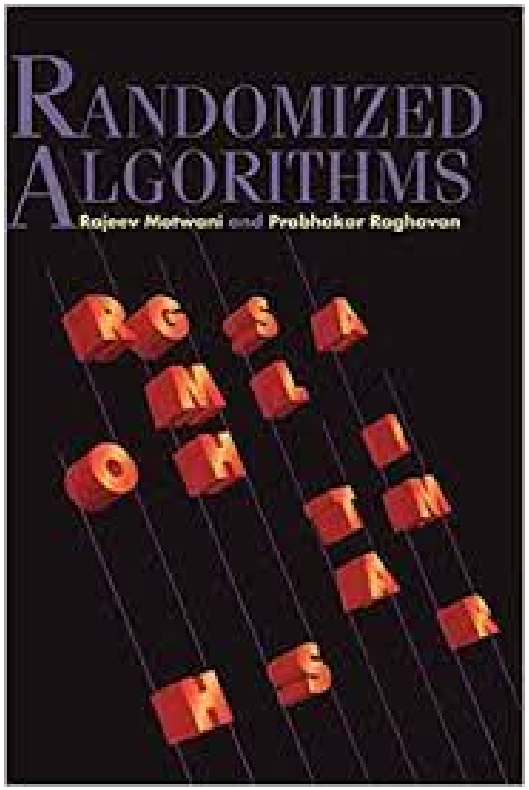
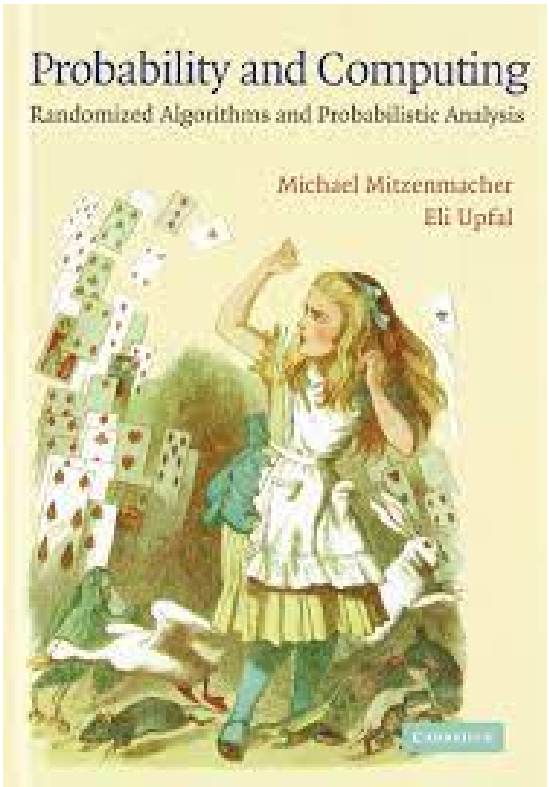
Probabilistic algorithms

AA-GEI FIB, UPC

Spring 2025-2026

More references





Probability and computers

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

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

for a, b constants and m a large integer.

Probability and computers

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

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

for a, b constants and m a large integer.

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

Probability and computers

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

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

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.

Probability and computers

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

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

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.

The usual way to generate a vector with a sequence of pseudorandom bits is:

Probability and computers

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

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

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.

The usual way to generate a vector with a sequence of pseudorandom bits is:

```
for (i = 0; i < n; i++) do
    values[i]=rand() % 2;
end for
```

Some applications of probability in CS

Some applications of probability in CS

- Algorithm design: Making algorithms run faster (in average) by introducing probability choices, against "bad" inputs.

Some applications of probability in CS

- Algorithm design: Making algorithms run faster (in average) by introducing probability choices, against "bad" inputs.
- Data structure: when implementing most of the used data structures, e.g. dictionaries, the use of probability helps to speed up search and reduce space. probability distributions.

Some applications of probability in CS

- Algorithm design: Making algorithms run faster (in average) by introducing probability choices, against "bad" inputs.
- Data structure: when implementing most of the used data structures, e.g. dictionaries, the use of probability helps to speed up search and reduce space. probability distributions.
- Studying and design mechanisms for large complex networks: The design of algorithms for Internet, WWW, Facebook, etc, is based in the design of realistic probabilistic models for those huge networks.

Randomization and algorithms: Probabilistic analysis

For a deterministic algorithm, when it seems that only a few "instances" may bias the complexity of the algorithm, we can perform a probabilistic analysis of the deterministic algorithm as follows:

Randomization and algorithms: Probabilistic analysis

For a deterministic algorithm, when it seems that only a few "instances" may bias the complexity of the algorithm, 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.

Randomization and algorithms: Probabilistic analysis

For a deterministic algorithm, when it seems that only a few "instances" may bias the complexity of the algorithm, 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.
- Then, the number of steps is a random variable $T(n)$. We can compute its expected value $\mu = \mathbf{E}[T(n)]$.

Randomization and algorithms: Probabilistic analysis

For a deterministic algorithm, when it seems that only a few "instances" may bias the complexity of the algorithm, 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.
- Then, the number of steps is a random variable $T(n)$. We can compute its expected value $\mu = \mathbf{E}[T(n)]$.
- We also need to prove concentration, i.e. with high probability, for most of the imputs, $T(n)$ is near μ.

Randomized algorithms

A randomized algorithms takes random choices and continues the computation according to the output of the random choices.

Randomized algorithms

A randomized algorithms takes random choices and continues the computation according to the output of the random choices.
Note that now the time complexity and the algorithm's output are random variables.

Randomized algorithms

A randomized algorithms takes random choices and continues the computation according to the output of the random choices.
Note that now the time complexity and the algorithm's output are random variables.
There are two main types of probabilistic algorithms:

Randomized algorithms

A randomized algorithms takes random choices and continues the computation according to the output of the random choices.
Note that now the time complexity and the algorithm's output are random variables.
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 (yes/not) the error can be in one/both direction, *one-side/two-sided error*. In Monte-Carlo algorithms it is important to bound the error probability.

Randomized algorithms

A randomized algorithms takes random choices and continues the computation according to the output of the random choices.
Note that now the time complexity and the algorithm's output are random variables.
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 (yes/not) the error can be in one/both direction, *one-side/two-sided 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.

Randomized algorithms

A randomized algorithms takes random choices and continues the computation according to the output of the random choices.
Note that now the time complexity and the algorithm's output are random variables.
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 (yes/not) the error can be in one/both direction, *one-side/two-sided 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.

Randomized algorithms

A randomized algorithms takes random choices and continues the computation according to the output of the random choices.
Note that now the time complexity and the algorithm's output are random variables.
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 (yes/not) the error can be in one/both direction, *one-side/two-sided 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 mostly with Monte-Carlo algs.

A randomized sorting algorithm

What do you know about QuickSort?

A randomized sorting algorithm

What do you know about QuickSort?

- General deterministic sorting algorithm

A randomized sorting algorithm

What do you know about QuickSort?

- General deterministic sorting algorithm
- Runs in time

A randomized sorting algorithm

What do you know about QuickSort?

- General deterministic sorting algorithm
- Runs in time $O(n^2)$

A randomized sorting algorithm

What do you know about QuickSort?

- General deterministic sorting algorithm
- Runs in time $O(n^2)$
- Average time

A randomized sorting algorithm

What do you know about QuickSort?

- General deterministic sorting algorithm
- Runs in time $O(n^2)$
- Average time $O(n \log n)$

A randomized sorting algorithm

What do you know about QuickSort?

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

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

A randomized sorting algorithm

A randomized sorting algorithm

Input a vector $A[n]$
Compute a uniform random permutation of $[n]$ in B
Rearrange A according to B
Run Quicksort on A

A randomized sorting algorithm

Input a vector $A[n]$
Compute a uniform random permutation of $[n]$ in B
Rearrange A according to B
Run Quicksort on A

The algorithm reaches our goal, if we can compute a random permutation within the right 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 $\Omega = \{\pi_1, \pi_2, \ldots, \pi_{n!}\}$, i.e. $|\Omega| = n!$.

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)

Random-Perm (n)
for $i = 0$ to $n-1$ **do**
  $\pi[i] = i$
end for
for $i = n-1$ to 1 **do**
  choose $j = \text{Rand}(i+1)$
  Interchange $\pi[j]$ and $\pi[i]$
end for

Rand(i) provides a random number in $[0, i)$.

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 $O(n)$

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 $O(n)$
- 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).

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 $O(n)$
- 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

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

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 $O(n)$
- 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.

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 $O(n)$
- 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.

Principle of deferred decisions

Not to assume that the entire set of random choices is made in advance. Rather, at each step of the process concentrate only on the random choices that are relevant to the algorithm outcome

When applicable it provides a simplified probability space to perform the probabilistic analysis.

Analyzing the Clock Solitaire game

From MR 3.5

The Clock Solitaire game: randomly shuffle a standard pack of 52 cards. Then, split the cards into 13 piles of 4 cards each; label piles as A, 2, . . . , 10, J, Q, K; take the first card from the "K" pile; take the next card from the pile "X", where X is the value of the previous card taken; repeat until:

- either all cards removed ("win")
- or you get stuck ("lose")

We want to evaluate the probability of "win".

Game termination?

- The last card we take before the game ends (either winning or loosing) is a "K".

Game termination?

- The last card we take before the game ends (either winning or loosing) is a "K".
- Let us assume that at iteration j we draw card X but the pile X is empty (thus the game terminates).

Game termination?

- The last card we take before the game ends (either winning or loosing) is a "K".
- Let us assume that at iteration j we draw card X but the pile X is empty (thus the game terminates).
- Let $X \neq K$ (i.e. we lose). Because pile X is empty and $X \neq K$, we must have already drawn (prior to draw j) 4 X cards. But then, we can not draw an X card at the j-th iteration, a contradiction.

Game termination?

- The last card we take before the game ends (either winning or loosing) is a "K".
- Let us assume that at iteration j we draw card X but the pile X is empty (thus the game terminates).
- Let $X \neq K$ (i.e. we lose). Because pile X is empty and $X \neq K$, we must have already drawn (prior to draw j) 4 X cards. But then, we can not draw an X card at the j-th iteration, a contradiction.
- There is no contradiction if the last card is a "K" and all other cards have been already removed (in that case the game terminates with win).

Game win?

- We win if the fourth "K" card is drawn at the 52 iteration.

Game win?

- We win if the fourth "K" card is drawn at the 52 iteration.
- Whenever we draw for the 1st, 2nd or 3rd time a "K" card, the game does not terminate because the K pile is not empty so we can continue.

Game win?

- We win if the fourth "K" card is drawn at the 52 iteration.
- Whenever we draw for the 1st, 2nd or 3rd time a "K" card, the game does not terminate because the K pile is not empty so we can continue.
- When the fourth K is drawn at the 52nd iteration then all cards are removed and the game's result is "win"

The probability of win?

According to the previous observations

$$\begin{aligned} Pr\{win\} &= Pr\{\text{4th ``K'' at the 52nd iteration}\} \\ &= \frac{\#\text{game evolutions: 52nd card} = \text{4th ``K''}}{\#\text{all game evolutions}} \end{aligned}$$

Considering all possible game evolutions is a rather naive approach since we have to count all ways to partition the 52 cards into 13 distinct piles, with an ordering on the 4 cards in each pile. This complicates the probability evaluation because of the dependence introduced by each random draw of a card.

We define another probability space that better captures the random dynamics of the game evolution.

The principle of deferred decisions

- Basic idea: rather than fix (and enumerate) the entire set of potential random choices in advance, instead let the random choices unfold with the progress of the random experiment.

The principle of deferred decisions

- Basic idea: rather than fix (and enumerate) the entire set of potential random choices in advance, instead let the random choices unfold with the progress of the random experiment.
- In this particular game at each draw any card not drawn yet is equally likely to be drawn.

The principle of deferred decisions

- Basic idea: rather than fix (and enumerate) the entire set of potential random choices in advance, instead let the random choices unfold with the progress of the random experiment.
- In this particular game at each draw any card not drawn yet is equally likely to be drawn.
- A winning game corresponds to a dynamics where the first 51 random draws include 3 "K" cards exactly.

The principle of deferred decisions

- Basic idea: rather than fix (and enumerate) the entire set of potential random choices in advance, instead let the random choices unfold with the progress of the random experiment.
- In this particular game at each draw any card not drawn yet is equally likely to be drawn.
- A winning game corresponds to a dynamics where the first 51 random draws include 3 "K" cards exactly.
- This is equivalent to draw the 4th "K" at the 52nd iteration.

The principle of deferred decisions

- Basic idea: rather than fix (and enumerate) the entire set of potential random choices in advance, instead let the random choices unfold with the progress of the random experiment.
- In this particular game at each draw any card not drawn yet is equally likely to be drawn.
- A winning game corresponds to a dynamics where the first 51 random draws include 3 "K" cards exactly.
- This is equivalent to draw the 4th "K" at the 52nd iteration.
- So we forget how the first 51 draws came out and focus on the 52nd draw, which must be a "K".

The probability of win

- We actually have 13x4=52 distinct positions (13 piles, 4 positions each) where 52 distinct cards are placed. This gives a total of 52! different placements.

The probability of win

- We actually have 13x4=52 distinct positions (13 piles, 4 positions each) where 52 distinct cards are placed. This gives a total of 52! different placements.
- Each game evolution actually corresponds to an ordered permutation of the 52 cards.

The probability of win

- We actually have 13x4=52 distinct positions (13 piles, 4 positions each) where 52 distinct cards are placed. This gives a total of 52! different placements.
- Each game evolution actually corresponds to an ordered permutation of the 52 cards.
- The winning permutations are those where the 52nd card is a "K" (4 ways) and the 51 preceding cards are arbitrarily chosen (51!). Thus:

$$Pr\{win\} = \frac{4\,51!}{52!} = \frac{4}{52} = \frac{1}{13}.$$

The probability of win

- We actually have 13x4=52 distinct positions (13 piles, 4 positions each) where 52 distinct cards are placed. This gives a total of 52! different placements.
- Each game evolution actually corresponds to an ordered permutation of the 52 cards.
- The winning permutations are those where the 52nd card is a "K" (4 ways) and the 51 preceding cards are arbitrarily chosen (51!). Thus:

$$Pr\{win\} = \frac{4\,51!}{52!} = \frac{4}{52} = \frac{1}{13}.$$

- A simpler way to get the same: The probability is $\frac{1}{13}$ because of symmetry (e.g. the type of the 52nd card is random uniform among all 13 types).

The probability of win

- We actually have 13x4=52 distinct positions (13 piles, 4 positions each) where 52 distinct cards are placed. This gives a total of 52! different placements.
- Each game evolution actually corresponds to an ordered permutation of the 52 cards.
- The winning permutations are those where the 52nd card is a "K" (4 ways) and the 51 preceding cards are arbitrarily chosen (51!). Thus:

$$Pr\{win\} = \frac{4\,51!}{52!} = \frac{4}{52} = \frac{1}{13}.$$

- A simpler way to get the same: The probability is $\frac{1}{13}$ because of symmetry (e.g. the type of the 52nd card is random uniform among all 13 types).
- The idea was to defer, i.e. first consider the last choice and then conditionally the previous ones!

Checking matrix multiplication

Problem: Given 3 square matrices ($n \times n$), A, B and C, we want to see if $A \times B = C$.
Easy solution: compute $A \times B$ and compare with C.

$n \times n$ matrix multiplication:

1. Naive algorithm: $O(n^3)$
2. Strassen (1969): $O(n^{2.81})$
3. Coppersmith-Winograd (1987): $O(n^{2.376})$
4. Vassilevska (2015): $O(n^{2.373})$

Can we (randomly) check in $O(n^2)$ if $A \times B = C$?

Freivald's algorithm for checking if $A \times B = C$ (1977)

From MU 1.3, MR 3.5

Given $n \times n$ matrices A, B, C

Freivald(A, B, C)
choose u.a.r. $r \in \{0,1\}^n$
if $A(Br) = Cr$ **then**
  output true
else
  output false
end if

Choosing u.a.r. r can be done choosing independently with probability $1/2$ each of its n bits. This makes the probability of any given r $1/2^n$, and the cost of generate the vector $O(n)$.

Freivald's algorithm for checking if $A \times B = C$ (1977)

From MU 1.3, MR 3.5

Given $n \times n$ matrices A, B, C

Freivald(A, B, C)
choose u.a.r. $r \in \{0,1\}^n$
if $A(Br) = Cr$ **then**
  output true
else
  output false
end if

The time complexity of Freivald's is $\Theta(n^2)$.
Notice: if $AB = C$ the algorithm yields always the correct answer.
It could be that $AB \neq C$ and the algorithms may yield the wrong answer

Error probability

Theorem

If $AB \neq C$ *then* $\mathbf{Pr}\left[A(B(r)) = Cr\right] \leq \frac{1}{2}$.

Proof.

- First trick: As $AB \neq C$ taking $D = AB - C$, then $D \neq (0)$.
  $\Rightarrow \exists d_{ij} \in D$ s.t. $d_{ij} \neq 0$. W.l.o.g. assume $d_{11} \neq 0$.

Error probability

Theorem

If $AB \neq C$ *then* $\mathbf{Pr}\left[A(B(r)) = Cr\right] \leq \frac{1}{2}$.

Proof.

- First trick: As $AB \neq C$ taking $D = AB - C$, then $D \neq (0)$.
  $\Rightarrow \exists d_{ij} \in D$ s.t. $d_{ij} \neq 0$. W.l.o.g. assume $d_{11} \neq 0$.
  If $\exists r$ s.t. $A(Br)) = Cr$ then $Dr = 0$.

Error probability

Theorem

If $AB \neq C$ *then* $\mathbf{Pr}\left[A(B(r)) = Cr\right] \leq \frac{1}{2}$.

Proof.

- First trick: As $AB \neq C$ taking $D = AB - C$, then $D \neq (0)$.
  $\Rightarrow \exists d_{ij} \in D$ s.t. $d_{ij} \neq 0$. W.l.o.g. assume $d_{11} \neq 0$.
  If $\exists r$ s.t. $A(Br)) = Cr$ then $Dr = 0$.
  $Dr = 0 \Rightarrow \sum_{j=1}^{n} d_{1j} r_j = 0$, but as $d_{11} \neq 0$ then $r_1 = \frac{-\sum_{j=2}^{n} d_{1j} r_j}{d_{11}}$.

Error probability

Theorem

If $AB \neq C$ *then* $\mathbf{Pr}\left[A(B(r)) = Cr\right] \leq \frac{1}{2}$.

Proof.

- First trick: As $AB \neq C$ taking $D = AB - C$, then $D \neq (0)$.
  $\Rightarrow \exists d_{ij} \in D$ s.t. $d_{ij} \neq 0$. W.l.o.g. assume $d_{11} \neq 0$.
  If $\exists r$ s.t. $A(Br)) = Cr$ then $Dr = 0$.
  $Dr = 0 \Rightarrow \sum_{j=1}^{n} d_{1j} r_j = 0$, but as $d_{11} \neq 0$ then $r_1 = \frac{-\sum_{j=2}^{n} d_{1j} r_j}{d_{11}}$.
- Second trick: Choose $r = (r_1, \ldots, r_n)$ from r_n to r_1 and stop at r_2, just before choosing r_1.

Error probability

Theorem

If $AB \neq C$ *then* $\mathbf{Pr}\left[A(B(r)) = Cr\right] \leq \frac{1}{2}$.

Proof.

- First trick: As $AB \neq C$ taking $D = AB - C$, then $D \neq (0)$.
  $\Rightarrow \exists d_{ij} \in D$ s.t. $d_{ij} \neq 0$. W.l.o.g. assume $d_{11} \neq 0$.
  If $\exists r$ s.t. $A(Br)) = Cr$ then $Dr = 0$.
  $Dr = 0 \Rightarrow \sum_{j=1}^{n} d_{1j} r_j = 0$, but as $d_{11} \neq 0$ then $r_1 = \frac{-\sum_{j=2}^{n} d_{1j} r_j}{d_{11}}$.
- Second trick: Choose $r = (r_1, \ldots, r_n)$ from r_n to r_1 and stop at r_2, just before choosing r_1.
  But then, $r_1 = \frac{-\sum_{j=2}^{n} d_{1j} r_j}{d_{11}}$, for r_1 u.a.r. the eq holds with prob. $1/2$

□

Randomized algorithms and amplification

- Freivald's algorithm finish always in finite time ($\Theta(n^2)$) but may output the wrong answer. Thus it is a one-side error Monte-Carlo algorithm, we may get the wrong answer with a "small" probability ($\leq 1/2$).

Randomized algorithms and amplification

- Freivald's algorithm finish always in finite time ($\Theta(n^2)$) but may output the wrong answer. Thus it is a one-side error Monte-Carlo algorithm, we may get the wrong answer with a "small" probability ($\leq 1/2$).
- One-side error Monte-Carlo algorithms have the nice characteristic that can be amplified:
  Each run of the algorithm can be considered as an independent "experiment", so they can be repeated decreasing the probability of error.
- If we repeat k times Freivald's, each time we generate a new v, the answer keep being true, the probability of error is $\leq 1/2^k$.