= 1 - \lim_{n \to \infty} (1 - p)^n \\
&= 1
\end{aligned}
$$

ALMOST-SURE TERMINATION (AST)

Intricacy

Probabilistic programs may terminate with probability 1, and still admit diverging executions.

Insight: (set of) diverging executions have probability 0

Probabilistic Programs — The Verification Problem

Let *p* be probability that after the random choices of the node names, at least one name is unique. We know that $0 < p < 1$.

$$\begin{aligned}
\Pr[\mathit{term}] &= 1 - \Pr[\mathit{non{-}term}] \\
&= 1 - \Pr\left[\begin{array}{c}\text{in all rounds, there}\\ \text{is no unique name}\end{array}\right] \\
&= 1 - \lim_{n \to \infty} (1 - p)^n \\
&= 1
\end{aligned}$$

ALMOST-SURE TERMINATION (AST)

Intricacy

Probabilistic programs may terminate with probability 1, and still admit diverging executions.

Insight: (set of) diverging executions have probability 0

Another example of AST:

$$\begin{array}{l}
\texttt{repeat} \\
\quad b := \mathit{flip_coin}(); \\
\texttt{until } (b = \texttt{heads})
\end{array}$$

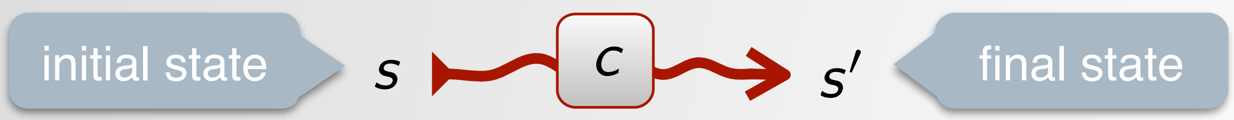
Probabilistic Programs — The Verification Problem

How to verify probabilistic assertions?

It is possible to extend standard verification techniques of sequential programs:

Hoare logic

- Assertions of the form {*P*} *c* {*Q*}, where *P* and *Q* are predicates over program states



$$\{P\}\ c\ \{Q\} \text{ is } \textit{valid} \text{ iff } P(s) \implies Q(s')$$

Example: $\{x \geq 0\}\ \ x := x{+}2\ \ \{x \geq 0\}$

- Deductive system (ie proof rules) to derive valid assertions (one rule per language construction)

$$\frac{}{\{P\}\texttt{skip}\{P\}}$$

$$\frac{}{\{P[x \leftarrow e]\}x := e\{P\}}$$

$$\frac{\{P\}s_1\{Q\} \qquad \{Q\}s_2\{R\}}{\{P\}s_1; s_2\{R\}}$$

- Proof objects are derivations (trees)

Weakest precondition calculus

- Given in terms of predicate transformer

$$\mathsf{wp}[c] \colon \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$$

$$\mathsf{wp}[c](Q) = \left\{ \begin{array}{l} \text{set of initial states that lead} \\ \text{to a final state satisfying } Q \end{array} \right\}$$

Example: $\mathsf{wp}[x := x{+}2](x \geq 0) = x \geq -2$

- Connection to Hoare logic

$$\{P\}\ c\ \{Q\} \qquad \text{iff} \qquad P \implies \mathsf{wp}[c](Q)$$

- Transformer$\mathsf{wp}[c]$ is defined by induction on the structure of *c:*

$$\mathsf{wp}[\texttt{skip}](Q) = Q$$
$$\mathsf{wp}[x := e](Q) = Q[x/E]$$
$$\mathsf{wp}[c_1; c_2](Q) = (\mathsf{wp}[c_1] \circ \mathsf{wp}[c_2])(Q)$$

Agenda

- Introduction to probabilistic programs
- The problem of probabilistic program verification
- Seminar content
- Summary

Seminar Content

Different extension of Hoare logic and weakest precondition calculus for probabilistic programs.

- Probabilistic predicate transformers [McIver & Morgan '96]

  Reward function $f : \Sigma \to \mathbb{R}$ over the set of final states.

  $\mathrm{wp}[c](f) =$ Expected reward of c wrt f

  expected value of *f* wrt distribution of final states

- Relational Hoare logics [Barthe '09,'12]

  Relates the executions of a program from two different initial states.

  Pre- and post-conditions are relations (rather than predicates) over program states.

  $\{=_L\} \;\; c \;\; \{=_L\}$

  c in non-interferent

- Hartog's Hoare logic [Hartog '02]

$\{\mathrm{true}\} \;\; c \;\; \{\forall i \bullet \; (i \leq 0) \vee \Pr[x = i] = (1/2)^i\}$

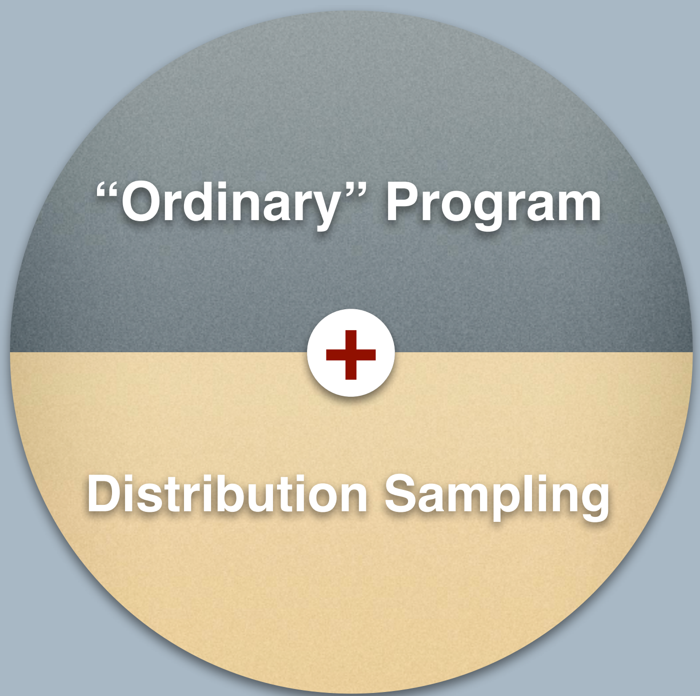
variable x is geometrically distributed

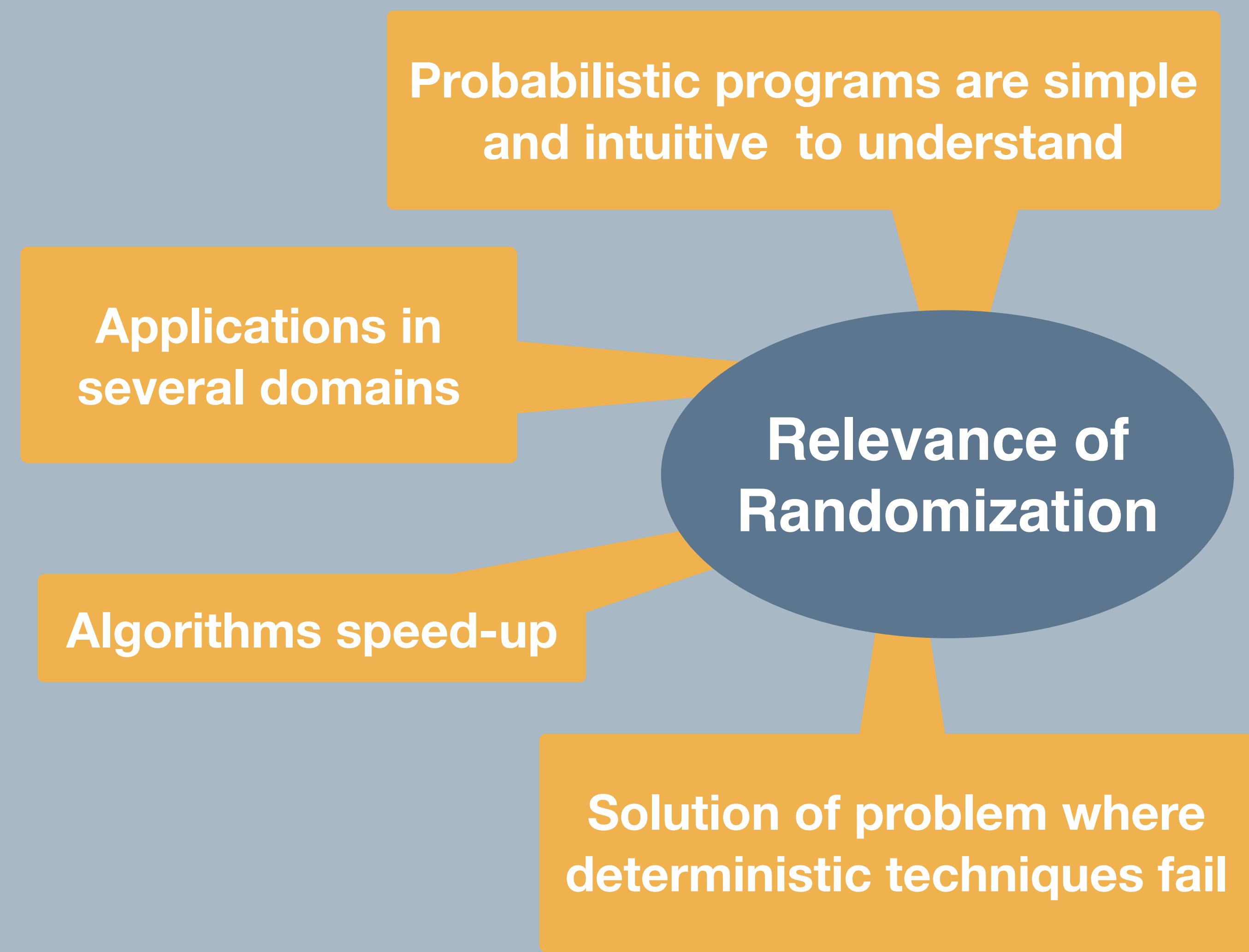
Agenda

- Introduction to probabilistic programs
- The problem of probabilistic program verification
- Seminar content
- Summary

Summary

What is a probabilistic program?





We can extend traditional program verification techniques to probabilistic programs.