On Randomization in (Higher-Order) Programming

Part I

Ugo Dal Lago

Escuela de Ciencias Informáticas, Buenos Aires, July 2023

Probabilistic Models

- The **environment** is supposed not to behave *deterministically*, but *probabilistically*.

Probabilistic Models

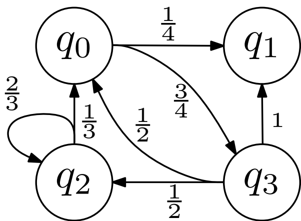
- The **environment** is supposed not to behave *deterministically*, but *probabilistically*.
- Crucial when modeling **uncertainty**.

Probabilistic Models

- The **environment** is supposed not to behave *deterministically*, but *probabilistically*.
- Crucial when modeling **uncertainty**.
- Useful to handle **complex** domains.

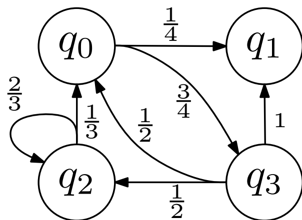
Probabilistic Models

- The **environment** is supposed not to behave *deterministically*, but *probabilistically*.
- Crucial when modeling **uncertainty**.
- Useful to handle **complex** domains.
- **Example:**



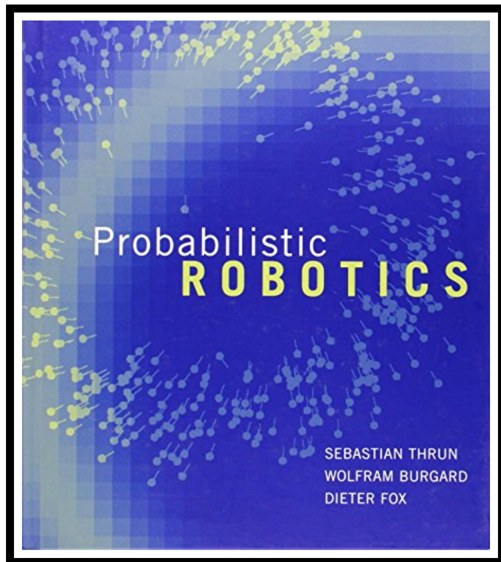
Probabilistic Models

- The **environment** is supposed not to behave *deterministically*, but *probabilistically*.
- Crucial when modeling **uncertainty**.
- Useful to handle **complex** domains.
- **Example:**



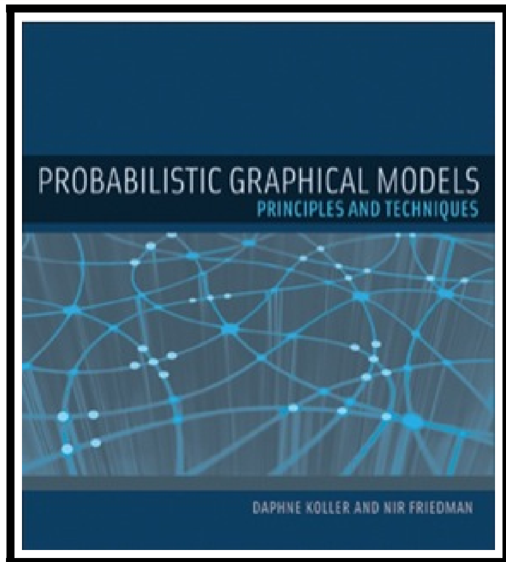
- **Abstractions:**
  - (Labelled) Markov Chains.

Probabilistic Models



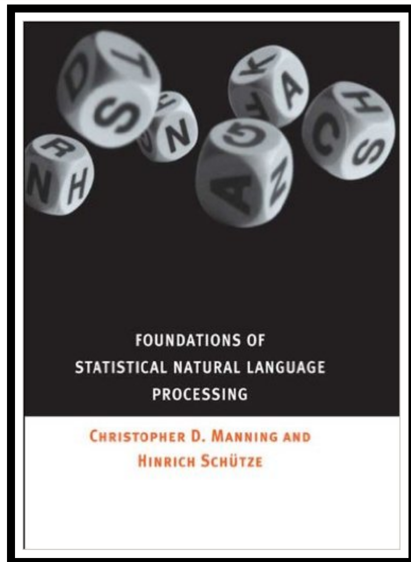
ROBOTICS

Probabilistic Models



ARTIFICIAL INTELLIGENCE

Probabilistic Models



NATURAL LANGUAGE
PROCESSING

Randomized Computation

- Algorithms and automata are assumed to have the ability to **sample** from a distribution [dLMSS1956,R1963].

Randomized Computation

- Algorithms and automata are assumed to have the ability to **sample** from a distribution [dLMSS1956,R1963].
- This is a **powerful tool** when solving computational problems.

Randomized Computation

- Algorithms and automata are assumed to have the ability to **sample** from a distribution [dLMSS1956,R1963].
- This is a **powerful tool** when solving computational problems.
- **Example:**

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.**

Randomized Computation

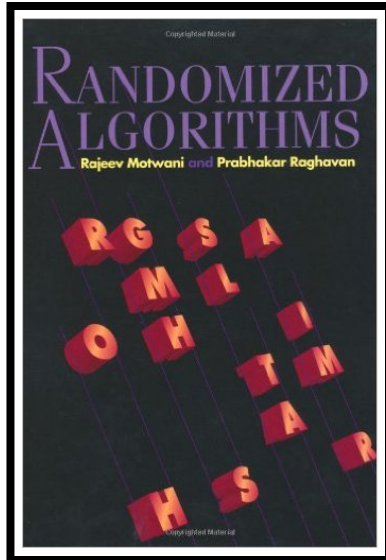
- Algorithms and automata are assumed to have the ability to **sample** from a distribution [dLMSS1956,R1963].
- This is a **powerful tool** when solving computational problems.
- **Example:**

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.**

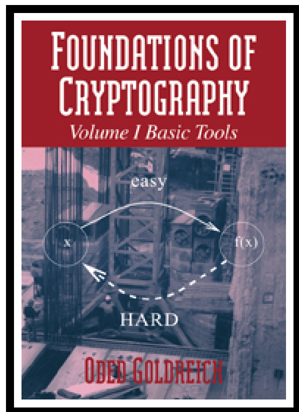
- **Abstractions:**
  - Randomized algorithms;
  - Probabilistic Turing machines.
  - Labelled Markov chains.

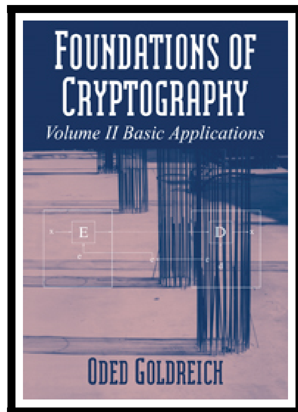
Randomized Computation



ALGORITHMICS

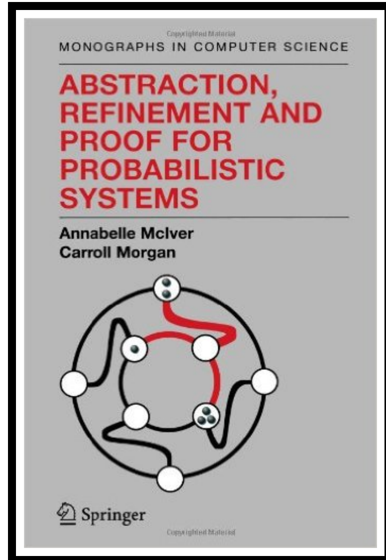
Randomized Computation





CRYPTOGRAPHY

Randomized Computation



PROGRAM VERIFICATION

Higher-Order Computation

- Useful in **programming**.

Higher-Order Computation

- Useful in **programming**.
- Functions are **first-class citizens**:
  - They can be passed as *arguments*;
  - They can be obtained as *results*.

Higher-Order Computation

- Useful in **programming**.
- Functions are **first-class citizens**:
  - They can be passed as *arguments*;
  - They can be obtained as *results*.
- **Motivations**:
  - *Modularity*;
  - *Code reuse*;
  - *Conciseness*.

Higher-Order Computation

- Useful in **programming**.
- Functions are **first-class citizens**:
  - They can be passed as *arguments*;
  - They can be obtained as *results*.
- **Motivations**:
  - *Modularity*;
  - *Code reuse*;
  - *Conciseness*.
- **Example**:

```
foldr              :: (a -> b -> b) -> b -> [a] -> b
foldr f acc []      = acc
foldr f acc (x:xs) = f x (foldr f acc xs)
```

Higher-Order Computation

- Useful in **programming**.
- Functions are **first-class citizens**:
  - They can be passed as *arguments*;
  - They can be obtained as *results*.
- **Motivations**:
  - *Modularity*;
  - *Code reuse*;
  - *Conciseness*.
- **Example**:

```
foldr               :: (a -> b -> b) -> b -> [a] -> b
foldr f acc []       = acc
foldr f acc (x:xs)   = f x (foldr f acc xs)
```

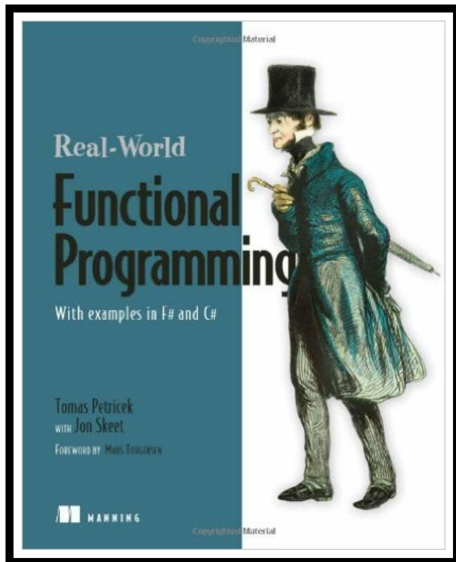
Higher-Order Computation

- Useful in **programming**.
- Functions are **first-class citizens**:
  - They can be passed as *arguments*;
  - They can be obtained as *results*.
- **Motivations**:
  - *Modularity*;
  - *Code reuse*;
  - *Conciseness*.
- **Example**:

```
foldr               :: (a -> b -> b) -> b -> [a] -> b
foldr f acc []       = acc
foldr f acc (x:xs)   = f x (foldr f acc xs)
```

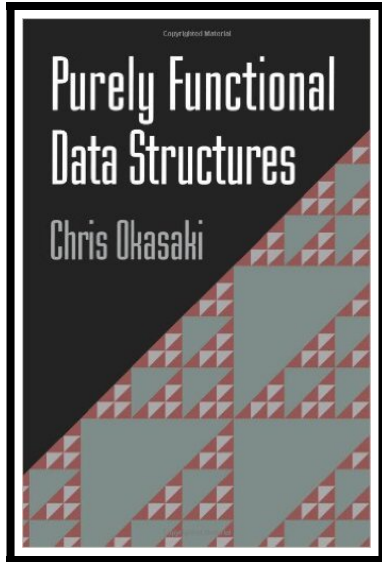
- **Models**:
  - λ-calculus

Higher-Order Computation



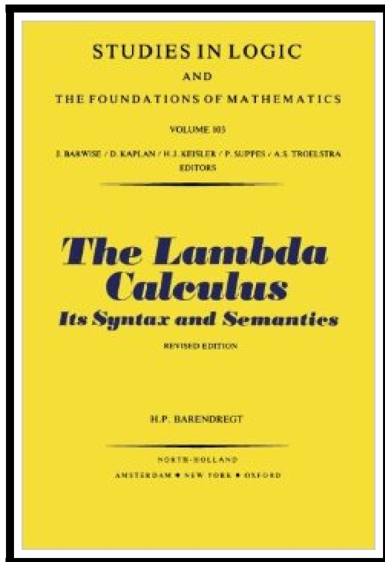
FUNCTIONAL PROGRAMMING

Higher-Order Computation



FUNCTIONAL
DATA STRUCTURES

Higher-Order Computation



λ-CALCULUS

Does it Make Sense?

Does it Make Sense?

What Kind of Metatheory Does it Have?

Does it Make Sense?

What Kind of Metatheory Does it Have?

Interesting Research Problems?

This Course

- Five lectures:
  1. **Randomized Computation, Complexity, and Programming**.
  2. **Calculi for Randomized Higher-Order Computation**.
  3. **Relational Reasoning**.
  4. **Termination and Complexity Analysis**.
  5. **Beyond Randomization: Bayesian and Quantum Programming**

This Course

- Five lectures:
  1. **Randomized Computation, Complexity, and Programming**.
  2. **Calculi for Randomized Higher-Order Computation**.
  3. **Relational Reasoning**.
  4. **Termination and Complexity Analysis**.
  5. **Beyond Randomization: Bayesian and Quantum Programming**
- The course material will be made available on the course's webpage:

`http://www.cs.unibo.it/~dallago/ECI2023/`

The Lectures' Structure

- Each lecture will be divided into *three parts*:
  1. An **overview**, based on some slides.
     - Around *90 minutes* long.
     - Slides are progressively made available on the course webpage.
  2. An **in-depth analysis** of some of the presented concepts, using an online whiteboard.
     - Around *60 minutes* long.
     - The transcript will be made available, again at the course webpage.
  3. **A short exercise session** about the topics of the days.
     - Around *30 minutes*.
     - I will ask you to solve some simple exercises about the topics of the day. Please try to have with you pen and paper, and a laptop if possible.
- There will be **a break** between the first and second part of the lecutre, around *10 minutes*.
- I would be glad to answer **your questions**, either during the lecture, or at the end of it.

The Exam

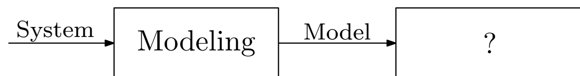
- It will consist of an **handout exam**, which will be made public after the last lecture.
- You have to work **individually**.
- You are required to complete as many exercises you can in at most ten days, namely by **August 8th**.
- Your solutions to the exercises have to be typeset, preferably in LaTeX, and **sent to me** via email, the subject beginning with `[ECI2023]`.

Part I

The Basics of Randomized Computation

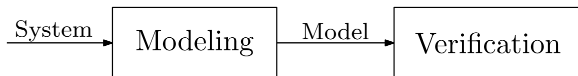
Incepting Probability into Algorithms

- Probability theory provides effective tools to facilitate **modeling** of existing systems, e.g.



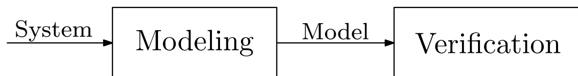
Incepting Probability into Algorithms

- Probability theory provides effective tools to facilitate **modeling** of existing systems, e.g.



Incepting Probability into Algorithms

- Probability theory provides effective tools to facilitate **modeling** of existing systems, e.g.



- Probability, however, can also be seen as a very powerful way to **compute**.
  - Algorithms can "flip a coin";
  - This is somehow a paradigm shift: probabilism is not there for modeling uncertainty, but rather for the purpose of **relaxing determinism** as an algorithm design principle.
  - This "new" computation paradigm is called **randomized computation**.
  - Your algorithms (then your programs) can be *seen* and *treated* as probabilistic systems.

But Why?

- **Achieving Better Performances**
  - Allowing computation to proceed probabilistically *somehow* allows to exploit different computation paths **at the same time**.
  - Of course, each computation path comes with its own probability.
  - ... but all this can (sometime) be turned into a way to alleviate exponential blowups.

But Why?

- **Achieving Better Performances**
  - Allowing computation to proceed probabilistically *somehow* allows to exploit different computation paths **at the same time**.
  - Of course, each computation path comes with its own probability.
  - ...but all this can (sometime) be turned into a way to alleviate exponential blowups.
- **Allowing a Form of "Confusion" in Computation**
  - Deterministic computation has its own shortcomings.
  - In particular, in some interactive scenarios probabilistic computation is a way to make computation unpredictable.

Example: Primality Testing

```
MillerRabin(n)

If n > 2 and n is even, return composite.
/* Factor n − 1 as 2^s t where t is odd. */
s ← 0
t ← n − 1
while t is even
    s ← s + 1
    t ← t/2
end /* Done. n − 1 = 2^s t. */
Choose x ∈ {1, 2, ..., n − 1} uniformly at random.
Compute each of the numbers x^t, x^{2t}, x^{4t}, ..., x^{2^s t} = x^{n−1} mod n.
If x^{n−1} ≢ 1 (mod n), return composite.
for i = 1, 2, ..., s
    If x^{2^i t} ≡ 1 (mod n) and x^{2^{i−1} t} ≢ ±1 (mod n), return composite.
end /* Done checking for fake square roots. */
Return probably prime.
```

Example: Cryptography

- Suppose given a public-key encryption scheme $\Pi = (GEN, ENC, DEC)$ and an adversary $\mathcal{A}$.

Example: Cryptography

- Suppose given a public-key encryption scheme $\Pi = (GEN, ENC, DEC)$ and an adversary $\mathcal{A}$.
- In principle, ENC can be any algorithm, and in particular it can be a purely deterministic algorithm.

Example: Cryptography

- Suppose given a public-key encryption scheme $\Pi = (GEN, ENC, DEC)$ and an adversary $\mathcal{A}$.
- In principle, ENC can be any algorithm, and in particular it can be a purely deterministic algorithm.
- If ENC is **not randomized**, however, there is simply *no hope* that it can be secure.
  - We will see a proof of it later today.

Example: Cryptography

- Suppose given a public-key encryption scheme $\Pi = (GEN, ENC, DEC)$ and an adversary $\mathcal{A}$.
- In principle, ENC can be any algorithm, and in particular it can be a purely deterministic algorithm.
- If ENC is **not randomized**, however, there is simply *no hope* that it can be secure.
  - We will see a proof of it later today.
- Similarly when Π is **private-key** and we want to prove it to be secure against *chosen-plaintext* attacks.
  - Saying it differently, the so called **one-time-pad** is simply not secure against active attacks.

What do we Assume?

- Familiarity with **Turing machines and automata**.
  - We briefly review the rudiments of computability and complexity theory.
- The basics of **elementary probability theory**.
- Some knowledge about the **theory of programming languages**.
  - We will review almost all what is needed to understand randomized programming languages.

Probabilistic Turing Machines

- Generalizing ordinary Turing Machines to the probabilistic setting is a relatively easy task.
- A **Turing Machine** is a quadruple (Q, Σ, δ, F).
  - In presence of **determinism**,

$$\delta : Q \times \Sigma \to Q \times \Sigma \times \{\leftarrow, \downarrow, \to\}.$$

Probabilistic Turing Machines

- Generalizing ordinary Turing Machines to the probabilistic setting is a relatively easy task.
- A **Turing Machine** is a quadruple (Q, Σ, δ, F).
  - In presence of **determinism**,
$$\delta : Q \times \Sigma \to Q \times \Sigma \times \{\leftarrow, \downarrow, \rightarrow\}.$$
  - When evolution is **nondeterministic**,
$$\delta : Q \times \Sigma \to \mathbf{P}(Q \times \Sigma \times \{\leftarrow, \downarrow, \rightarrow\}).$$

Probabilistic Turing Machines

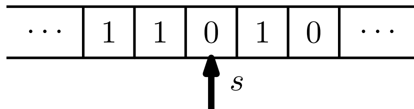
- Generalizing ordinary Turing Machines to the probabilistic setting is a relatively easy task.
- A **Turing Machine** is a quadruple (Q, Σ, δ, F).
  - In presence of **determinism**,
    $$\delta : Q \times \Sigma \to Q \times \Sigma \times \{\leftarrow, \downarrow, \rightarrow\}.$$
  - When evolution is **nondeterministic**,
    $$\delta : Q \times \Sigma \to \mathbf{P}(Q \times \Sigma \times \{\leftarrow, \downarrow, \rightarrow\}).$$
  - When evolution is **probabilistic**,
    $$\delta : Q \times \Sigma \to \mathbf{D}(Q \times \Sigma \times \{\leftarrow, \downarrow, \rightarrow\}).$$

    where $\mathbf{D}(A)$ is the set of all distributions on A. It is often assumed that the distribution returned by δ has a support equal to 2 and that each outcome has probability $\frac{1}{2}$.

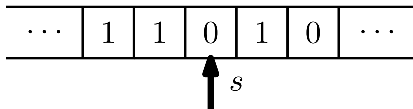
Probabilistic Turing Machines

- Generalizing ordinary Turing Machines to the probabilistic setting is a relatively easy task.
- A **Turing Machine** is a quadruple (Q, Σ, δ, F).
  - In presence of **determinism**,
  $$\delta : Q \times \Sigma \to Q \times \Sigma \times \{\leftarrow, \downarrow, \rightarrow\}.$$
  - When evolution is **nondeterministic**,
  $$\delta : Q \times \Sigma \to \mathbf{P}(Q \times \Sigma \times \{\leftarrow, \downarrow, \rightarrow\}).$$
  - When evolution is **probabilistic**,
  $$\delta : Q \times \Sigma \to \mathbf{D}(Q \times \Sigma \times \{\leftarrow, \downarrow, \rightarrow\}).$$
  where $\mathbf{D}(A)$ is the set of all distributions on A. It is often assumed that the distribution returned by δ has a support equal to 2 and that each outcome has probability $\frac{1}{2}$.
- Every time it makes a step, a PTM *flips a coin*, without reusing the results of previous coin flips.

Probabilistic Turing Machines

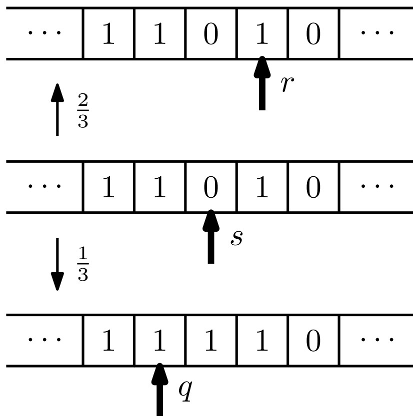


Probabilistic Turing Machines



$$\delta(s,0) = \{(q,1,\leftarrow)^{\frac{1}{3}}, (r,0,\rightarrow)^{\frac{2}{3}}\}$$

Probabilistic Turing Machines



$$\delta(s,0) = \{(q,1,\leftarrow)^{\frac{1}{3}}, (r,0,\rightarrow)^{\frac{2}{3}}\}$$

The Probability Theory of PTMs

- Suppose to work with binary fair probabilistic choices.

The Probability Theory of PTMs

- Suppose to work with binary fair probabilistic choices.
- At each step the PTM flips a coin, proceeding based on the outcome of the flip.

The Probability Theory of PTMs

- Suppose to work with binary fair probabilistic choices.
- At each step the PTM flips a coin, proceeding based on the outcome of the flip.
- The **probabilistic space** we work with has the form

$$(\mathbb{F}^{\mathbb{N}}, \mathcal{F}, \mu)$$

  where $\mathbb{F} = \{0, 1\}$, $\mathcal{F}$ is any σ-algebra on $\mathbb{F}^{\mathbb{N}}$ including all the cylinders, and μ assigns probability $\frac{1}{2}$ to each cylinder $C_{b,i}$.
  - $C_{b,i}$ is the subset of $\mathbb{F}^{\mathbb{N}}$ consisting of all functions f such that $f(i) = b$.

The Probability Theory of PTMs

- Suppose to work with binary fair probabilistic choices.
- At each step the PTM flips a coin, proceeding based on the outcome of the flip.
- The **probabilistic space** we work with has the form

$$(\mathbb{F}^{\mathbb{N}}, \mathcal{F}, \mu)$$

  where $\mathbb{F} = \{0, 1\}$, $\mathcal{F}$ is any σ-algebra on $\mathbb{F}^{\mathbb{N}}$ including all the cylinders, and μ assigns probability $\frac{1}{2}$ to each cylinder $C_{b,i}$.
  - $C_{b,i}$ is the subset of $\mathbb{F}^{\mathbb{N}}$ consisting of all functions f such that $f(i) = b$.
- Any property of interest (e.g. the outcome of the computation process or the number of computation steps) can be seen as a **random variable**.
  - Very often, we do not care of mentioning the random variable itself, and concentrate on its distribution.

PTMs, Languages and Functions

- In PTM, **termination** and **acceptance of a string** become probabilistic events.
- A PTM can terminate with probability 1 even if there is the *possibility* of nontermination.
- A given string $s \in \Sigma^*$ is accepted *with a probability* between 0 and 1, included.
- How can we define language recognition?
  - One can consider different, alternative, constraints on the **error** probability:
    - It must be 0.
    - It must be smaller than $\frac{1}{2}$;
    - It must be smaller than a given fixed constant ε, itself smaller than $\frac{1}{2}$.
  - One can stipulate that the given result is the one with **highest probability**.

What Algorithms Compute

- **Deterministic Computation**
  - For every input x, there is *at most* one output y any algorithm $\mathcal{A}$ produces when fed with x.
  - As a consequence:

$$\mathcal{A} \qquad \rightsquigarrow \qquad [\![\mathcal{A}]\!] : \mathbb{F}^* \rightharpoonup \mathbb{F}^*.$$

What Algorithms Compute

- **Deterministic Computation**
  - For every input x, there is *at most* one output y any algorithm $\mathcal{A}$ produces when fed with x.
  - As a consequence:

$$\mathcal{A} \qquad \rightsquigarrow \qquad [\![\mathcal{A}]\!] : \mathbb{F}^* \rightharpoonup \mathbb{F}^*.$$

- **Randomized Computation**
  - For every input x, any algorithm $\mathcal{A}$ outputs y with a probability $0 \leq p \leq 1$.
  - As a consequence:

$$\mathcal{A} \qquad \rightsquigarrow \qquad [\![\mathcal{A}]\!] : \mathbb{F}^* \rightarrow \mathbf{D}(\mathbb{F}^*).$$

  - The distribution $[\![\mathcal{A}]\!](n)$ sums to anything between 0 and 1, thus accounting for the probability of divergence.

Quick Recap of Computability Theory

- How could we formalize that a function $f : \mathbb{F}^* \to \mathbb{F}^*$ can be computed **mechanically**?

Quick Recap of Computability Theory

- How could we formalize that a function $f : \mathbb{F}^* \to \mathbb{F}^*$ can be computed **mechanically**?
  - The answer is simple: is there a TM computing f?
  - A TM **computes** f iff whenever $s \in \mathbb{F}^*$ the machine always halts producing $f(s) \in \mathbb{F}^*$ in output.
  - $\mathfrak{FR}$ is the corresponding class of functions.

Quick Recap of Computability Theory

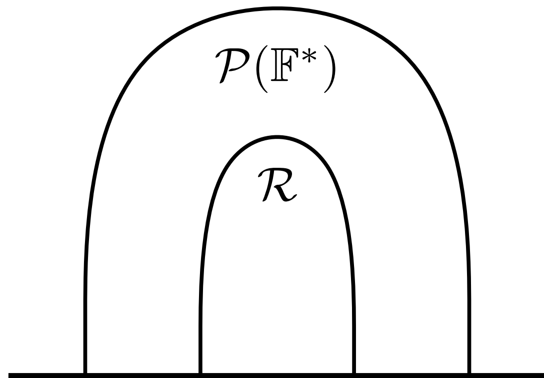
- How could we formalize that a function $f : \mathbb{F}^* \rightarrow \mathbb{F}^*$ can be computed **mechanically**?
  - The answer is simple: is there a TM computing f?
  - A TM **computes** f iff whenever $s \in \mathbb{F}^*$ the machine always halts producing $f(s) \in \mathbb{F}^*$ in output.
  - $\mathfrak{FR}$ is the corresponding class of functions.
- What could we say about **computational problems**, i.e., about any $L \subseteq \mathbb{F}^*$?

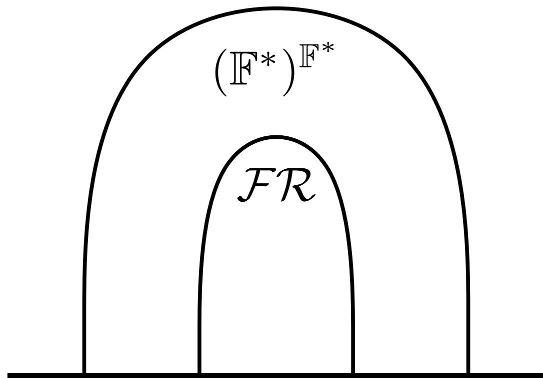
Quick Recap of Computability Theory

- How could we formalize that a function $f : \mathbb{F}^* \to \mathbb{F}^*$ can be computed **mechanically**?
  - The answer is simple: is there a TM computing f?
  - A TM **computes** f iff whenever $s \in \mathbb{F}^*$ the machine always halts producing $f(s) \in \mathbb{F}^*$ in output.
  - $\mathfrak{FR}$ is the corresponding class of functions.
- What could we say about **computational problems**, i.e., about any $L \subseteq \mathbb{F}^*$?
  - L is said to be **recursive** iff the function f_L defined as follows is computable:

$$f_L(x) = \begin{cases} 0 & \text{if } x \in L \\ 1 & \text{otherwise} \end{cases}$$

  - $\mathfrak{R}$ is the corresponding class of functions.


$\mathcal{P}(\mathbb{F}^*)$
$\mathcal{R}$


$(\mathbb{F}^*)^{\mathbb{F}^*}$
$\mathcal{FR}$

Computing with Limited Resources

- Some computable functions seem to require too much **time** (and/or **space**) to be computed.
- Let us consider the function $A : \mathbb{N}^2 \to \mathbb{N}$ defined as follows:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0; \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0; \\ A(m-1, A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

- The value $A(m,n)$ increases **too fast** when m and n grow, e.g.,

$$A(4,3) \approx 10^{6031 \cdot 10^{19727}}.$$

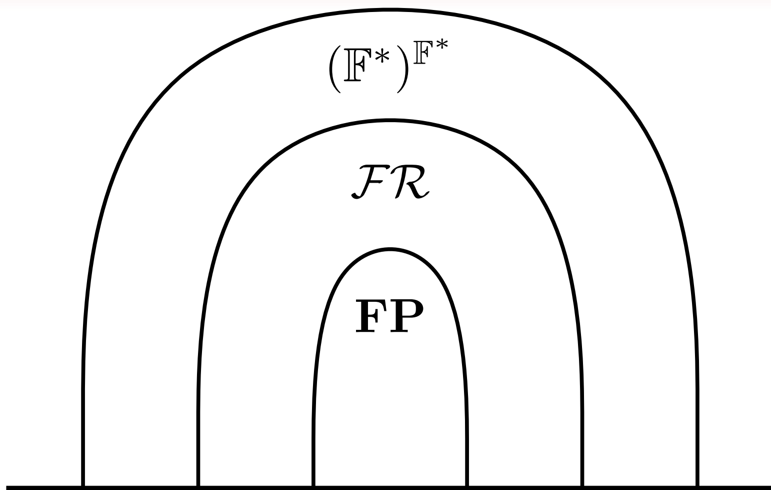
- Ideally, we would like the time necessary to compute a function on $s \in \mathbb{F}^*$ to be bounded by a **fixed** polynomial on $|s|$.

Polynomial Time

- A TM M **works** in time $g : \mathbb{N} \to \mathbb{N}$ iff for every $s \in \mathbb{F}^*$, the computation of M on input s takes at most $g(s)$ steps. In this case, we will write that $M \in \mathit{Time}(g)$.
- The complexity class **FP** is the collection of all functions from $\mathbb{F}^*$ in $\mathbb{F}^*$ which are computable by a TM M in

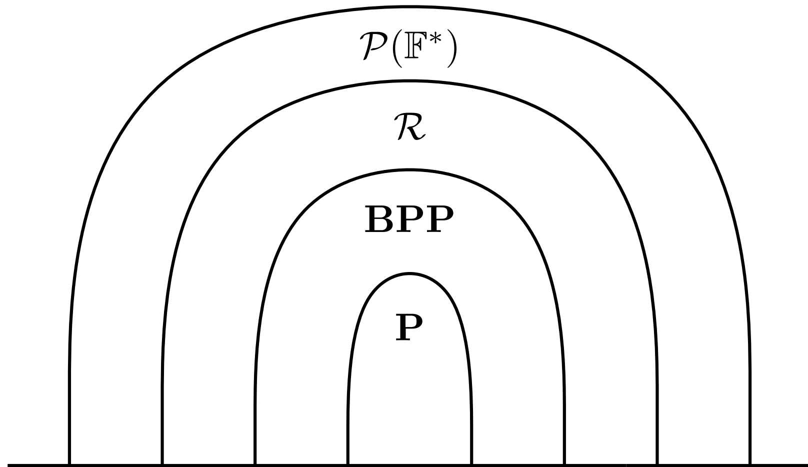
$$\bigcup_{g \in \mathsf{POLY}} \mathit{Time}(g).$$

  where POLY is the space of all polynomials from $\mathbb{N}$ to $\mathbb{N}$.
- Knowing that a function is **not** in **FP** can sometime help a lot. . . .

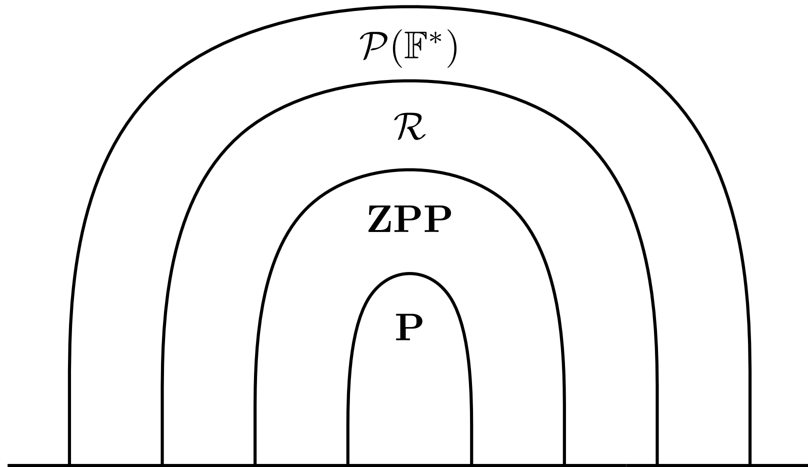

$(\mathbb{F}^*)^{\mathbb{F}^*}$
$\mathcal{FR}$
$\mathbf{FP}$

Probabilistic Polynomial Time

- How should we capture the concept of feasibility in a probabilistic setting?
- There is a tradeoff between bounding running times and bounding the error...
  - We can insist on having **no** error, but allow computation to take polynomial time only on the average, obtaining the class **ZPP**.
  - We can allow the error to be smaller than a constant $\varepsilon < \frac{1}{2}$, but insist on polynomial time bounds to hold independently on the random choices, getting the class **BPP**.
- There are many other possibilities, but these are somehow less interesting, giving rise to classes which are too large.



$\mathcal{P}(\mathbb{F}^*)$
$\mathcal{R}$
BPP
P


$\mathcal{P}(\mathbb{F}^*)$
$\mathcal{R}$
ZPP
P

$$\mathrm{BPP} \stackrel{?}{=} \mathrm{P}$$

Part II

Why Higher-Order (Randomized) Programming?

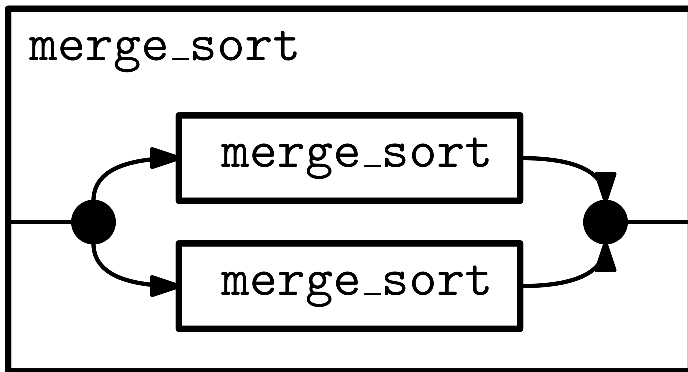
MergeSort (1)

```
let rec merge = function
      | list, []
      | [], list -> list
      | h1::t1, h2::t2 ->
          if h1 <= h2 then
            h1 :: merge (t1, h2::t2)
          else
            h2 :: merge (h1::t1, t2);;

let rec halve = function
      | []
      | [_] as t1 -> t1, []
      | h::t ->
          let t1, t2 = halve t in
            h::t2, t1;;
```

MergeSort (2)

```
let rec merge_sort = function
      | []
      | [_] as list -> list
      | list ->
          let l1, l2 = halve list in
            merge (merge_sort l1, merge_sort l2);;
```

The Structure of MergeSort



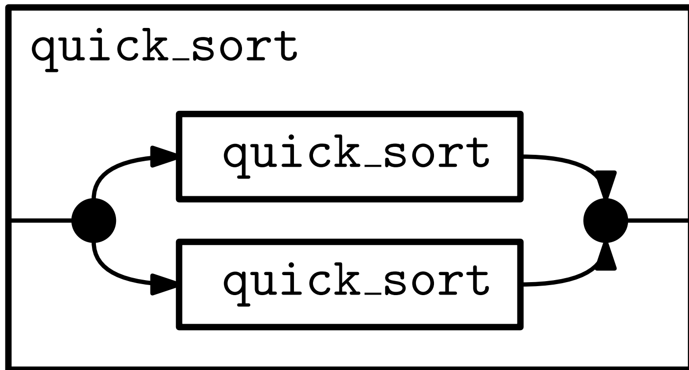
MergeSort, HO

```
let rec merge = function
      | (list, []), _ -> list
      | ([], list), _ -> list
      | (h1::t1, h2::t2),el ->
        if h1 <= h2 then
          h1 :: merge ((t1, h2::t2),el)
        else
          h2 :: merge ((h1::t1, t2),el);;

let rec halve = function
      | []
      | [_] as t1 -> (t1, []),()
      | h::t ->
        let (t1, t2),el = halve t in
          (h::t2, t1),el;;

let rec dac divide conquer = function
      | []
      | [_] as list -> list
      | list ->
        let (l1, l2),el = divide list in
          conquer ((dac divide conquer l1, dac divide conquer l2),el);;

let rec merge_sort = dac halve merge;;
```

QuickSort

```
let app x y z = x @ (z::y);;

let partition = function
    | pivot :: rest -> (List.partition (( > ) pivot) (rest)),pivot;;

let rec quick_sort = function
    | []
    | [_] as list -> list
    | list ->
      let (l1, l2), el = partition list in
        app (quick_sort l1) (quick_sort l2) el;;
```

The Structure of MergeSort



QuickSort, HO

```
let app = function
    | (x,y),z -> x @ (z::y);;

let partition = function
    | pivot :: rest -> (List.partition (( > ) pivot) (rest)),pivot;;

let rec dac divide conquer = function
    | []
    | [_] as list -> list
    | list ->
        let (l1, l2),el = divide list in
          conquer ((dac divide conquer l1, dac divide conquer l2),el);;

let quick_sort = dac partition app;;
```

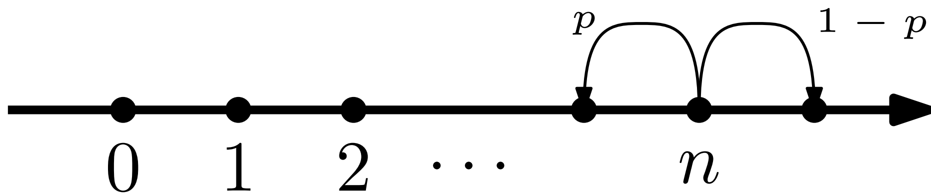
Randomized QuickSort (1)

```
let app = function
    | (x,y),z -> x @ (z::y);;

let rec extract = function
  | [],_ -> ([],0)
  | hd::tl,n ->
     if n==0 then
       (tl,hd)
     else
       let (l,el) = extract(tl,n-1) in
         (hd::l,el);;

let partition list =
  let (rest,pivot) = extract (list,(Random.int (List.length list))) in
    (List.partition (( > ) pivot) (rest)),pivot;;
```

Randomized QuickSort (2)

```
let rec dac divide conquer = function
      | []
      | [_] as list -> list
      | list ->
          let (l1, l2),el = divide list in
            conquer ((dac divide conquer l1, dac divide conquer l2),el);;

let rand_quick_sort = dac partition app;;
```

Random Walk



Two Kinds of Random Walks

```
let rec iter f g n = if n==0 then g else let m=pred(n) in f m (iter f g m);;

let mult m n = succ(m)*n;;

let fact = iter mult 1;;

let rec param_iter f g step n =
    if n==0 then g else let m=step(n) in f m (param_iter f g step m);;

let succ_2 m n = n+1;;

let updown_fair x = x+(2*Random.int(2)-1);;

let fair_random_walk = param_iter succ_2 0 updown_fair;;

let updown_biased x = if Random.int(3)==0 then x+1 else x-1;;

let biased_random_walk = param_iter succ_2 0 updown_biased;;
```

The Coupon Collector

- A brand distributes a large quantity of coupons, each labelled with $i \in \{1, \dots, n\}$.

The Coupon Collector

- A brand distributes a large quantity of coupons, each labelled with $i \in \{1, \ldots, n\}$.
- Every day, you collect one coupon at a local store. Any label i has probability $\frac{1}{n}$ to occur.

The Coupon Collector

- A brand distributes a large quantity of coupons, each labelled with $i \in \{1, \ldots, n\}$.
- Every day, you collect one coupon at a local store. Any label i has probability $\frac{1}{n}$ to occur.
- You win a prize when you collect a set of n coupons, each with a distinct label.

The Coupon Collector

- A brand distributes a large quantity of coupons, each labelled with $i \in \{1, \ldots, n\}$.
- Every day, you collect one coupon at a local store. Any label i has probability $\frac{1}{n}$ to occur.
- You win a prize when you collect a set of n coupons, each with a distinct label.
- **Example**: if $n = 5$, you could get the following coupons:

$$3, 1, 5, 2, 3, 1, 5, 2, 3, 1, 5, 2, \ldots$$

The Coupon Collector

- A brand distributes a large quantity of coupons, each labelled with $i \in \{1, \ldots, n\}$.
- Every day, you collect one coupon at a local store. Any label i has probability $\frac{1}{n}$ to occur.
- You win a prize when you collect a set of n coupons, each with a distinct label.
- **Example**: if $n = 5$, you could get the following coupons:

$$3, 1, 5, 2, 3, 1, 5, 2, 3, 1, 5, 2, \ldots$$

- Are you guaranteed to win the prize with probability 1? After how many days, on the average?

The Coupon Collector

```
let rec base_param_iter f g step base e =
  if base(e) then g else let d=step(e) in f d (base_param_iter f g step base d);;

let second_zero = function
  | (_,0) -> true
  | _ -> false;;

let succ_2 m n = n+1;;

let step_2 = function
  | (n,m) -> if Random.int(n)<=m then (n,m-1) else (n,m);;

let coupon_collector x = base_param_iter succ_2 0 step_2 second_zero (x,x);;
```

Part III

Randomized Programming in Other Paradigms

Imperative Programming

- Programming randomized algorithms is possible in most modern imperative programming languages, e.g., `Python` or `Rust`.

Imperative Programming

- Programming randomized algorithms is possible in most modern imperative programming languages, e.g., `Python` or `Rust`.
- A `Python` program implementing the **fair random walk** would, e.g., look like the following:

```
import random

def fair_random_walk(x):
        if x==0:
            return True
        else:
            b=random.choice([-1,1])
            return fair_random_walk(x+b)
```

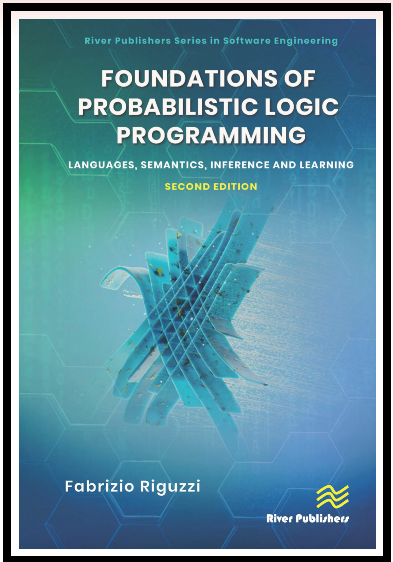
- Many of the verification techniques which are well-known in the deterministic setting (e.g. Floyd-Hoare logics, symbolic execution, or abstract interpretation) can be generalized to the presence of probabilistic choice.
  - This is highly nontrivial, we will see something about that, if time permits.

Logic Programming

- In logic programming, computation is seen as deduction: programs are sets of **clauses** and inputs to them are **facts**, e.g.

$$edge(a,b) \quad edge(b,c) \quad edge(a,c) \quad edge(b,d)$$
$$reachable(x,x) \leftarrow$$
$$reachable(x,y) \leftarrow edge(x,z) \wedge reachable(z,y)$$

- Computation consists in trying deducing the truth of a **query**, e.g. $reachable(a,d)$, from the facts and clauses.
- Various ways of generalizing logic programming to accommodate for randomization have been introduced, e.g.:
  - Facts can be hold **with a certain probability**, but clauses remain deterministic.
  - Whenever the variables in a clause are **substituted**, randomization enters into the playground.



LOGIC PROGRAMMING