Probabilistic Tools in Algorithms

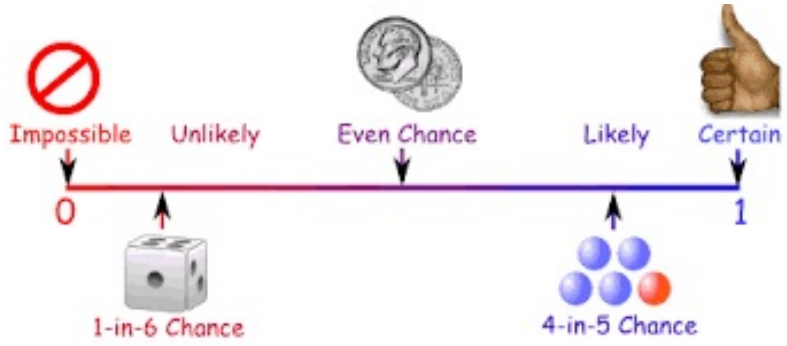
Josep Díaz Maria J. Serna Conrado Martínez
U. Politècnica de Catalunya

RA-MIRI 2022–2023

What is probability?

Probability: useful technique to simulate and explain real world.

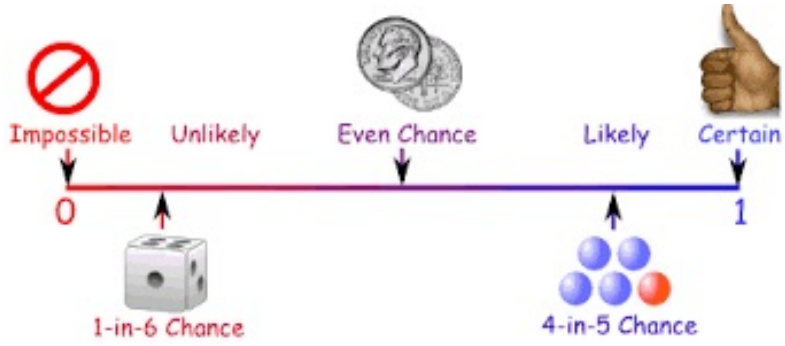
Any English speaking person understands the words likely and unlikely.



But in everyday life, do we consciously think in terms of probability?

What is probability?

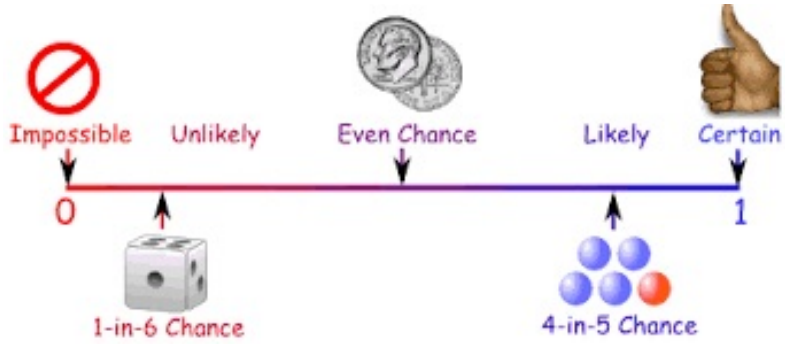
Probability: useful technique to simulate and explain real world.
Any English speaking person understands the words likely and unlikely.



But in everyday life, do we consciously think in terms of probability?

What is probability?

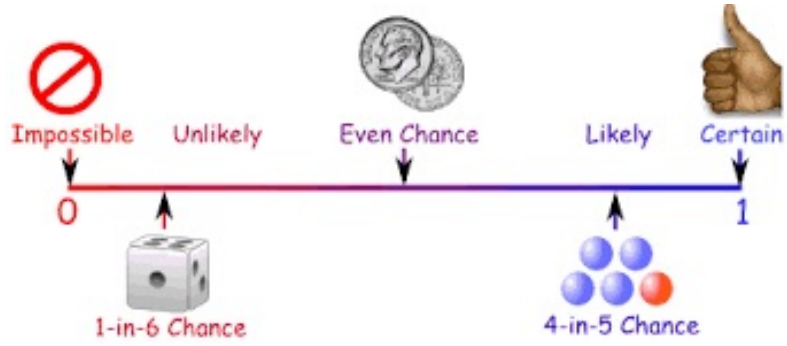
Probability: useful technique to simulate and explain real world.
Any English speaking person understands the words likely and unlikely.



But in everyday life, do we consciously think in terms of probability?

What is probability?

Probability: useful technique to simulate and explain real world.
Any English speaking person understands the words likely and unlikely.



But in everyday life, do we consciously think in terms of probability?

What is probability?

As far as we know, many phenomena in *nature* seem to be generated by random choices, but it is difficult to simulate truly unpredictable random experiments:

Flipping a coin or tossing a dice are *deterministic* experiments; Given the initial angle of the coin, the spin, humidity, etc. we can predict the outcome of flipping a coin.

In the same way, in today's computers, the *random generation* functions are deterministic and simulate randomness $\Longrightarrow$ pseudorandom number generators.

What is probability?

As far as we know, many phenomena in *nature* seem to be generated by random choices, but it is difficult to simulate truly unpredictable random experiments:

Flipping a coin or tossing a dice are *deterministic* experiments; Given the initial angle of the coin, the spin, humidity, etc. we can predict the outcome of flipping a coin.

In the same way, in today's computers, the *random generation* functions are deterministic and simulate randomness $\Longrightarrow$ pseudorandom number generators.

What is probability?

As far as we know, many phenomena in *nature* seem to be generated by random choices, but it is difficult to simulate truly unpredictable random experiments:

Flipping a coin or tossing a dice are *deterministic* experiments; Given the initial angle of the coin, the spin, humidity, etc. we can predict the outcome of flipping a coin.

In the same way, in today's computers, the *random generation* functions are deterministic and simulate randomness $\Longrightarrow$ pseudorandom number generators.

Probability and computers

The most basic method is the linear congruential generator: from a seed integer $x_0 \in \mathbb{N}$, produce a sequence of pseudo-random values

$$x_{n+1} = (a\,x_0 + b) \bmod m,$$

for a, b constants and m a large integer.

In C/C++ `rand( )`, m is a 32-bit integer, $a = 22695477$, $b = 1$

A computer deterministically generates pseudorandom numbers.

How would you generate a vector with a sequence of pseudorandom bits?

```
for (int i = 0; i < n; ++i)
    bit[i] = rand() % 2;
```

Some applications of probability in CS

- Algorithm design: Making algorithms run faster by introducing probability choices, against “bad” inputs.
- Data structures: when implementing most of the used data structures, e.g. dictionaries, the use of probability helps to speed up search and reduce space.
- Learning theory: in learning theory one assumes the data is generated according to specific probability distributions.
- Studying and design mechanisms for large complex networks: The design of algorithms for Internet, WWW, Facebook, etc, is based in the design realistic probabilistic models for those huge networks.

Some applications of probability in CS

- Algorithm design: Making algorithms run faster by introducing probability choices, against “bad” inputs.
- Data structures: when implementing most of the used data structures, e.g. dictionaries, the use of probability helps to speed up search and reduce space.
- Learning theory: in learning theory one assumes the data is generated according to specific probability distributions.
- Studying and design mechanisms for large complex networks: The design of algorithms for Internet, WWW, Facebook, etc, is based in the design realistic probabilistic models for those huge networks.

Some applications of probability in CS

- Algorithm design: Making algorithms run faster by introducing probability choices, against “bad” inputs.
- Data structures: when implementing most of the used data structures, e.g. dictionaries, the use of probability helps to speed up search and reduce space.
- Learning theory: in learning theory one assumes the data is generated according to specific probability distributions.
- Studying and design mechanisms for large complex networks: The design of algorithms for Internet, WWW, Facebook, etc, is based in the design realistic probabilistic models for those huge networks.

Some applications of probability in CS

- Algorithm design: Making algorithms run faster by introducing probability choices, against “bad” inputs.
- Data structures: when implementing most of the used data structures, e.g. dictionaries, the use of probability helps to speed up search and reduce space.
- Learning theory: in learning theory one assumes the data is generated according to specific probability distributions.
- Studying and design mechanisms for large complex networks: The design of algorithms for Internet, WWW, Facebook, etc, is based in the design realistic probabilistic models for those huge networks.

Some applications of probability in CS

- Data science: To design efficient algorithm for huge data set, usually we do keep a relevant sample, rather than keep all the data.
- Cryptography: Randomness and number theory, are essential for cryptography and crypto-hashing.
- Data compression: improving data compression algorithms passes through analysing and modelling the underlying probability distribution of the data, and evaluating its information-theoretic contents.
- Percolation & Diffusion: Probabilistic ad-hoc graph models and techniques have played an important role in modellizing diffusion process such as epidemics or the spreading of rumors and news, helping to stop or mitigated massive infections, including e-infections or to boost the spreading of information.

Some applications of probability in CS

- Data science: To design efficient algorithm for huge data set, usually we do keep a relevant sample, rather than keep all the data.
- Cryptography: Randomness and number theory, are essential for cryptography and crypto-hashing.
- Data compression: improving data compression algorithms passes through analysing and modelling the underlying probability distribution of the data, and evaluating its information-theoretic contents.
- Percolation & Diffusion: Probabilistic ad-hoc graph models and techniques have played an important role in modellizing diffusion process such as epidemics or the spreading of rumors and news, helping to stop or mitigated massive infections, including e-infections or to boost the spreading of information.

Some applications of probability in CS

- Data science: To design efficient algorithm for huge data set, usually we do keep a relevant sample, rather than keep all the data.
- Cryptography: Randomness and number theory, are essential for cryptography and crypto-hashing.
- Data compression: improving data compression algorithms passes through analysing and modelling the underlying probability distribution of the data, and evaluating its information-theoretic contents.
- Percolation & Diffusion: Probabilistic ad-hoc graph models and techniques have played an important role in modellizing diffusion process such as epidemics or the spreading of rumors and news, helping to stop or mitigated massive infections, including e-infections or to boost the spreading of information.

Some applications of probability in CS

- Data science: To design efficient algorithm for huge data set, usually we do keep a relevant sample, rather than keep all the data.
- Cryptography: Randomness and number theory, are essential for cryptography and crypto-hashing.
- Data compression: improving data compression algorithms passes through analysing and modelling the underlying probability distribution of the data, and evaluating its information-theoretic contents.
- Percolation & Diffusion: Probabilistic ad-hoc graph models and techniques have played an important role in modellizing diffusion process such as epidemics or the spreading of rumors and news, helping to stop or mitigated massive infections, including e-infections or to boost the spreading of information.

Randomization and algorithmis: Probabilistic analysis

Given a deterministic algorithm, it happens that a few "instances" may bias the complexity outcome of the algorithm, which for most of the instances seem to work well, for example, Quicksort.

In this case, we can perform a probabilistic analysis of the deterministic algorithm as follows:

Fix a probability distribution on the set of inputs, parametrized by input size. Often the distribution is the uniform, but not always.

We see the number of steps as a random variable $T(n)$ and compute its expected value $\mu = \mathbb{E}[T(n)]$.

We also need to prove concentration, that is, with high probability, $T(n)$ is "close" to μ.

Randomization and algorithmis: Probabilistic analysis

Given a deterministic algorithm, it happens that a few "instances" may bias the complexity outcome of the algorithm, which for most of the instances seem to work well, for example, Quicksort.

In this case, we can perform a probabilistic analysis of the deterministic algorithm as follows:

Fix a probability distribution on the set of inputs, parametrized by input size. Often the distribution is the uniform, but not always.

We see the number of steps as a random variable $T(n)$ and compute its expected value $\mu = \mathbb{E}[T(n)]$.

We also need to prove concentration, that is, with high probability, $T(n)$ is "close" to μ.

Randomization and algorithmis: Probabilistic analysis

Given a deterministic algorithm, it happens that a few "instances" may bias the complexity outcome of the algorithm, which for most of the instances seem to work well, for example, Quicksort.

In this case, we can perform a probabilistic analysis of the deterministic algorithm as follows:

Fix a probability distribution on the set of inputs, parametrized by input size. Often the distribution is the uniform, but not always.

We see the number of steps as a random variable $T(n)$ and compute its expected value $\mu = \mathbb{E}[T(n)]$.

We also need to prove concentration, that is, with high probability, $T(n)$ is "close" to μ.

Randomization and algorithmis: Probabilistic analysis

Given a deterministic algorithm, it happens that a few "instances" may bias the complexity outcome of the algorithm, which for most of the instances seem to work well, for example, Quicksort.

In this case, we can perform a probabilistic analysis of the deterministic algorithm as follows:

Fix a probability distribution on the set of inputs, parametrized by input size. Often the distribution is the uniform, but not always.

We see the number of steps as a random variable $T(n)$ and compute its expected value $\mu = \mathbb{E}[T(n)]$.

We also need to prove concentration, that is, with high probability, $T(n)$ is "close" to μ.

Randomization and algorithmis: Probabilistic analysis

Given a deterministic algorithm, it happens that a few "instances" may bias the complexity outcome of the algorithm, which for most of the instances seem to work well, for example, Quicksort.

In this case, we can perform a probabilistic analysis of the deterministic algorithm as follows:

Fix a probability distribution on the set of inputs, parametrized by input size. Often the distribution is the uniform, but not always.

We see the number of steps as a random variable $T(n)$ and compute its expected value $\mu = \mathbb{E}[T(n)]$.

We also need to prove concentration, that is, with high probability, $T(n)$ is "close" to μ.

Randomization and algorithmis: Randomized algorithms

We can design a randomized algorithms, where the algorithm takes random choices and continues the computation according to the output of the random choices.

In this case, we must perform a probabilistic analysis of the complexity, the worst-case complexity is pointless!

Randomization and algorithmis: Randomized algorithms

We can design a randomized algorithms, where the algorithm takes random choices and continues the computation according to the output of the random choices.
In this case, we must perform a probabilistic analysis of the complexity, the worst-case complexity is pointless!

Randomization and algorithmis: Randomized algorithms

There are two main types of probabilistic algorithms:

- Monte Carlo: Always halt in finite time, but may output the wrong answer. If the answer is binary (yes/not) the error can be in one direction, *one-side error*, or the error could be in both answers *two-side error*. In Monte Carlo algorithms it is important to bound the error probability.
- Las Vegas: The output is always correct but the running time may be unbounded.

It is easy to convert a Las Vegas algorithm into a Monte Carlo, how?. The contrary is not always true.

In this course we will be working mainly with Monte Carlo algorithms.

Randomization and algorithmis: Randomized algorithms

There are two main types of probabilistic algorithms:

- Monte Carlo: Always halt in finite time, but may output the wrong answer. If the answer is binary (yes/not) the error can be in one direction, *one-side error*, or the error could be in both answers *two-side error*. In Monte Carlo algorithms it is important to bound the error probability.
- Las Vegas: The output is always correct but the running time may be unbounded.

It is easy to convert a Las Vegas algorithm into a Monte Carlo, how?. The contrary is not always true.

In this course we will be working mainly with Monte Carlo algorithms.

Randomization and algorithmis: Randomized algorithms

There are two main types of probabilistic algorithms:

- Monte Carlo: Always halt in finite time, but may output the wrong answer. If the answer is binary (yes/not) the error can be in one direction, *one-side error*, or the error could be in both answers *two-side error*. In Monte Carlo algorithms it is important to bound the error probability.
- Las Vegas: The output is always correct but the running time may be unbounded.

It is easy to convert a Las Vegas algorithm into a Monte Carlo, how?. The contrary is not always true.

In this course we will be working mainly with Monte Carlo algorithms.

Randomization and algorithmis: Randomized algorithms

There are two main types of probabilistic algorithms:

- Monte Carlo: Always halt in finite time, but may output the wrong answer. If the answer is binary (yes/not) the error can be in one direction, *one-side error*, or the error could be in both answers *two-side error*. In Monte Carlo algorithms it is important to bound the error probability.
- Las Vegas: The output is always correct but the running time may be unbounded.

It is easy to convert a Las Vegas algorithm into a Monte Carlo, how?. The contrary is not always true.

In this course we will be working mainly with Monte Carlo algorithms.

Randomization and algorithmis: Randomized algorithms

There are two main types of probabilistic algorithms:

- Monte Carlo: Always halt in finite time, but may output the wrong answer. If the answer is binary (yes/not) the error can be in one direction, *one-side error*, or the error could be in both answers *two-side error*. In Monte Carlo algorithms it is important to bound the error probability.
- Las Vegas: The output is always correct but the running time may be unbounded.

It is easy to convert a Las Vegas algorithm into a Monte Carlo, how?. The contrary is not always true.

In this course we will be working mainly with Monte Carlo algorithms.

A randomized sorting algorithm

What do you know about QuickSort?

- General deterministic sorting algorithm
- Runs in time $\mathcal{O}(n^2)$
- Average time is $\Theta(n \log n)$ when the input follows the uniform distribution.

We want to keep the input deterministic and devise a randomized algorithm that sorts in expected $\Theta(n \log n)$ time.

A randomized sorting algorithm

What do you know about QuickSort?

- General deterministic sorting algorithm
- Runs in time $\mathcal{O}(n^2)$
- Average time is $\Theta(n \log n)$ when the input follows the uniform distribution.

We want to keep the input deterministic and devise a randomized algorithm that sorts in expected $\Theta(n \log n)$ time.

A randomized sorting algorithm

What do you know about QuickSort?

- General deterministic sorting algorithm
- Runs in time $\mathcal{O}(n^2)$
- Average time is $\Theta(n \log n)$ when the input follows the uniform distribution.

We want to keep the input deterministic and devise a randomized algorithm that sorts in expected $\Theta(n \log n)$ time.

A randomized sorting algorithm

```
procedure RAND-QUICKSORT(A[0..n − 1])
    Compute a uniform random permutation of [n]          =
{1, . . . , n} in P

    ▷ Rearrange A according to P
    ▷ Average cost is Θ(n log n) but uses no extra space
    for i := 0 to n − 1 do
        j := P[i]
        while j < i do
            j := P[j]
        end while
        A[i] :=: A[j]
    end for

    QUICKSORT(A)
end procedure
```

The algorithm reaches our goal, if we can compute a random permutation within the right (expected) time.

Generating a permutation uniformly at random

A permutation Π over $[n]$ defines a re-ordering of the elements, formally a bijective function $\pi : [n] \to n$.

The number of different permutations is $n!$.

Considering the experiment of generating a uniformly random permutation, we get the probability space $\Omega = \{\pi_1, \pi_2, \ldots, \pi_{n!}\}$, that is, $|\Omega| = n!$.

Generating a permutation uniformly at random (u.a.r.) means, for each n, generate a particular permutation π with probability

$$\frac{1}{|\Omega|} = \frac{1}{n!}.$$

Randomized algorithm to generate u.a.r. a permutation

Fisher-Yates Algorithm (also known as Knuth's algorithm)

```
procedure RANDOM-PERM(n)
    for i := 0 to n − 1 do
        π[i] := i
    end for
    for i := n − 1 downto 1 do
        j := RAND(i + 1)
        π[j] :=: π[i]
    end for
end procedure
```

`Rand(i)` returns a random integer in $[0, i-1]$.

Fisher-Yates algorithm

- The algorithm considers the items in the array one at a time from the end and swaps each element with an element in the array from that point to the beginning. This has cost $\mathcal{O}(n)$
- Notice that each element has an equal probability, of $1/n$, of being chosen as the last element in the array (including the element that starts out in that position).
- Applying this analysis recursively, we see that the output permutation has probability

$$\frac{1}{n}\frac{1}{n-1}\cdots\frac{1}{2} = \frac{1}{n!}$$

- That is, each permutation is equally likely.

Lemma Random-Perm (n) produces a u.a.r. permutation of $[n]$ in $\Theta(n)$ steps.

Fisher-Yates algorithm

- The algorithm considers the items in the array one at a time from the end and swaps each element with an element in the array from that point to the beginning. This has cost $\mathcal{O}(n)$
- Notice that each element has an equal probability, of $1/n$, of being chosen as the last element in the array (including the element that starts out in that position).
- Applying this analysis recursively, we see that the output permutation has probability

$$\frac{1}{n}\frac{1}{n-1}\cdots\frac{1}{2} = \frac{1}{n!}$$

- That is, each permutation is equally likely.

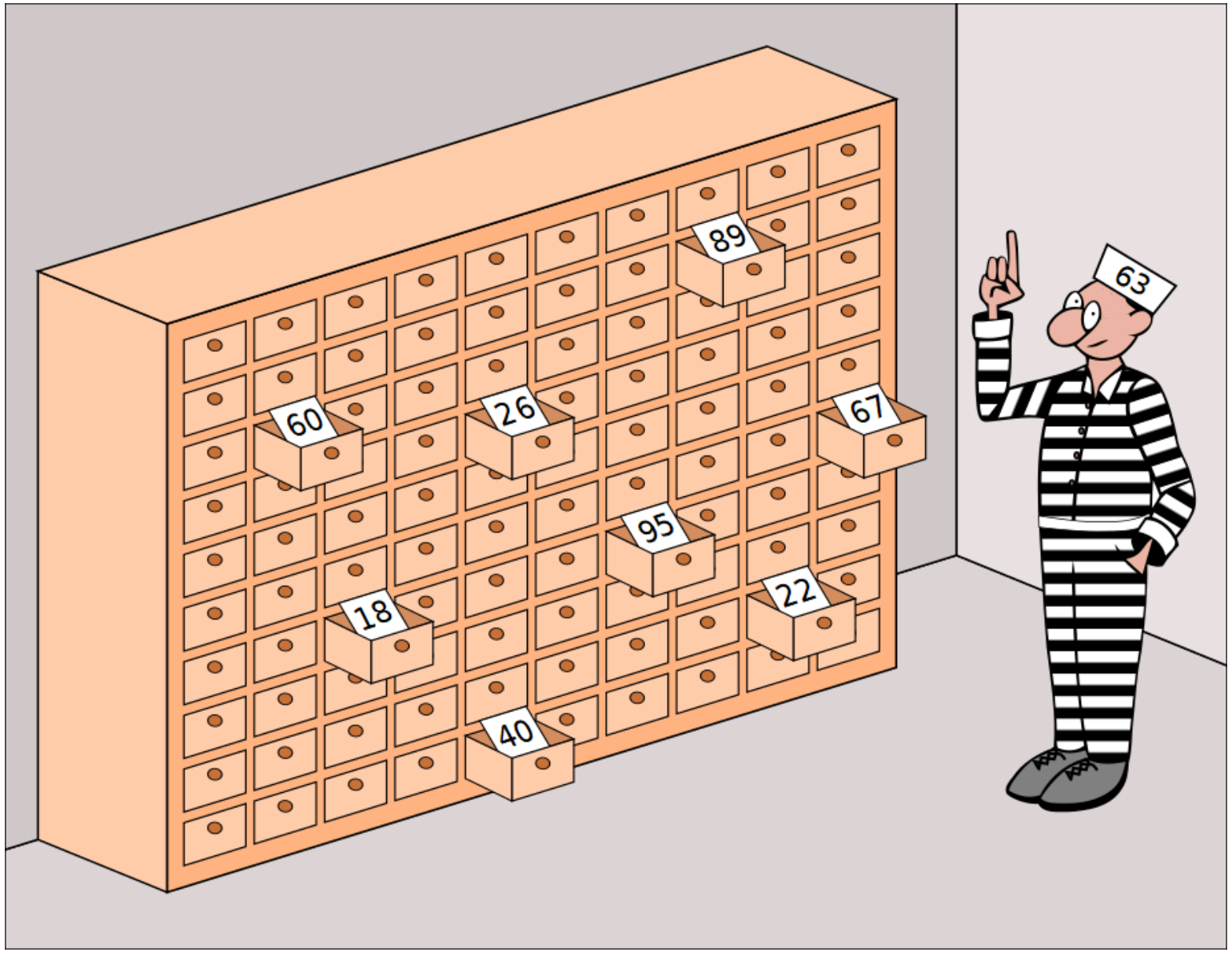
Lemma Random-Perm (n) produces a u.a.r. permutation of $[n]$ in $\Theta(n)$ steps.

The 100 prisoners problem

Here's a popular formulation of this problem, first proposed by P.B. Miltersen in 2003:

> *The director of a prison offers 100 death row prisoners, who are numbered from 1 to 100, a last chance. A room contains a cupboard with 100 drawers. The director randomly puts one prisoner's number in each closed drawer. The prisoners enter the room, one after another. Each prisoner may open and look into 50 drawers in any order. The drawers are closed again afterwards. If, during this search, every prisoner finds his number in one of the drawers, all prisoners are pardoned. If just one prisoner does not find his number, all prisoners die.*
>
> *Before the first prisoner enters the room, the prisoners may discuss strategy—but may not communicate once the first prisoner enters to look in the drawers.*

What is the prisoners' best strategy? Are they doomed to die?


89
60
26
67
95
22
18
40
63

The 100 prisoners problem

Consider the first prisoner. As the permutation of numbers in the drawers is random the probability of finding his/her number is clearly 1/2. As prisoners cannot communicate, it seems that the probability that **all** them find their respective numbers is

$$\frac{1}{2^{100}} \approx 8 \cdot 10^{-31},$$

an extremely tiny number and the situation looks hopeless...

The 100 prisoners problem

But the analysis above only applies if each prisoner makes his choices at random (or the director changed the permutation for every prisoner). However, the permutation of numbers in the cupboard remains fixed through the process and the prisoners can devise and agree on a strategy that improves their chances of survival.

The 100 prisoners problem

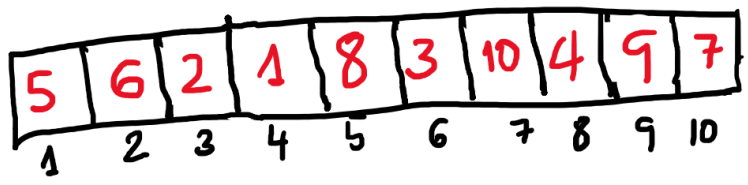
For example, suppose that all prisoners with numbers 1 to 50 open the drawers 1 to 50 and the prisoners with numbers 51 to 100 open drawers 51 to 100. If the random permutation is such that all numbers 1 to 50 are (in some order) in the drawers 1 to 50—and hence the numbers 51 to 100 are in the drawers 51 to 100 the prisoners succeed. There are $50! \cdot 50!$ permutations for which the prisoners strategy succeeds, but the probability of survival $(50! \cdot 50!)/100! \approx 10^{-29}$ is still too low (albeit $> 1/2^{100}$, it is now 12.56 times larger).

The 100 prisoners problem

The prisoners can do much better though. Each prisoner opens the drawer labeled with his/her number. If the number there is his/her number, done. Otherwise, opens the drawer indicated by the number just seen. This step is repeated until the prisoner finds his numer or has opened 50 drawers (and everyone is then executed).

Recall that a permutation is a set of cycles. If the number of a prisoner belongs to a cycle of length $\leqslant 50$ that prisoner will find it using the procedure described above. If the number belongs to a cycle of length > 50 then that prisoner might not find his/her number and then the whole pack of 100 prisoners is executed.

The 100 prisoners problem



$n = 10$

Prisoner #1 opens: 1 5 8 4

Prisoner #2 opens 2 6 3

...

All open ≤ 5 drawers! Success!!

The 100 prisoners problem

A permutation of 100 numbers can contain at most one cycle of length > 50. We can then claim the following: if all cycles in the random permutation are of length $\leqslant 50$ then the prisoners survive. If the random permutation contains one cycle of length $\ell > 50$ then all the prisoners die, because at least one of the prisoners in the cycle of length ℓ won't find his/her number after opening 50 drawers.

The 100 prisoners problem

Then

$$\begin{aligned}\mathbb{P}[\text{all die}] &= \mathbb{P}[\text{random permutation contains cycle of length} > 50] \\ &= \sum_{\ell=51}^{100} \mathbb{P}[\text{random permutation contains cycle of length } \ell]\end{aligned}$$

The 100 prisoners problem

Consider a permutation with a largest cycle of length $\ell > 50$. All other cycles must be of length < 50.
The ℓ numbers in the largest cycle can be arranged in $(\ell - 1)!$ different ways (fix the smallest and then add the other $\ell - 1$ numbers in any way). The other $(100 - \ell)$ numbers not in the largest cycle can be arranged in any way: there are $(100 - \ell)!$ such ways. And there are $\binom{100}{\ell}$ ways to choose the numbers which make up the largest cycle.

$$\mathbb{P}[\text{random permutation contains cycle of length } \ell] = \frac{1}{100!} \cdot \binom{100}{\ell} \cdot (\ell - 1)! \cdot (100 - \ell)! = \frac{1}{\ell}$$

The 100 prisoners problem

Then, with $H_n := \sum_{1 \leqslant k \leqslant n} 1/k$ (the nth harmonic number)

$$\mathbb{P}[\text{all die}] = H_{100} - H_{50} \approx 0.6881721793$$

But that means that the probability that all survive "skyrockets" to $1 - (H_{100} - H_{50}) \approx 0.3118278207$!!
In 2006 Eugene Curtin and Max Warshauer proved that the cycle-following strategy is actually optimal.