Part I

1.Basic concepts and Examples of Randomized Algorithms

Chapter 1. INTRODUCTION

The main aim of the first chapter of the lecture is:

Chapter 1. INTRODUCTION

The main aim of the first chapter of the lecture is:

1. To present several views of randomized algorithms

Chapter 1. INTRODUCTION

The main aim of the first chapter of the lecture is:

1. To present several views of randomized algorithms
2. to present several interesting examples of simple randomized algorithms;

Chapter 1. INTRODUCTION

The main aim of the first chapter of the lecture is:

1. To present several views of randomized algorithms
2. to present several interesting examples of simple randomized algorithms;
3. to demonstrate advantages of randomized algorithms and methods of their analysis.

Chapter 1. INTRODUCTION

The main aim of the first chapter of the lecture is:

1. To present several views of randomized algorithms
2. to present several interesting examples of simple randomized algorithms;
3. to demonstrate advantages of randomized algorithms and methods of their analysis.

The second aim of this chapter is to introduce main complexity classes for randomized algorithms.

Chapter 1. INTRODUCTION

The main aim of the first chapter of the lecture is:

1. To present several views of randomized algorithms
2. to present several interesting examples of simple randomized algorithms;
3. to demonstrate advantages of randomized algorithms and methods of their analysis.

The second aim of this chapter is to introduce main complexity classes for randomized algorithms.

Third aim is to show relations between randomized and deterministic complexity classes.

Fourth aim is to discuss in some details puzzling concept of randomness, at least in some details.

Revolution in designing algorithms

The idea that randomized algorithm can be VERY useful can be seen as the main revolutionary idea in the design of algorithms in the last 2200 years.

Deterministic versus randomized algorithms

Usual (deterministic) algorithm is a set of rules how to solve some problem, step by step, in which each next step is uniquely determined.

Deterministic versus randomized algorithms

Usual (deterministic) algorithm is a set of rules how to solve some problem, step by step, in which each next step is uniquely determined. As a consequence, each time a deterministic algorithm A is applied on the same input it produces the same output.

Randomized (probabilistic) algorithm is a set of rules how to solve some problem, step by step, in which each next step is chosen, with a determined probability, from a finite set of possible steps.

Deterministic versus randomized algorithms

Usual (deterministic) algorithm is a set of rules how to solve some problem, step by step, in which each next step is uniquely determined. As a consequence, each time a deterministic algorithm A is applied on the same input it produces the same output.

Randomized (probabilistic) algorithm is a set of rules how to solve some problem, step by step, in which each next step is chosen, with a determined probability, from a finite set of possible steps. As a consequence, a randomized algorithm A may produce different outputs when applied more than one times to the same input.

WHY to use RANDOMIZED ALGORITHMS?

WHY to use RANDOMIZED ALGORITHMS?

Randomized algorithms are such algorithms that may make random choices (such as ones obtained using coin-tossing) concerning the ways they have to continue, during their executions.

WHY to use RANDOMIZED ALGORITHMS?

Randomized algorithms are such algorithms that may make random choices (such as ones obtained using coin-tossing) concerning the ways they have to continue, during their executions. As a consequence, their outcomes do not depend only on their (external) problem inputs.

WHY to use RANDOMIZED ALGORITHMS?

Randomized algorithms are such algorithms that may make random choices (such as ones obtained using coin-tossing) concerning the ways they have to continue, during their executions. As a consequence, their outcomes do not depend only on their (external) problem inputs.

Advantages: There are several important reasons why randomized algorithms are of increasing importance:

WHY to use RANDOMIZED ALGORITHMS?

Randomized algorithms are such algorithms that may make random choices (such as ones obtained using coin-tossing) concerning the ways they have to continue, during their executions. As a consequence, their outcomes do not depend only on their (external) problem inputs.

Advantages: There are several important reasons why randomized algorithms are of increasing importance:

1. **Randomized algorithms are often faster than deterministic ones for the same problem either from the worst-case asymptotic point of view or/and from the numerical implementations point of view;**

WHY to use RANDOMIZED ALGORITHMS?

Randomized algorithms are such algorithms that may make random choices (such as ones obtained using coin-tossing) concerning the ways they have to continue, during their executions. As a consequence, their outcomes do not depend only on their (external) problem inputs.

Advantages: There are several important reasons why randomized algorithms are of increasing importance:

1. **Randomized algorithms are often faster than deterministic ones for the same problem either from the worst-case asymptotic point of view or/and from the numerical implementations point of view;**
2. Randomized algorithms **are often (much) simpler** than deterministic ones for the same problem;

WHY to use RANDOMIZED ALGORITHMS?

Randomized algorithms are such algorithms that may make random choices (such as ones obtained using coin-tossing) concerning the ways they have to continue, during their executions. As a consequence, their outcomes do not depend only on their (external) problem inputs.

Advantages: There are several important reasons why randomized algorithms are of increasing importance:

1. **Randomized algorithms are often faster than deterministic ones for the same problem either from the worst-case asymptotic point of view or/and from the numerical implementations point of view;**
2. Randomized algorithms **are often (much) simpler** than deterministic ones for the same problem;
3. **Randomized algorithms are often easier to analyze and/or reason about than deterministic ones**

WHY to use RANDOMIZED ALGORITHMS?

Randomized algorithms are such algorithms that may make random choices (such as ones obtained using coin-tossing) concerning the ways they have to continue, during their executions. As a consequence, their outcomes do not depend only on their (external) problem inputs.

Advantages: There are several important reasons why randomized algorithms are of increasing importance:

1. **Randomized algorithms are often faster than deterministic ones for the same problem either from the worst-case asymptotic point of view or/and from the numerical implementations point of view;**
2. Randomized algorithms **are often (much) simpler** than deterministic ones for the same problem;
3. **Randomized algorithms are often easier to analyze and/or reason about than deterministic ones especially when applied in counter-intuitive settings;**

WHY to use RANDOMIZED ALGORITHMS?

Randomized algorithms are such algorithms that may make random choices (such as ones obtained using coin-tossing) concerning the ways they have to continue, during their executions. As a consequence, their outcomes do not depend only on their (external) problem inputs.

Advantages: There are several important reasons why randomized algorithms are of increasing importance:

1. **Randomized algorithms are often faster than deterministic ones for the same problem either from the worst-case asymptotic point of view or/and from the numerical implementations point of view;**
2. Randomized algorithms **are often (much) simpler** than deterministic ones for the same problem;
3. **Randomized algorithms are often easier to analyze and/or reason about than deterministic ones especially when applied in counter-intuitive settings;**
4. Randomized algorithms **have often more easily interpretable outputs**, which is of interests in applications where analyst's time rather than just computation time is also of interest;

WHY to use RANDOMIZED ALGORITHMS?

Randomized algorithms are such algorithms that may make random choices (such as ones obtained using coin-tossing) concerning the ways they have to continue, during their executions. As a consequence, their outcomes do not depend only on their (external) problem inputs.

Advantages: There are several important reasons why randomized algorithms are of increasing importance:

1. **Randomized algorithms are often faster than deterministic ones for the same problem either from the worst-case asymptotic point of view or/and from the numerical implementations point of view;**
2. Randomized algorithms **are often (much) simpler** than deterministic ones for the same problem;
3. **Randomized algorithms are often easier to analyze and/or reason about than deterministic ones especially when applied in counter-intuitive settings;**
4. Randomized algorithms **have often more easily interpretable outputs**, which is of interests in applications where analyst's time rather than just computation time is also of interest;
5. **Randomized numerical algorithms are often better organized better to exploit parallelism of modern computer architectures**

WHY CAN RANDOMIZED ALGORITHMS BE MORE EFFICIENT?

Two simplified explanations:

WHY CAN RANDOMIZED ALGORITHMS BE MORE EFFICIENT?

Two simplified explanations:

(1) A systematic search for a solution must often go through a time-consuming computation paths corresponding to some (few) very unlikely pathological cases.

WHY CAN RANDOMIZED ALGORITHMS BE MORE EFFICIENT?

Two simplified explanations:

(1) A systematic search for a solution must often go through a time-consuming computation paths corresponding to some (few) very unlikely pathological cases. **A randomized search for a solution can often avoid, with a sufficiently large probability, such time-consuming paths.**

WHY CAN RANDOMIZED ALGORITHMS BE MORE EFFICIENT?

Two simplified explanations:

(1) A systematic search for a solution must often go through a time-consuming computation paths corresponding to some (few) very unlikely pathological cases. **A randomized search for a solution can often avoid, with a sufficiently large probability, such time-consuming paths.**

(2) For some algorithmic problems P, for each deterministic algorithm for P there are also bad inputs that force the algorithm to do very long computations.

WHY CAN RANDOMIZED ALGORITHMS BE MORE EFFICIENT?

Two simplified explanations:

(1) A systematic search for a solution must often go through a time-consuming computation paths corresponding to some (few) very unlikely pathological cases. **A randomized search for a solution can often avoid, with a sufficiently large probability, such time-consuming paths.**

(2) For some algorithmic problems P, for each deterministic algorithm for P there are also bad inputs that force the algorithm to do very long computations. However, for P there may be also sets of deterministic algorithms such that for any input most of these algorithms are fast and a random choice of one of the algorithms from such a set provides very likely fast a proper output.

WHY CAN RANDOMIZED ALGORITHMS BE MORE EFFICIENT?

Two simplified explanations:

(1) A systematic search for a solution must often go through a time-consuming computation paths corresponding to some (few) very unlikely pathological cases. **A randomized search for a solution can often avoid, with a sufficiently large probability, such time-consuming paths.**

(2) For some algorithmic problems P, for each deterministic algorithm for P there are also bad inputs that force the algorithm to do very long computations. However, for P there may be also sets of deterministic algorithms such that for any input most of these algorithms are fast and a random choice of one of the algorithms from such a set provides very likely fast a proper output.

Moreover, **quantum algorithms** are, in principle, randomized.

WHY CAN RANDOMIZED ALGORITHMS BE MORE EFFICIENT?

Two simplified explanations:

(1) A systematic search for a solution must often go through a time-consuming computation paths corresponding to some (few) very unlikely pathological cases. **A randomized search for a solution can often avoid, with a sufficiently large probability, such time-consuming paths.**

(2) For some algorithmic problems P, for each deterministic algorithm for P there are also bad inputs that force the algorithm to do very long computations. However, for P there may be also sets of deterministic algorithms such that for any input most of these algorithms are fast and a random choice of one of the algorithms from such a set provides very likely fast a proper output.

Moreover, **quantum algorithms** are, in principle, randomized.

Randomized complexity classes offer also a plausible way to extend the very important *feasibility* concept.

VIEWS of RANDOMIZED ALGORITHMS:

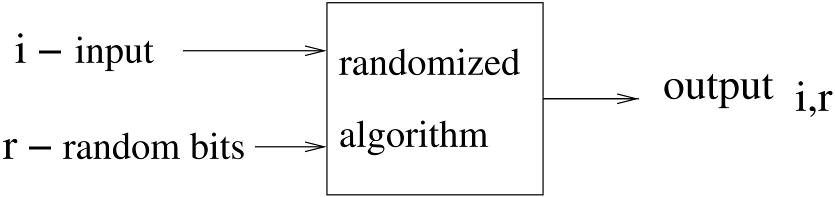
VIEWS of RANDOMIZED ALGORITHMS:

A **randomized algorithm** $\mathcal{A}$ is an algorithm that at each new run receives, in addition to its input i, a new stream/string r of random bits which are then used to specify outcomes of the subsequent random choices (or coin tossing) during the execution of the algorithm.

VIEWS of RANDOMIZED ALGORITHMS:

A **randomized algorithm** $\mathcal{A}$ is an algorithm that at each new run receives, in addition to its input i, a new stream/string r of random bits which are then used to specify outcomes of the subsequent random choices (or coin tossing) during the execution of the algorithm.

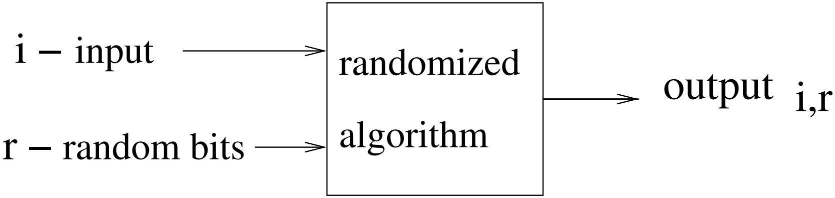
Streams r of random bits are assumed to be independent of the input i for the algorithm $\mathcal{A}$.



VIEWS of RANDOMIZED ALGORITHMS:

A **randomized algorithm** $\mathcal{A}$ is an algorithm that at each new run receives, in addition to its input i, a new stream/string r of random bits which are then used to specify outcomes of the subsequent random choices (or coin tossing) during the execution of the algorithm.

Streams r of random bits are assumed to be independent of the input i for the algorithm $\mathcal{A}$.

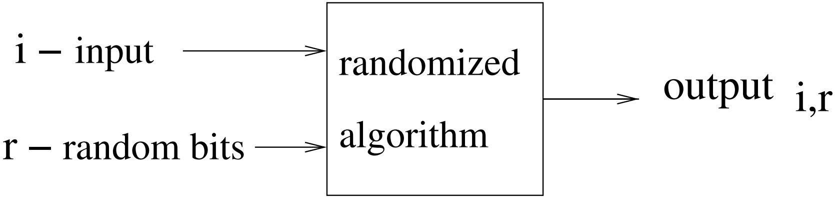


Important comment: Repeated runs of a randomized algorithm with the same input data (but not same random input strings) may not, in general, produce the same results.

VIEWS of RANDOMIZED ALGORITHMS:

A **randomized algorithm** $\mathcal{A}$ is an algorithm that at each new run receives, in addition to its input i, a new stream/string r of random bits which are then used to specify outcomes of the subsequent random choices (or coin tossing) during the execution of the algorithm.

Streams r of random bits are assumed to be independent of the input i for the algorithm $\mathcal{A}$.



Important comment: Repeated runs of a randomized algorithm with the same input data (but not same random input strings) may not, in general, produce the same results. Outcomes, of $\mathcal{A}(i, r)$, will depend not only on i, but also on r.

A BIT of HISTORY

The concept of **algorithm** is very old.

A BIT of HISTORY

The concept of **algorithm** is very old. It goes back to Euclid and Al Khwarizmi in around 300 BC and 800 AC.

A BIT of HISTORY

The concept of **algorithm** is very old. It goes back to Euclid and Al Khwarizmi in around 300 BC and 800 AC.

One of the key points of this concept was that each time a (deterministic) algorithm takes the same input it provides the same output.

A BIT of HISTORY

The concept of **algorithm** is very old. It goes back to Euclid and Al Khwarizmi in around 300 BC and 800 AC.

One of the key points of this concept was that each time a (deterministic) algorithm takes the same input it provides the same output.

The concept of **randomized algorithm** is from 20th century and got larger attention practically only after 1977.

A BIT of HISTORY

The concept of **algorithm** is very old. It goes back to Euclid and Al Khwarizmi in around 300 BC and 800 AC.

One of the key points of this concept was that each time a (deterministic) algorithm takes the same input it provides the same output.

The concept of **randomized algorithm** is from 20th century and got larger attention practically only after 1977.

One of the key points of this concept is that each time a (randomized) algorithm takes the same input it may provide different outcomes.

MODELS of RANDOMIZED ALGORITHMS I.

MODELS of RANDOMIZED ALGORITHMS I.

A randomized algorithm can be seen also in other ways:

MODELS of RANDOMIZED ALGORITHMS I.

A randomized algorithm can be seen also in other ways:

- As an algorithm that may, from time to time, toss a coin,

MODELS of RANDOMIZED ALGORITHMS I.

A randomized algorithm can be seen also in other ways:

- As an algorithm that may, from time to time, toss a coin, or read a (next) random bit from its special input stream of random bits,

MODELS of RANDOMIZED ALGORITHMS I.

A randomized algorithm can be seen also in other ways:

- As an algorithm that may, from time to time, toss a coin, or read a (next) random bit from its special input stream of random bits, and then proceeds depending on the outcome of the coin tossing (or of a chosen random bit).

MODELS of RANDOMIZED ALGORITHMS I.

A randomized algorithm can be seen also in other ways:

- As an algorithm that may, from time to time, toss a coin, or read a (next) random bit from its special input stream of random bits, and then proceeds depending on the outcome of the coin tossing (or of a chosen random bit).
- As a nondeterministic-like algorithm which has a probability assigned to each possible transition.

MODELS of RANDOMIZED ALGORITHMS I.

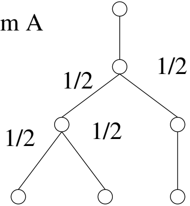
A randomized algorithm can be seen also in other ways:

- As an algorithm that may, from time to time, toss a coin, or read a (next) random bit from its special input stream of random bits, and then proceeds depending on the outcome of the coin tossing (or of a chosen random bit).

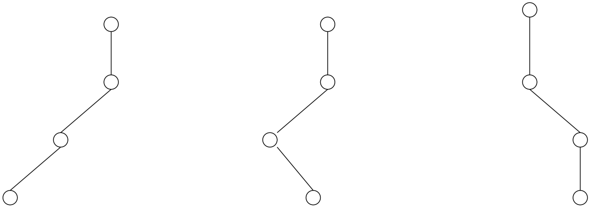
- As a nondeterministic-like algorithm which has a probability assigned to each possible transition.
- As a probability distribution on a set of deterministic algorithms - $\{\mathcal{A}_i, p_i\}_{i=1}^n$.

RANDOMIZED ALGORITHMS as PROBABILISTIC DISTRIBUTIONS on DETERMINISTIC ALGORITHMS

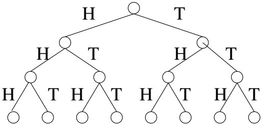
randomized algorithm A

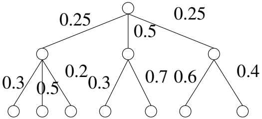


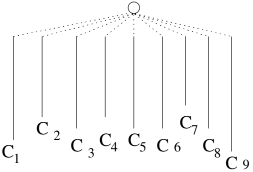
as a probabilistic distribution on three deterministic algorithms B, C, D



MODELS of RANDOMIZED ALGORITHMS II







$Pr(C_i)$

C_i are runs of dif. determ. alg.

STORY of RANDOMNESS

DOES RANDOMNESS EXIST? - I

DOES RANDOMNESS EXIST? - I

One of the fundamental questions (of science) has been, and actually still is, whether randomness really exists or whether term **randomness** is used only to deal with events the laws of which we do not fully understand.

DOES RANDOMNESS EXIST? - I

One of the fundamental questions (of science) has been, and actually still is, whether randomness really exists or whether term **randomness** is used only to deal with events the laws of which we do not fully understand. Two early views are:

DOES RANDOMNESS EXIST? - I

One of the fundamental questions (of science) has been, and actually still is, whether randomness really exists or whether term **randomness** is used only to deal with events the laws of which we do not fully understand. Two early views are:

> *The* ***randomness is the unknown*** *and Nature is determined in its fundamentals.*

Democritos (470-404 BC)

DOES RANDOMNESS EXIST? - I

One of the fundamental questions (of science) has been, and actually still is, whether randomness really exists or whether term **randomness** is used only to deal with events the laws of which we do not fully understand. Two early views are:

> *The* ***randomness is the unknown*** *and Nature is determined in its fundamentals.*

Democritos (470-404 BC)

By Democritos, the order conquered the world and this order is governed by unambiguous laws. By Leucippus, the teacher of Democritos.

DOES RANDOMNESS EXIST? - I

One of the fundamental questions (of science) has been, and actually still is, whether randomness really exists or whether term **randomness** is used only to deal with events the laws of which we do not fully understand. Two early views are:

> *The* ***randomness is the unknown*** *and Nature is determined in its fundamentals.*

Democritos (470-404 BC)

By Democritos, the order conquered the world and this order is governed by unambiguous laws. By Leucippus, the teacher of Democritos.

Nothing occurs at random, but everything for a reason and necessity.

DOES RANDOMNESS EXIST? - I

One of the fundamental questions (of science) has been, and actually still is, whether randomness really exists or whether term **randomness** is used only to deal with events the laws of which we do not fully understand. Two early views are:

> *The* **randomness is the unknown** *and Nature is determined in its fundamentals.*

Democritos (470-404 BC)

By Democritos, the order conquered the world and this order is governed by unambiguous laws. By Leucippus, the teacher of Democritos.

Nothing occurs at random, but everything for a reason and necessity.

By Democritus and Leucippus, the word **random** is used when we have an incomplete knowledge of some phenomena.

DOES RANDOMNESS EXIST? - I

One of the fundamental questions (of science) has been, and actually still is, whether randomness really exists or whether term **randomness** is used only to deal with events the laws of which we do not fully understand. Two early views are:

> *The* ***randomness is the unknown*** *and Nature is determined in its fundamentals.*

Democritos (470-404 BC)

By Democritos, the order conquered the world and this order is governed by unambiguous laws. By Leucippus, the teacher of Democritos.

Nothing occurs at random, but everything for a reason and necessity.

By Democritus and Leucippus, the word **random** is used when we have an incomplete knowledge of some phenomena. On the other side:

DOES RANDOMNESS EXIST? - I

One of the fundamental questions (of science) has been, and actually still is, whether randomness really exists or whether term **randomness** is used only to deal with events the laws of which we do not fully understand. Two early views are:

> *The* **randomness is the unknown** *and Nature is determined in its fundamentals.*

Democritos (470-404 BC)

By Democritos, the order conquered the world and this order is governed by unambiguous laws. By Leucippus, the teacher of Democritos.

Nothing occurs at random, but everything for a reason and necessity.

By Democritus and Leucippus, the word **random** is used when we have an incomplete knowledge of some phenomena. On the other side:

> **The randomness is objective, it is the proper nature of some events.**

Epikurus (341-270 BC)

DOES RANDOMNESS EXIST? - I

One of the fundamental questions (of science) has been, and actually still is, whether randomness really exists or whether term **randomness** is used only to deal with events the laws of which we do not fully understand. Two early views are:

> *The* **randomness is the unknown** *and Nature is determined in its fundamentals.*

Democritos (470-404 BC)

By Democritos, the order conquered the world and this order is governed by unambiguous laws. By Leucippus, the teacher of Democritos.

Nothing occurs at random, but everything for a reason and necessity.

By Democritus and Leucippus, the word **random** is used when we have an incomplete knowledge of some phenomena. On the other side:

> **The randomness is objective, it is the proper nature of some events.**

Epikurus (341-270 BC)

By Epikurus, there exists a true randomness that is independent of our knowledge.

VIEWS on RANDOMNESS in 19th CENTURY

Main arguments, before 20th century, why randomness does not exist:

VIEWS on RANDOMNESS in 19th CENTURY

Main arguments, before 20th century, why randomness does not exist:

God-argument: There is no place for randomness in a world created by God.

VIEWS on RANDOMNESS in 19th CENTURY

Main arguments, before 20th century, why randomness does not exist:

God-argument: There is no place for randomness in a world created by God.

Science-argument: Success of natural sciences and mechanical engineering in 19th century led to a belief that everything could be discovered and explained by deterministic causalities of a cause and the resulting effect.

VIEWS on RANDOMNESS in 19th CENTURY

Main arguments, before 20th century, why randomness does not exist:

God-argument: There is no place for randomness in a world created by God.

Science-argument: Success of natural sciences and mechanical engineering in 19th century led to a belief that everything could be discovered and explained by deterministic causalities of a cause and the resulting effect.

Emotional-argument: Randomness used to be identified with uncertainty or unpredictability or even chaos.

VIEWS on RANDOMNESS in 19th CENTURY

Main arguments, before 20th century, why randomness does not exist:

God-argument: There is no place for randomness in a world created by God.

Science-argument: Success of natural sciences and mechanical engineering in 19th century led to a belief that everything could be discovered and explained by deterministic causalities of a cause and the resulting effect.

Emotional-argument: Randomness used to be identified with uncertainty or unpredictability or even chaos.

There are only two possibilities, either a big chaos conquers the world, or order and law.

Marcus Aurelius

EINSTEIN versus BOHR

God does not roll dice.

EINSTEIN versus BOHR

God does not roll dice.

Albert Einstein, 1935, a strong opponent of randomness.

EINSTEIN versus BOHR

God does not roll dice.

Albert Einstein, 1935, a strong opponent of randomness.

The true God does not allow anybody to prescribe what he has to do.

EINSTEIN versus BOHR

God does not roll dice.

Albert Einstein, 1935, a strong opponent of randomness.

The true God does not allow anybody to prescribe what he has to do.

Famous reply by Niels Bohr - one of the fathers of quantum mechanics.

DOES GOD PLAY DICE? - NEW VIEWS

God does play even non-local dice.

DOES GOD PLAY DICE? - NEW VIEWS

God does play even non-local dice.

An observation, due to N. Gisin, on the basis that measurement of entangled states produces shared randomness.

DOES GOD PLAY DICE? - NEW VIEWS

God does play even non-local dice.

An observation, due to N. Gisin, on the basis that measurement of entangled states produces shared randomness.

God is not malicious and made Nature to produce, so useful, (shared) randomness.

This is what the outcomes of the theoretical informatics imply.

RANDOMNESS in NATURE

RANDOMNESS in NATURE

Two big scientific discoveries of 20th century changed the view on usefulness of randomness.

RANDOMNESS in NATURE

Two big scientific discoveries of 20th century changed the view on usefulness of randomness.

1. It has turned out that random mutations of DNA have to be considered as a crucial instrument of evolution.

RANDOMNESS in NATURE

Two big scientific discoveries of 20th century changed the view on usefulness of randomness.

1. It has turned out that random mutations of DNA have to be considered as a crucial instrument of evolution.
2. Quantum measurement yields, in principle, random outcomes.

RANDOMNESS

RANDOMNESS

- Randomness as a mathematical topic has been studied since 17th century.

RANDOMNESS

- Randomness as a mathematical topic has been studied since 17th century.
- Attempts to formalize chance by mathematical laws is somehow paradoxical because, a priory, chance (randomness) is the subject of no law.

RANDOMNESS

- Randomness as a mathematical topic has been studied since 17th century.
- Attempts to formalize chance by mathematical laws is somehow paradoxical because, a priory, chance (randomness) is the subject of no law.
- There is no proof that perfect randomness exists in the real world.

RANDOMNESS

- Randomness as a mathematical topic has been studied since 17th century.
- Attempts to formalize chance by mathematical laws is somehow paradoxical because, a priory, chance (randomness) is the subject of no law.
- There is no proof that perfect randomness exists in the real world.
- More exactly, there is no proof that quantum mechanical phenomena of the microworld can be exploited to provide a perfect source of randomness for the macroworld.

KOLMOGOROV COMPLEXITY

KOLMOGOROV COMPLEXITY

- Kolmogorov complexity $K_C(x)$ of a binary string x with respect to a universal computer C is the length of the shortest program for C that produces x.

KOLMOGOROV COMPLEXITY

- Kolmogorov complexity $K_C(x)$ of a binary string x with respect to a universal computer C is the length of the shortest program for C that produces x.
- The above definition is *basically* independent of the choice of C. Namely, it holds that for any other universal computer C' there is a constant $a_{C,C'}$ such that for any string x, $K_{C'}(x) \leq K_C(x) + a_{C,C'}$.

KOLMOGOROV COMPLEXITY

- Kolmogorov complexity $K_C(x)$ of a binary string x with respect to a universal computer C is the length of the shortest program for C that produces x.
- The above definition is *basically* independent of the choice of C. Namely, it holds that for any other universal computer C' there is a constant $a_{C,C'}$ such that for any string x, $K_{C'}(x) \leq K_C(x) + a_{C,C'}$.
- A string x is considered as **random** if $K_C(x) \approx |x|$, that is if x is incompressible.

KOLMOGOROV COMPLEXITY

- Kolmogorov complexity $K_C(x)$ of a binary string x with respect to a universal computer C is the length of the shortest program for C that produces x.
- The above definition is *basically* independent of the choice of C. Namely, it holds that for any other universal computer C' there is a constant $a_{C,C'}$ such that for any string x, $K_{C'}(x) \leq K_C(x) + a_{C,C'}$.
- A string x is considered as **random** if $K_C(x) \approx |x|$, that is if x is incompressible.
- Kolmogorov complexity is not computable.

KOLMOGOROV COMPLEXITY

- Kolmogorov complexity $K_C(x)$ of a binary string x with respect to a universal computer C is the length of the shortest program for C that produces x.
- The above definition is *basically* independent of the choice of C. Namely, it holds that for any other universal computer C' there is a constant $a_{C,C'}$ such that for any string x, $K_{C'}(x) \leq K_C(x) + a_{C,C'}$.
- A string x is considered as **random** if $K_C(x) \approx |x|$, that is if x is incompressible.
- Kolmogorov complexity is not computable.
- It is undecidable whether a given string is random.

KOLMOGOROV COMPLEXITY

- Kolmogorov complexity $K_C(x)$ of a binary string x with respect to a universal computer C is the length of the shortest program for C that produces x.
- The above definition is *basically* independent of the choice of C. Namely, it holds that for any other universal computer C' there is a constant $a_{C,C'}$ such that for any string x, $K_{C'}(x) \leq K_C(x) + a_{C,C'}$.
- A string x is considered as **random** if $K_C(x) \approx |x|$, that is if x is incompressible.
- Kolmogorov complexity is not computable.
- It is undecidable whether a given string is random.
- Until Kolmogorov complexity was introduced we had no meaningful way to talk about a given object being random.

PSEUDORANDOM GENERATORS STORY

PSEUDORANDOM GENERATORS STORY

Pseudorandom generators are algorithms that generate pseudorandom (almost random) strings or integers.

PSEUDORANDOM GENERATORS STORY

Pseudorandom generators are algorithms that generate pseudorandom (almost random) strings or integers.

Pseudorandom generators is an additional key concept of cryptography and of the design of efficient algorithms.

PSEUDORANDOM GENERATORS STORY

Pseudorandom generators are algorithms that generate pseudorandom (almost random) strings or integers.

Pseudorandom generators is an additional key concept of cryptography and of the design of efficient algorithms.

There is a variety of classical algorithms capable to generate pseudorandomness of different quality concerning randomness.

PSEUDORANDOM GENERATORS STORY

Pseudorandom generators are algorithms that generate pseudorandom (almost random) strings or integers.

Pseudorandom generators is an additional key concept of cryptography and of the design of efficient algorithms.

There is a variety of classical algorithms capable to generate pseudorandomness of different quality concerning randomness.

Quantum processes can generate perfect randomness and on this basis quantum (almost perfect) generators of randomness are already commercially available.

von NEUMANN EXAMPLE

von NEUMANN EXAMPLE

The concept of pseudorandom generators is quite old. An interesting example is due to John von Neumann:

von NEUMANN EXAMPLE

The concept of pseudorandom generators is quite old. An interesting example is due to John von Neumann:

Take an arbitrary integer x as the "seed"

von NEUMANN EXAMPLE

The concept of pseudorandom generators is quite old. An interesting example is due to John von Neumann:

Take an arbitrary integer x as the "seed"
and repeat the following process:

von NEUMANN EXAMPLE

The concept of pseudorandom generators is quite old. An interesting example is due to John von Neumann:

Take an arbitrary integer x as the "seed"
and repeat the following process:

compute x^2 and take a sequence of the middle digits of x^2 as a new "seed" x.

von NEUMANN EXAMPLE

The concept of pseudorandom generators is quite old. An interesting example is due to John von Neumann:

Take an arbitrary integer x as the "seed"
and repeat the following process:

compute x^2 and take a sequence of the middle digits of x^2 as a new "seed" x.

whenever you end such an iterative process, the final seed is a pseudorandom string of digits.

Von NEUMANN PSEUDORANDOM GENERATION

$$2356^2 = 5550736$$

Von NEUMANN PSEUDORANDOM GENERATION

$$2356^2 = 5550736$$
$$55073^2$$

Von NEUMANN PSEUDORANDOM GENERATION

$$2356^2 = 5550736$$
$$55073^2 = 3033035329$$

Von NEUMANN PSEUDORANDOM GENERATION

$$2356^2 = 5550736$$
$$55073^2 = 3033035329$$
$$330353^2$$

Von NEUMANN PSEUDORANDOM GENERATION

$$2356^2 = 5550736$$
$$55073^2 = 3033035329$$
$$330353^2 = 109133104609$$

Von NEUMANN PSEUDORANDOM GENERATION

$$2356^2 = 5550736$$
$$55073^2 = 3033035329$$
$$330353^2 = 109133104609$$
$$1331046^2 =$$

Von NEUMANN PSEUDORANDOM GENERATION

$$2356^2 = 5550736$$
$$55073^2 = 3033035329$$
$$330353^2 = 109133104609$$
$$1331046^2 =$$

SIMPLE PSEUDORANDOM GENERATORS

SIMPLE PSEUDORANDOM GENERATORS

Informally, a **pseudorandom generator** is a deterministic polynomial time algorithm which expands short random sequences (called **seeds**) into longer bit sequences such that the resulting probability distribution is in polynomial time indistinguishable from the uniform probability distribution.

SIMPLE PSEUDORANDOM GENERATORS

Informally, a **pseudorandom generator** is a deterministic polynomial time algorithm which expands short random sequences (called **seeds**) into longer bit sequences such that the resulting probability distribution is in polynomial time indistinguishable from the uniform probability distribution.

Example. **Linear congruential generator**

SIMPLE PSEUDORANDOM GENERATORS

Informally, a **pseudorandom generator** is a deterministic polynomial time algorithm which expands short random sequences (called **seeds**) into longer bit sequences such that the resulting probability distribution is in polynomial time indistinguishable from the uniform probability distribution.

Example. **Linear congruential generator**

One chooses n-bit numbers m, a, b, X_0 and generates an n^2 element sequence

$$X_1 X_2 \ldots X_{n^2}$$

of n-bit numbers by the iterative process

$$X_{i+1} = (aX_i + b) \bmod m.$$

SIMPLE PSEUDORANDOM GENERATORS

Informally, a **pseudorandom generator** is a deterministic polynomial time algorithm which expands short random sequences (called **seeds**) into longer bit sequences such that the resulting probability distribution is in polynomial time indistinguishable from the uniform probability distribution.

Example. Linear congruential generator

One chooses n-bit numbers m, a, b, X_0 and generates an n^2 element sequence

$$X_1 X_2 \dots X_{n^2}$$

of n-bit numbers by the iterative process

$$X_{i+1} = (aX_i + b) \bmod m.$$

There is a variety of classical algorithms capable to generate pseudorandomness of different quality concerning randomness.

SIMPLE PSEUDORANDOM GENERATORS

Informally, a **pseudorandom generator** is a deterministic polynomial time algorithm which expands short random sequences (called **seeds**) into longer bit sequences such that the resulting probability distribution is in polynomial time indistinguishable from the uniform probability distribution.

Example. **Linear congruential generator**

One chooses n-bit numbers m, a, b, X_0 and generates an n^2 element sequence

$$X_1 X_2 \dots X_{n^2}$$

of n-bit numbers by the iterative process

$$X_{i+1} = (aX_i + b) \bmod m.$$

There is a variety of classical algorithms capable to generate pseudorandomness of different quality concerning randomness.

Quantum processes can generate perfect randomness and on this basis quantum (almost perfect) generators of randomness are already commercially available.

CRYPTOGRAPHICALY STRONG PSEUDORANDOM GENERATORS

In cryptography **random sequences** can usually be replaced by pseudorandom sequences generated by **(cryptographically perfect/strong) pseudorandom generators**.

CRYPTOGRAPHICALY STRONG PSEUDORANDOM GENERATORS

In cryptography **random sequences** can usually be replaced by pseudorandom sequences generated by **(cryptographically perfect/strong) pseudorandom generators**.

Definition. Let $l(n) : N \rightarrow N$ be such that $l(n) > n$ for all n. A **(cryptographically strong) pseudorandom generator with a stretch function** l, is an efficient deterministic algorithm which on the input of a random n-bit seed outputs a $l(n)$-bit sequence which is computationally indistinguishable from any random $l(n)$-bit sequence.

CRYPTOGRAPHICALY STRONG PSEUDORANDOM GENERATORS

In cryptography **random sequences** can usually be replaced by pseudorandom sequences generated by **(cryptographically perfect/strong) pseudorandom generators**.

Definition. Let $l(n) : N \rightarrow N$ be such that $l(n) > n$ for all n. A **(cryptographically strong) pseudorandom generator with a stretch function** l, is an efficient deterministic algorithm which on the input of a random n-bit seed outputs a $l(n)$-bit sequence which is computationally indistinguishable from any random $l(n)$-bit sequence.

Candidate for a cryptographically strong pseudorandom generator:

CRYPTOGRAPHICALY STRONG PSEUDORANDOM GENERATORS

In cryptography **random sequences** can usually be replaced by pseudorandom sequences generated by **(cryptographically perfect/strong) pseudorandom generators**.

Definition. Let $l(n) : N \rightarrow N$ be such that $l(n) > n$ for all n. A **(cryptographically strong) pseudorandom generator with a stretch function** l, is an efficient deterministic algorithm which on the input of a random n-bit seed outputs a $l(n)$-bit sequence which is computationally indistinguishable from any random $l(n)$-bit sequence.

Candidate for a cryptographically strong pseudorandom generator:

A very fundamental concept: A predicate b is a hard core predicate of the function f if b is easy to evaluate, but $b(x)$ is hard to predict from f(x). (That is, it is unfeasible, given f(x) where x is uniformly chosen, to predict $b(x)$ substantially better than with the probability $1/2$.)

CRYPTOGRAPHICALY STRONG PSEUDORANDOM GENERATORS

In cryptography **random sequences** can usually be replaced by pseudorandom sequences generated by **(cryptographically perfect/strong) pseudorandom generators**.

Definition. Let $l(n) : N \rightarrow N$ be such that $l(n) > n$ for all n. A **(cryptographically strong) pseudorandom generator with a stretch function** l, is an efficient deterministic algorithm which on the input of a random n-bit seed outputs a $l(n)$-bit sequence which is computationally indistinguishable from any random $l(n)$-bit sequence.

Candidate for a cryptographically strong pseudorandom generator:

A very fundamental concept: A predicate b is a hard core predicate of the function f if b is easy to evaluate, but $b(x)$ is hard to predict from f(x). (That is, it is unfeasible, given f(x) where x is uniformly chosen, to predict $b(x)$ substantially better than with the probability $1/2$.)

Conjecture: The least significant bit of x^2 **mod** n is a hard-core predicate.

CRYPTOGRAPHICALY STRONG PSEUDORANDOM GENERATORS

In cryptography **random sequences** can usually be replaced by pseudorandom sequences generated by **(cryptographically perfect/strong) pseudorandom generators**.

Definition. Let $l(n) : N \rightarrow N$ be such that $l(n) > n$ for all n. A **(cryptographically strong) pseudorandom generator with a stretch function** l, is an efficient deterministic algorithm which on the input of a random n-bit seed outputs a $l(n)$-bit sequence which is computationally indistinguishable from any random $l(n)$-bit sequence.

Candidate for a cryptographically strong pseudorandom generator:

A very fundamental concept: A predicate b is a hard core predicate of the function f if b is easy to evaluate, but $b(x)$ is hard to predict from f(x). (That is, it is unfeasible, given f(x) where x is uniformly chosen, to predict $b(x)$ substantially better than with the probability $1/2$.)

Conjecture: The least significant bit of x^2 **mod** n is a hard-core predicate.

Theorem Let f be a one-way function which is length preserving and efficiently computable, and b be a hard core predicate of f, then

$$G(s) = b(s) \cdot b(f(s)) \cdots b\left(f^{l(|s|)-1}(s)\right)$$

is a (cryptographically strong) pseudorandom generator with stretch function $l(n)$.

EXAMPLES
of
RANDOMIZED ALGORITHMS

EXAMPLE 1. MONOPOLIST GAME

EXAMPLE 1. MONOPOLIST GAME

Game Given are n active players each having w one dollar coins. They play, in rounds, the following game until all,

EXAMPLE 1. MONOPOLIST GAME

Game Given are n active players each having w one dollar coins. They play, in rounds, the following game until all, but one player,

EXAMPLE 1. MONOPOLIST GAME

Game Given are n active players each having w one dollar coins. They play, in rounds, the following game until all, but one player, become bankrupt:

EXAMPLE 1. MONOPOLIST GAME

Game Given are n active players each having w one dollar coins. They play, in rounds, the following game until all, but one player, become bankrupt:

1. In each round every active player puts $1 on the table and the roulette wheel is spined to determine the winner,

EXAMPLE 1. MONOPOLIST GAME

Game Given are n active players each having w one dollar coins. They play, in rounds, the following game until all, but one player, become bankrupt:

1. In each round every active player puts $1 on the table and the roulette wheel is spined to determine the winner, who then takes all money on the table.

EXAMPLE 1. MONOPOLIST GAME

Game Given are n active players each having w one dollar coins. They play, in rounds, the following game until all, but one player, become bankrupt:

1. In each round every active player puts $1 on the table and the roulette wheel is spined to determine the winner, who then takes all money on the table.
2. A player who looses all his money declares bankruptcy and becomes inactive.

EXAMPLE 1. MONOPOLIST GAME

Game Given are n active players each having w one dollar coins. They play, in rounds, the following game until all, but one player, become bankrupt:

1. In each round every active player puts $1 on the table and the roulette wheel is spined to determine the winner, who then takes all money on the table.
2. A player who looses all his money declares bankruptcy and becomes inactive.

Will the game end? If not, why? If yes, when?

EXAMPLE 1. MONOPOLIST GAME - again

EXAMPLE 1. MONOPOLIST GAME - again

Game Given are n active players each having w one dollar coins. They play, in rounds, the following game until all but one player become bankrupt:

1. In each round every active player puts $1 on the table and the roulette wheel is spined to determine the winner who then takes all money on the table.
2. A player who looses all his money declares bankruptcy and becomes inactive.

EXAMPLE 1. MONOPOLIST GAME - again

Game Given are n active players each having w one dollar coins. They play, in rounds, the following game until all but one player become bankrupt:

1. In each round every active player puts $1 on the table and the roulette wheel is spined to determine the winner who then takes all money on the table.
2. A player who looses all his money declares bankruptcy and becomes inactive.

Will the game end?

EXAMPLE 1. MONOPOLIST GAME - again

Game Given are n active players each having w one dollar coins. They play, in rounds, the following game until all but one player become bankrupt:

1. In each round every active player puts \$1 on the table and the roulette wheel is spined to determine the winner who then takes all money on the table.
2. A player who looses all his money declares bankruptcy and becomes inactive.

Will the game end? It can be shown that it ends almost always in approximately at most $(nw)^2$ steps.

EXAMPLE 2 - ELECTION of a LEADER

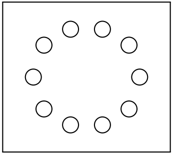
EXAMPLE 2 - ELECTION of a LEADER

In some cases randomization is the only way to solve the problem.

EXAMPLE 2 - ELECTION of a LEADER

In some cases randomization is the only way to solve the problem.

Example Let n identical processors, connected into a ring, have to choose one of them to be a "leader", under the assumption that each of the processors knows n.



EXAMPLE 2 - ELECTION of a LEADER - I.

EXAMPLE 2 - ELECTION of a LEADER - I.

Algorithm (Election of a leader - a symmetry breaking protocol)

EXAMPLE 2 - ELECTION of a LEADER - I.

Algorithm (Election of a leader - a symmetry breaking protocol)

1. Each processor sets its local variable V to n and starts to be *active*.

EXAMPLE 2 - ELECTION of a LEADER - I.

Algorithm (Election of a leader - a symmetry breaking protocol)

1. Each processor sets its local variable V to n and starts to be *active*.
2. Each active processor chooses, randomly and independently, an integer between 1 and V and put it into V.

EXAMPLE 2 - ELECTION of a LEADER - I.

Algorithm (Election of a leader - a symmetry breaking protocol)

1. Each processor sets its local variable V to n and starts to be *active*.
2. Each active processor chooses, randomly and independently, an integer between 1 and V and put it into V.
3. Those processors that choose 1 (if any), send one-bit message around the ring – clockwise - with the speed of one processor per time unit.

EXAMPLE 2 - ELECTION of a LEADER - I.

Algorithm (Election of a leader - a symmetry breaking protocol)

1. Each processor sets its local variable V to n and starts to be *active*.
2. Each active processor chooses, randomly and independently, an integer between 1 and V and put it into V.
3. Those processors that choose 1 (if any), send one-bit message around the ring – clockwise - with the speed of one processor per time unit.
4. After $n-1$ steps each processor knows the number l of processors that chosen 1. If $l=1$, the election ends and the leader introduces himself; if $l=0$, election continues by repeating Step 2.

EXAMPLE 2 - ELECTION of a LEADER - I.

Algorithm (Election of a leader - a symmetry breaking protocol)

1. Each processor sets its local variable V to n and starts to be *active*.
2. Each active processor chooses, randomly and independently, an integer between 1 and V and put it into V.
3. Those processors that choose 1 (if any), send one-bit message around the ring – clockwise - with the speed of one processor per time unit.
4. After $n-1$ steps each processor knows the number l of processors that chosen 1. If $l = 1$, the election ends and the leader introduces himself; if $l = 0$, election continues by repeating Step 2. If $l > 1$, the only processors remaining active will be those that have chosen 1 in Step 2. They set $V \leftarrow l$ and election continues with Step 2.

CLASSICAL versus QUANTUM RANDOMIZATION

CLASSICAL versus QUANTUM RANDOMIZATION

- Exact solvability of the leader election problem for regular graphs with identical node-processors is a celebrated unsolvable problem of classical distributed computing.

CLASSICAL versus QUANTUM RANDOMIZATION

- Exact solvability of the leader election problem for regular graphs with identical node-processors is a celebrated unsolvable problem of classical distributed computing.
- It can be shown that this problem cannot be solved exactly and in bounded time on classical computers even in the case processors know number of nodes (n) and topology of the network.

CLASSICAL versus QUANTUM RANDOMIZATION

- Exact solvability of the leader election problem for regular graphs with identical node-processors is a celebrated unsolvable problem of classical distributed computing.
- It can be shown that this problem cannot be solved exactly and in bounded time on classical computers even in the case processors know number of nodes (n) and topology of the network.
- However, there is quantum algorithm that runs in $\mathcal{O}(n^3)$ time, its communication complexity is $\mathcal{O}(n^4)$, and it can solve this problem exactly for any network topology, provided parties are connected by quantum communication links.

THE DINING CRYPTOGRAPHERS PROBLEM

THE DINING CRYPTOGRAPHERS PROBLEM

- Three cryptographers have dinner at a round table of a 5-star restaurant.

THE DINING CRYPTOGRAPHERS PROBLEM

- Three cryptographers have dinner at a round table of a 5-star restaurant.
- Their waiter tells them that an arrangement has been made that bill will be paid anonymously - either by one of them, or by NSA.

THE DINING CRYPTOGRAPHERS PROBLEM

- Three cryptographers have dinner at a round table of a 5-star restaurant.
- Their waiter tells them that an arrangement has been made that bill will be paid anonymously - either by one of them, or by NSA.
- They respect each others right to make an anonymous payment, but they wonder if NSA has payed the dinner.

THE DINING CRYPTOGRAPHERS PROBLEM

- Three cryptographers have dinner at a round table of a 5-star restaurant.
- Their waiter tells them that an arrangement has been made that bill will be paid anonymously - either by one of them, or by NSA.
- They respect each others right to make an anonymous payment, but they wonder if NSA has payed the dinner.
- How should they proceed to learn whether one of them paid the bill without learning which on e - for other two?

DINNING CRYPTOGRAPHERS - SOLUTION

DINNING CRYPTOGRAPHERS - SOLUTION

- **Protocol**

DINNING CRYPTOGRAPHERS - SOLUTION

- **Protocol**
  - Each cryptographer flips a perfect coin between him and the cryptographer on his right, so that only two of them can see the outcome.

DINNING CRYPTOGRAPHERS - SOLUTION

- **Protocol**
  - Each cryptographer flips a perfect coin between him and the cryptographer on his right, so that only two of them can see the outcome.
  - Each cryptographer who did not pay dinner states aloud whether the two coins he see - the one he flipped and the one his right-hand neighbour flipped - fell on the same side or not.

DINNING CRYPTOGRAPHERS - SOLUTION

- **Protocol**
  - Each cryptographer flips a perfect coin between him and the cryptographer on his right, so that only two of them can see the outcome.
  - Each cryptographer who did not pay dinner states aloud whether the two coins he see - the one he flipped and the one his right-hand neighbour flipped - fell on the same side or not.
  - The cryptographer who paid the dinner states aloud the opposite what he sees.

DINNING CRYPTOGRAPHERS - SOLUTION

- **Protocol**
  - Each cryptographer flips a perfect coin between him and the cryptographer on his right, so that only two of them can see the outcome.
  - Each cryptographer who did not pay dinner states aloud whether the two coins he see - the one he flipped and the one his right-hand neighbour flipped - fell on the same side or not.
  - The cryptographer who paid the dinner states aloud the opposite what he sees.
- Correctness:

DINNING CRYPTOGRAPHERS - SOLUTION

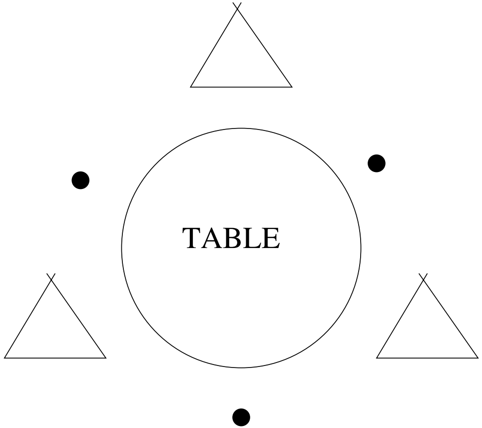
- **Protocol**
  - Each cryptographer flips a perfect coin between him and the cryptographer on his right, so that only two of them can see the outcome.
  - Each cryptographer who did not pay dinner states aloud whether the two coins he see - the one he flipped and the one his right-hand neighbour flipped - fell on the same side or not.
  - The cryptographer who paid the dinner states aloud the opposite what he sees.
- Correctness:
  - Odd number of differences uttered at the table implies that that a cryptographer paid the dinner.

DINNING CRYPTOGRAPHERS - SOLUTION

- **Protocol**
  - Each cryptographer flips a perfect coin between him and the cryptographer on his right, so that only two of them can see the outcome.
  - Each cryptographer who did not pay dinner states aloud whether the two coins he see - the one he flipped and the one his right-hand neighbour flipped - fell on the same side or not.
  - The cryptographer who paid the dinner states aloud the opposite what he sees.
- Correctness:
  - Odd number of differences uttered at the table implies that that a cryptographer paid the dinner.
  - Even number of differences uttered at the table implies that NSA paid the dinner.

DINNING CRYPTOGRAPHERS - SOLUTION

- **Protocol**
  - Each cryptographer flips a perfect coin between him and the cryptographer on his right, so that only two of them can see the outcome.
  - Each cryptographer who did not pay dinner states aloud whether the two coins he see - the one he flipped and the one his right-hand neighbour flipped - fell on the same side or not.
  - The cryptographer who paid the dinner states aloud the opposite what he sees.
- Correctness:
  - Odd number of differences uttered at the table implies that that a cryptographer paid the dinner.
  - Even number of differences uttered at the table implies that NSA paid the dinner.
  - In a case a cryptographer paid the dinner the other two cryptographers would have no idea he did that.



TABLE

TECHNICAL SOLUTION

Let three coin tossing made by cryptographers be represented by bits

$$b_1, b_2, b_3$$

TECHNICAL SOLUTION

Let three coin tossing made by cryptographers be represented by bits

$$b_1, b_2, b_3$$

In case none of them payed dinner, then what they say loudly are values

$$b_1 \oplus b_2, b_2 \oplus b_3, b_3 \oplus b_1$$

TECHNICAL SOLUTION

Let three coin tossing made by cryptographers be represented by bits

$$b_1, b_2, b_3$$

In case none of them payed dinner, then what they say loudly are values

$$b_1 \oplus b_2, b_2 \oplus b_3, b_3 \oplus b_1$$

and their parity is

TECHNICAL SOLUTION

Let three coin tossing made by cryptographers be represented by bits

$$b_1, b_2, b_3$$

In case none of them payed dinner, then what they say loudly are values

$$b_1 \oplus b_2, b_2 \oplus b_3, b_3 \oplus b_1$$

and their parity is

$$(b_1 \oplus b_2) \oplus (b_2 \oplus b_3) \oplus (b_3 \oplus b_1) = 0$$

TECHNICAL SOLUTION

Let three coin tossing made by cryptographers be represented by bits

$$b_1, b_2, b_3$$

In case none of them payed dinner, then what they say loudly are values

$$b_1 \oplus b_2, b_2 \oplus b_3, b_3 \oplus b_1$$

and their parity is

$$(b_1 \oplus b_2) \oplus (b_2 \oplus b_3) \oplus (b_3 \oplus b_1) = 0$$

In case one of them payed dinner, say Cryptographer 2, they say loudly:

$$b_1 \oplus b_2, \overline{b_2 \oplus b_3}, b_3 \oplus b_1$$

and

$$(b_1 \oplus b_2) \quad \oplus \quad (\overline{b_2 \oplus b_3}) \quad \oplus \quad (b_3 \oplus b_1) = 1$$

EXAMPLE: RANDOM COUNTING

EXAMPLE: RANDOM COUNTING

Problem: Determine the number, say n, of elements of a bag X, provided you can do, repeatedly, only the following operation: to pick up, randomly, an element of the bag X, to look at it, and to return it back to the bag.

EXAMPLE: RANDOM COUNTING

Problem: Determine the number, say n, of elements of a bag X, provided you can do, repeatedly, only the following operation: to pick up, randomly, an element of the bag X, to look at it, and to return it back to the bag.

Algorithm:
$k \leftarrow 0$;
do choose randomly an element from X, mark it and return it back; set $k \leftarrow k + 1$
until the just chosen element has already been chosen;

EXAMPLE: RANDOM COUNTING

Problem: Determine the number, say n, of elements of a bag X, provided you can do, repeatedly, only the following operation: to pick up, randomly, an element of the bag X, to look at it, and to return it back to the bag.

Algorithm:
$k \leftarrow 0$;
do choose randomly an element from X, mark it and return it back; set $k \leftarrow k + 1$
until the just chosen element has already been chosen;
$n \leftarrow \left\lfloor \frac{2k^2}{\pi} \right\rfloor$

EXAMPLE: ZERO POLYNOMIAL TESTING

EXAMPLE: ZERO POLYNOMIAL TESTING

Problem: Decide whether a given polynomial $p(x_1, \ldots, x_n)$, (given implicitly) with integer coefficients, and with each product of variables being of the degree at most k, is identically 0.

EXAMPLE: ZERO POLYNOMIAL TESTING

Problem: Decide whether a given polynomial $p(x_1, \ldots, x_n)$, (given implicitly) with integer coefficients, and with each product of variables being of the degree at most k, is identically 0.

Algorithm:
Compute $p(x_1, \ldots, x_n)$ N times, for sufficiently large N; each time with randomly chosen integer values for $x_1, \ldots, x_n$ from the interval $[0, 2kn]$.

EXAMPLE: ZERO POLYNOMIAL TESTING

Problem: Decide whether a given polynomial $p(x_1, \ldots, x_n)$, (given implicitly) with integer coefficients, and with each product of variables being of the degree at most k, is identically 0.

Algorithm:
Compute $p(x_1, \ldots, x_n)$ N times, for sufficiently large N; each time with randomly chosen integer values for $x_1, \ldots, x_n$ from the interval $[0, 2kn]$.

If, at the above process at least once a value different from 0 is obtained, then p is not identically 0.

EXAMPLE: ZERO POLYNOMIAL TESTING

Problem: Decide whether a given polynomial $p(x_1, \ldots, x_n)$, (given implicitly) with integer coefficients, and with each product of variables being of the degree at most k, is identically 0.

Algorithm:
Compute $p(x_1, \ldots, x_n)$ N times, for sufficiently large N; each time with randomly chosen integer values for $x_1, \ldots, x_n$ from the interval $[0, 2kn]$.

If, at the above process at least once a value different from 0 is obtained, then p is not identically 0.

If all N values obtained are 0, then we can consider p to be identically 0. The probability of error is at most 2^{-N}.

DESIGN of the SMALLEST ENCLOSING DISK

DESIGN of the SMALLEST ENCLOSING DISK

Task: Given is a set S of n points in the plane. Find the smallest disk (circle) $D(S)$ containing S.

DESIGN of the SMALLEST ENCLOSING DISK

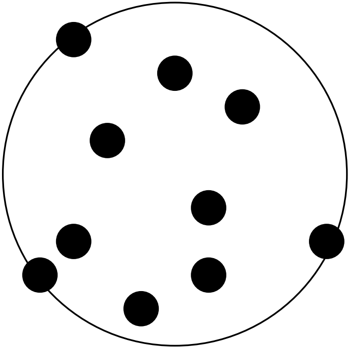
Task: Given is a set S of n points in the plane. Find the smallest disk (circle) $D(S)$ containing S.

Note $D(S)$ is determined by any three points on its edge.

DESIGN of the SMALLEST ENCLOSING DISK

Task: Given is a set S of n points in the plane. Find the smallest disk (circle) $D(S)$ containing S.

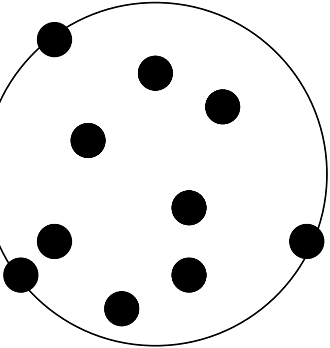
Note $D(S)$ is determined by any three points on its edge.



DESIGN of the SMALLEST ENCLOSING DISK

Task: Given is a set S of n points in the plane. Find the smallest disk (circle) $D(S)$ containing S.

Note $D(S)$ is determined by any three points on its edge.



Naive solution For any three points design a disk/circle passing through them - complexity of such an algorithm is $\mathcal{O}(n^3)$

Random $\mathcal{O}(n)$ algorithm - Welzl

Random $\mathcal{O}(n)$ algorithm - Welzl

For the start let us consider all points as having the weight 1

Algorithm

Random $\mathcal{O}(n)$ algorithm - Welzl

For the start let us consider all points as having the weight 1

Algorithm

1. Choose randomly (taking into considerations weights of points) a set of about 20 points S' and determine, somehow, $D(S')$.

Random $\mathcal{O}(n)$ algorithm - Welzl

For the start let us consider all points as having the weight 1

Algorithm

1. Choose randomly (taking into considerations weights of points) a set of about 20 points S' and determine, somehow, $D(S')$.
2. In case there are points of S that are out of $D(S')$,

Random $\mathcal{O}(n)$ algorithm - Welzl

For the start let us consider all points as having the weight 1

Algorithm

1. Choose randomly (taking into considerations weights of points) a set of about 20 points S' and determine, somehow, $D(S')$.
2. In case there are points of S that are out of $D(S')$, then double their weights and go to Step 1. Otherwise you are done

RANDOMIZED QUICKSORT

RANDOMIZED QUICKSORT

Problem: To sort a set S of n elements we can use the following algorithm.

RANDOMIZED QUICKSORT

Problem: To sort a set S of n elements we can use the following algorithm.

1. Choose a median y of S.

RANDOMIZED QUICKSORT

Problem: To sort a set S of n elements we can use the following algorithm.

1. Choose a median y of S.
2. Compare all elements of S with y and divide S into the set S_1 of elements smaller than y and into the set S_2 of the remaining elements.

RANDOMIZED QUICKSORT

Problem: To sort a set S of n elements we can use the following algorithm.

1. Choose a median y of S.
2. Compare all elements of S with y and divide S into the set S_1 of elements smaller than y and into the set S_2 of the remaining elements.
3. Sort recursively sets S_1 and S_2.

RANDOMIZED QUICKSORT

Problem: To sort a set S of n elements we can use the following algorithm.

1. Choose a median y of S.
2. Compare all elements of S with y and divide S into the set S_1 of elements smaller than y and into the set S_2 of the remaining elements.
3. Sort recursively sets S_1 and S_2.

Analysis of the number of comparisons: $T(n)$

$T(n) \leq 2T(\frac{n}{2}) + (c+1)n$

in case we can find y in cn steps for some constant c

RANDOMIZED QUICKSORT

Problem: To sort a set S of n elements we can use the following algorithm.

1. Choose a median y of S.
2. Compare all elements of S with y and divide S into the set S_1 of elements smaller than y and into the set S_2 of the remaining elements.
3. Sort recursively sets S_1 and S_2.

Analysis of the number of comparisons: $T(n)$

$T(n) \leq 2T(\frac{n}{2}) + (c+1)n$

in case we can find y in cn steps for some constant c

Solution of the above inequality:

$T(n) \leq c'n\lg n$

RANDOMIZED QUICKSORT

Problem: To sort a set S of n elements we can use the following algorithm.

1. Choose a median y of S.
2. Compare all elements of S with y and divide S into the set S_1 of elements smaller than y and into the set S_2 of the remaining elements.
3. Sort recursively sets S_1 and S_2.

Analysis of the number of comparisons: $T(n)$

$T(n) \leq 2T(\frac{n}{2}) + (c+1)n$

in case we can find y in cn steps for some constant c

Solution of the above inequality:

$T(n) \leq c'n \lg n$

Asymptotically, the same solution is obtained if we require only that none of the sets S_1, S_2 has more than $\frac{3}{4}n$ elements.

RANDOMIZED QUICKSORT

Problem: To sort a set S of n elements we can use the following algorithm.

1. Choose a median y of S.
2. Compare all elements of S with y and divide S into the set S_1 of elements smaller than y and into the set S_2 of the remaining elements.
3. Sort recursively sets S_1 and S_2.

Analysis of the number of comparisons: $T(n)$

$$T(n) \leq 2T(\tfrac{n}{2}) + (c+1)n$$

in case we can find y in cn steps for some constant c

Solution of the above inequality:

$$T(n) \leq c' n \lg n$$

Asymptotically, the same solution is obtained if we require only that none of the sets S_1, S_2 has more than $\frac{3}{4}n$ elements. Since there are at least $\frac{n}{2}$ elements y with the last property there is a good chance that if y is always chosen randomly, then we get a good performance.

RANDOMIZED QUICKSORT

Problem: To sort a set S of n elements we can use the following algorithm.

1. Choose a median y of S.
2. Compare all elements of S with y and divide S into the set S_1 of elements smaller than y and into the set S_2 of the remaining elements.
3. Sort recursively sets S_1 and S_2.

Analysis of the number of comparisons: $T(n)$

$T(n) \leq 2T(\frac{n}{2}) + (c+1)n$

in case we can find y in cn steps for some constant c

Solution of the above inequality:

$T(n) \leq c' n \lg n$

Asymptotically, the same solution is obtained if we require only that none of the sets S_1, S_2 has more than $\frac{3}{4}n$ elements. Since there are at least $\frac{n}{2}$ elements y with

the last property there is a good chance that if y is always chosen randomly, then we get a good performance.

This way we obtain *random QUICKSORT* or RQUICKSORT.

ANALYSIS of RQUICKSORT (RQS)

Let the set S to be sorted be given and let

ANALYSIS of RQUICKSORT (RQS)

Let the set S to be sorted be given and let

s_i – be the i-th smallest element of S;

ANALYSIS of RQUICKSORT (RQS)

Let the set S to be sorted be given and let

s_i – be the i-th smallest element of S;

s_{ij} – be a random variable having value 1 if s_i and s_j are being compared (during an execution of the RQS).

ANALYSIS of RQUICKSORT (RQS)

Let the set S to be sorted be given and let

s_i – be the i-th smallest element of S;

s_{ij} – be a random variable having value 1 if s_i and s_j are being compared (during an execution of the RQS).

Expected number of comparisons of RQS

$$E\left[\sum_{i=1}^{n}\sum_{j=1}^{n} s_{ij}\right] = \sum_{i=1}^{n}\sum_{j=1}^{n} E[s_{ij}]$$

ANALYSIS of RQUICKSORT (RQS)

Let the set S to be sorted be given and let

s_i – be the i-th smallest element of S;

s_{ij} – be a random variable having value 1 if s_i and s_j are being compared (during an execution of the RQS).

Expected number of comparisons of RQS

$$E\left[\sum_{i=1}^{n}\sum_{j=1}^{n} s_{ij}\right] = \sum_{i=1}^{n}\sum_{j=1}^{n} E[s_{ij}]$$

If p_{ij} is the probability that s_i and s_j are being compared during an execution of the algorithm, then $E[s_{ij}] = p_{ij}$.

In order to estimate p_{ij} it is enough to realize that if s_i and s_j are compared during an execution of the RQS, then one of these two elements has to be in the subtree headed by the other element in the comparison tree being created at that execution.

In order to estimate p_{ij} it is enough to realize that if s_i and s_j are compared during an execution of the RQS, then one of these two elements has to be in the subtree headed by the other element in the comparison tree being created at that execution. Moreover, in such a case all elements between s_i and s_j are still to be inserted into the tree being created. Therefore, at the moment other element (not the one in the root of the subtree), is chosen, it is chosen randomly from at least $|j - i| + 1$ elements.

In order to estimate p_{ij} it is enough to realize that if s_i and s_j are compared during an execution of the RQS, then one of these two elements has to be in the subtree headed by the other element in the comparison tree being created at that execution. Moreover, in such a case all elements between s_i and s_j are still to be inserted into the tree being created. Therefore, at the moment other element (not the one in the root of the subtree), is chosen, it is chosen randomly from at least $|j - i| + 1$ elements. Hence $p_{ij} \leq \frac{1}{|i-j|+1}$.

In order to estimate p_{ij} it is enough to realize that if s_i and s_j are compared during an execution of the RQS, then one of these two elements has to be in the subtree headed by the other element in the comparison tree being created at that execution. Moreover, in such a case all elements between s_i and s_j are still to be inserted into the tree being created. Therefore, at the moment other element (not the one in the root of the subtree), is chosen, it is chosen randomly from at least $|j-i|+1$ elements. Hence $p_{ij} \leq \frac{1}{|i-j|+1}$. Therefore we have (for $H_n = \sum_{i=1}^{n} \frac{1}{i}$):

$$\sum_{i=1}^{n} \sum_{j=1}^{n} p_{ij} \leq \sum_{i=1}^{n} \sum_{j=i}^{n} \frac{2}{j-i+1} \leq \sum_{i=1}^{n} \sum_{k=1}^{n-i+1} \frac{2}{k} \leq$$

$$2 \sum_{i=1}^{n} \sum_{k=1}^{n-i+1} \frac{1}{k} \leq 2nH_n = \Theta(n \log n)$$

SATISIFIABILITY of BOOLEAN FORMULAS

SATISIFIABILITY of BOOLEAN FORMULAS

The following algorithm finds, given a satisfiable Boolean formula F in 3-CNF, with very high probability, a satisfying assignment for F.

SATISIFIABILITY of BOOLEAN FORMULAS

The following algorithm finds, given a satisfiable Boolean formula F in 3-CNF, with very high probability, a satisfying assignment for F.

Algorithm:

SATISIFIABILITY of BOOLEAN FORMULAS

The following algorithm finds, given a satisfiable Boolean formula F in 3-CNF, with very high probability, a satisfying assignment for F.

Algorithm:
Choose randomly a truth assignment T for F;

SATISIFIABILITY of BOOLEAN FORMULAS

The following algorithm finds, given a satisfiable Boolean formula F in 3-CNF, with very high probability, a satisfying assignment for F.

Algorithm:

Choose randomly a truth assignment T for F;
while there is a truth assignment T' that differs from T in
  exactly one variable and satisfies more clauses of F than T
  do choose such of these T' that satisfy the most clauses and set $T \leftarrow T'$ **od**;

SATISIFIABILITY of BOOLEAN FORMULAS

The following algorithm finds, given a satisfiable Boolean formula F in 3-CNF, with very high probability, a satisfying assignment for F.

Algorithm:
Choose randomly a truth assignment T for F;
while there is a truth assignment T' that differs from T in
exactly one variable and satisfies more clauses of F than T
do choose such of these T' that satisfy the most clauses and set $T \leftarrow T'$ **od**;
return T

SATISIFIABILITY of BOOLEAN FORMULAS

The following algorithm finds, given a satisfiable Boolean formula F in 3-CNF, with very high probability, a satisfying assignment for F.

Algorithm:

Choose randomly a truth assignment T for F;
while there is a truth assignment T' that differs from T in
exactly one variable and satisfies more clauses of F than T
do choose such of these T' that satisfy the most clauses and set $T \leftarrow T'$ **od**;
return T

A natural question: How good is this simple algorithm?

SATISIFIABILITY of BOOLEAN FORMULAS

The following algorithm finds, given a satisfiable Boolean formula F in 3-CNF, with very high probability, a satisfying assignment for F.

Algorithm:
Choose randomly a truth assignment T for F;
while there is a truth assignment T' that differs from T in
exactly one variable and satisfies more clauses of F than T
do choose such of these T' that satisfy the most clauses and set $T \leftarrow T'$ **od**;
return T

A natural question: How good is this simple algorithm?

Theorem If $0 < \epsilon < \frac{1}{2}$, then there is a constant c such that for all but a fraction of at most $n2^n e^{-\frac{\epsilon n^2}{2}}$ of satisfiable 3-CNF Boolean formulas with n variables, the probability that the above algorithm succeeds in discovering a truth assignment in each independent trial from a random start is at least $1 - e^{-\epsilon^2 n}$.

EXAMPLE: CUTS in MULTIGRAPHS - PROBLEM

EXAMPLE: CUTS in MULTIGRAPHS - PROBLEM

Given is an undirected and loop-free multigraph G.

EXAMPLE: CUTS in MULTIGRAPHS - PROBLEM

Given is an undirected and loop-free multigraph G.The task is to find one of the smallest sets C of edges (called a cut) of G such that the removal of edges from C disconnects the multigraph G.

EXAMPLE: CUTS in MULTIGRAPHS - PROBLEM

Given is an undirected and loop-free multigraph G.The task is to find one of the smallest sets C of edges (called a cut) of G such that the removal of edges from C disconnects the multigraph G.

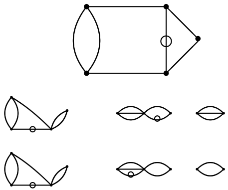
Basic operation is an edge contraction If e is an edge of a loop-free multigraph G, then the multigraph G/e is obtained from G by contracting the edge $e = \{x, y\}$, that is, we identify the vertices x and y and remove all resulting loops.

EXAMPLE: CUTS in MULTIGRAPHS - PROBLEM

Given is an undirected and loop-free multigraph G.The task is to find one of the smallest sets C of edges (called a cut) of G such that the removal of edges from C disconnects the multigraph G.

Basic operation is an edge contraction If e is an edge of a loop-free multigraph G, then the multigraph G/e is obtained from G by contracting the edge $e = \{x, y\}$, that is, we identify the vertices x and y and remove all resulting loops.

Example:



CUTS in MULTIGRAPHS - ALGORITHM

CUTS in MULTIGRAPHS - ALGORITHM

Basic idea of the algorithm given below: An edge contraction of a multigraph does not reduce the size of the minimal cut.

CUTS in MULTIGRAPHS - ALGORITHM

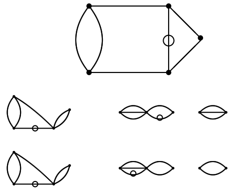
Basic idea of the algorithm given below: An edge contraction of a multigraph does not reduce the size of the minimal cut.

Contract algorithm:

while there are more than 2 vertices in the multigraph
    do edge-contraction of a randomly chosen edge **od**

Output the size of the minimal cut of the resulting 2 vertices multigraph.

Example:



CUTS in MULTIGRAPHS - ALGORITHM

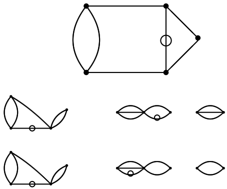
Basic idea of the algorithm given below: An edge contraction of a multigraph does not reduce the size of the minimal cut.

Contract algorithm:

while there are more than 2 vertices in the multigraph
    do edge-contraction of a randomly chosen edge **od**

Output the size of the minimal cut of the resulting 2 vertices multigraph.

Example:



In the above example, where two options are explored in the second step, we got once the optimal result, and once a non-optimal result.

HOW GOOD is the ABOVE ALGORITHM?

HOW GOOD is the ABOVE ALGORITHM?

How probable is that our algorithm produces an incorrect result?

HOW GOOD is the ABOVE ALGORITHM?

How probable is that our algorithm produces an incorrect result?

Let G be a multigraph with n vertices and k be the size of its minimal cut;

HOW GOOD is the ABOVE ALGORITHM?

How probable is that our algorithm produces an incorrect result?

Let G be a multigraph with n vertices and k be the size of its minimal cut;
C - be a particular minimal cut of size k.

HOW GOOD is the ABOVE ALGORITHM?

How probable is that our algorithm produces an incorrect result?

Let G be a multigraph with n vertices and k be the size of its minimal cut;
C - be a particular minimal cut of size k.

Observation: G has to have at least $\frac{kn}{2}$ edges. (Why?)

HOW GOOD is the ABOVE ALGORITHM?

How probable is that our algorithm produces an incorrect result?

Let G be a multigraph with n vertices and k be the size of its minimal cut;
C - be a particular minimal cut of size k.
Observation: G has to have at least $\frac{kn}{2}$ edges. (Why?)

We derive a lower bound on the probability that no edge of C is ever contracted during an execution of the algorithm.

HOW GOOD is the ABOVE ALGORITHM?

How probable is that our algorithm produces an incorrect result?

Let G be a multigraph with n vertices and k be the size of its minimal cut;
C - be a particular minimal cut of size k.
Observation: G has to have at least $\frac{kn}{2}$ edges. (Why?)

We derive a lower bound on the probability that no edge of C is ever contracted during an execution of the algorithm.

Let E_i be the event of non-choosing an edge of C at the i-th step of the algorithm.

HOW GOOD is the ABOVE ALGORITHM?

How probable is that our algorithm produces an incorrect result?

Let G be a multigraph with n vertices and k be the size of its minimal cut;
C - be a particular minimal cut of size k.
Observation: G has to have at least $\frac{kn}{2}$ edges. (Why?)

We derive a lower bound on the probability that no edge of C is ever contracted during an execution of the algorithm.

Let E_i be the event of non-choosing an edge of C at the i-th step of the algorithm. The probability that the edge randomly chosen in the first step is in C is at most $\frac{k}{\frac{nk}{2}} = \frac{2}{n}$ and therefore $Pr(E_1) \geq 1 - \frac{2}{n}$.

HOW GOOD is the ABOVE ALGORITHM?

How probable is that our algorithm produces an incorrect result?

Let G be a multigraph with n vertices and k be the size of its minimal cut;
C - be a particular minimal cut of size k.
Observation: G has to have at least $\frac{kn}{2}$ edges. (Why?)

We derive a lower bound on the probability that no edge of C is ever contracted during an execution of the algorithm.

Let E_i be the event of non-choosing an edge of C at the i-th step of the algorithm. The probability that the edge randomly chosen in the first step is in C

is at most $\frac{k}{\frac{nk}{2}} = \frac{2}{n}$ and therefore $Pr(E_1) \geq 1 - \frac{2}{n}$.

If E_1 occurs, then at the second contraction step there are at least $\frac{k(n-1)}{2}$ edges.
Hence $Pr(E_2|E_1) \geq 1 - \frac{2}{n-1}$

HOW GOOD is the ABOVE ALGORITHM?

How probable is that our algorithm produces an incorrect result?

Let G be a multigraph with n vertices and k be the size of its minimal cut;
C - be a particular minimal cut of size k.
Observation: G has to have at least $\frac{kn}{2}$ edges. (Why?)

We derive a lower bound on the probability that no edge of C is ever contracted during an execution of the algorithm.

Let E_i be the event of non-choosing an edge of C at the i-th step of the algorithm. The probability that the edge randomly chosen in the first step is in C

is at most $\frac{k}{\frac{nk}{2}} = \frac{2}{n}$ and therefore $Pr(E_1) \geq 1 - \frac{2}{n}$.

If E_1 occurs, then at the second contraction step there are at least $\frac{k(n-1)}{2}$ edges.
Hence $Pr(E_2|E_1) \geq 1 - \frac{2}{n-1}$

Similarly, in the i-th step

$$Pr\left[E_i|\bigcap_{j=1}^{i-1} E_j\right] \geq 1 - \frac{2}{n-i+1} = \frac{n-i-1}{n-i+1}$$

PROOF CONTINUATION

PROOF CONTINUATION

Therefore, the probability that no edge of C is ever contracted during an execution of the algorithm, that is that algorithm gives correct output, can be lower bounded by

$$Pr\left[\bigcap_{i=1}^{n-2} E_i\right] \geq \prod_{i=1}^{n-2}\left(1-\frac{2}{n-i+1}\right) = \prod_{i=1}^{n-2}\left(\frac{n-i-1}{n-i+1}\right) = \frac{2}{n(n-1)} = \Omega(\frac{1}{n^2})$$

PROOF CONTINUATION

Therefore, the probability that no edge of C is ever contracted during an execution of the algorithm, that is that algorithm gives correct output, can be lower bounded by

$$Pr\left[\bigcap_{i=1}^{n-2} E_i\right] \geq \prod_{i=1}^{n-2}\left(1-\frac{2}{n-i+1}\right) = \prod_{i=1}^{n-2}\left(\frac{n-i-1}{n-i+1}\right) = \frac{2}{n(n-1)} = \Omega(\frac{1}{n^2})$$

Hence, the probability of an incorrect result is $\leq 1-\frac{2}{n(n-1)}$.

PROOF CONTINUATION

Therefore, the probability that no edge of C is ever contracted during an execution of the algorithm, that is that algorithm gives correct output, can be lower bounded by

$$Pr\left[\bigcap_{i=1}^{n-2} E_i\right] \geq \prod_{i=1}^{n-2}\left(1-\frac{2}{n-i+1}\right)=\prod_{i=1}^{n-2}\left(\frac{n-i-1}{n-i+1}\right)=\frac{2}{n(n-1)}=\Omega(\frac{1}{n^2})$$

Hence, the probability of an incorrect result is $\leq 1-\frac{2}{n(n-1)}$.

Moreover, if the above algorithm is repeated $\frac{n^2}{2}$ times, making each time random decisions, then the probability that a minimal cut is not found is at most

PROOF CONTINUATION

Therefore, the probability that no edge of C is ever contracted during an execution of the algorithm, that is that algorithm gives correct output, can be lower bounded by

$$Pr\left[\bigcap_{i=1}^{n-2} E_i\right] \geq \prod_{i=1}^{n-2}\left(1-\frac{2}{n-i+1}\right) = \prod_{i=1}^{n-2}\left(\frac{n-i-1}{n-i+1}\right) = \frac{2}{n(n-1)} = \Omega(\frac{1}{n^2})$$

Hence, the probability of an incorrect result is $\leq 1-\frac{2}{n(n-1)}$.

Moreover, if the above algorithm is repeated $\frac{n^2}{2}$ times, making each time random decisions, then the probability that a minimal cut is not found is at most

$$\left(1-\frac{2}{n^2-n}\right)^{\frac{n^2}{2}} < \left(1-\frac{2}{n^2}\right)^{\frac{n^2}{2}} = \left(1-\frac{1}{\frac{n^2}{2}}\right)^{\frac{n^2}{2}} < \frac{1}{e}$$

PROOF CONTINUATION

Therefore, the probability that no edge of C is ever contracted during an execution of the algorithm, that is that algorithm gives correct output, can be lower bounded by

$$Pr\left[\bigcap_{i=1}^{n-2} E_i\right] \geq \prod_{i=1}^{n-2}\left(1-\frac{2}{n-i+1}\right) = \prod_{i=1}^{n-2}\left(\frac{n-i-1}{n-i+1}\right) = \frac{2}{n(n-1)} = \Omega(\frac{1}{n^2})$$

Hence, the probability of an incorrect result is $\leq 1-\frac{2}{n(n-1)}$.

Moreover, if the above algorithm is repeated $\frac{n^2}{2}$ times, making each time random decisions, then the probability that a minimal cut is not found is at most

$$\left(1-\frac{2}{n^2-n}\right)^{\frac{n^2}{2}} < \left(1-\frac{2}{n^2}\right)^{\frac{n^2}{2}} = \left(1-\frac{1}{\frac{n^2}{2}}\right)^{\frac{n^2}{2}} < \frac{1}{e}$$

Running time of the best deterministic minimum cut algorithm is $\mathcal{O}(nm+n^2 \lg n)$, where m is number of edges and n is number of vertices.

REMINDERS

REMINDERS

The following facts are well-known from mathematical analysis:

REMINDERS

The following facts are well-known from mathematical analysis:

- $(1 + \frac{x}{n})^n \leq e^x$;

REMINDERS

The following facts are well-known from mathematical analysis:

- $(1 + \frac{x}{n})^n \leq e^x$;
- $\lim_{n \to \infty}(1 + \frac{x}{n})^n = e^x$

PRIMES RECOGNITION

PRIMES RECOGNITION

The fastest known sequential deterministic algorithm to decide whether a given integer n is prime has complexity $O\left((\lg n)^{14}\right)$

PRIMES RECOGNITION

The fastest known sequential deterministic algorithm to decide whether a given integer n is prime has complexity $O\left((\lg n)^{14}\right)$

A simple randomized Rabin-Miller's Monte Carlo algorithm for prime recognition is based on the following result from the number theory.

Lemma Let $n \in \mathbf{N}$, $n = 2^s d + 1, d$ is odd. Denote, for $1 \leq x < n$, by $C(x)$ the condition:

PRIMES RECOGNITION

The fastest known sequential deterministic algorithm to decide whether a given integer n is prime has complexity $O\left((\lg n)^{14}\right)$

A simple randomized Rabin-Miller's Monte Carlo algorithm for prime recognition is based on the following result from the number theory.

Lemma Let $n \in \mathbf{N}$, $n = 2^s d + 1, d$ is odd. Denote, for $1 \leq x < n$, by $C(x)$ the condition: $x^d \not\equiv 1 \pmod{n}$ **and** $x^{2^r d} \not\equiv -1 \pmod{n}$ **for all** $1 < r < s$

PRIMES RECOGNITION

The fastest known sequential deterministic algorithm to decide whether a given integer n is prime has complexity $O\left((\lg n)^{14}\right)$

A simple randomized Rabin-Miller's Monte Carlo algorithm for prime recognition is based on the following result from the number theory.

Lemma Let $n \in \mathbf{N}$, $n = 2^s d + 1, d$ is odd. Denote, for $1 \leq x < n$, by $C(x)$ the condition: $x^d \not\equiv 1 \pmod{n}$ **and** $x^{2^r d} \not\equiv -1 \pmod{n}$ **for all** $1 < r < s$ **Key fact:** If $C(x)$ holds for some $1 \leq x < n$, then n is not prime (and x is a witness for compositness of n). If n is not prime, then $C(x)$ holds for at least half of x between 1 and n.

In other words most of the numbers between 1 and n are witnesses for composability of n.

Rabin-Miller algorithm

PRIMES RECOGNITION

The fastest known sequential deterministic algorithm to decide whether a given integer n is prime has complexity $O\left((\lg n)^{14}\right)$

A simple randomized Rabin-Miller's Monte Carlo algorithm for prime recognition is based on the following result from the number theory.

Lemma Let $n \in \mathbf{N}$, $n = 2^s d + 1, d$ is odd. Denote, for $1 \leq x < n$, by $C(x)$ the condition: $x^d \not\equiv 1 \pmod{n}$ **and** $x^{2^r d} \not\equiv -1 \pmod{n}$ **for all** $1 < r < s$ **Key fact:** If $C(x)$ holds for some $1 \leq x < n$, then n is not prime (and x is a witness for compositness of n). If n is not prime, then $C(x)$ holds for at least half of x between 1 and n.

In other words most of the numbers between 1 and n are witnesses for composability of n.

Rabin-Miller algorithm

- Choose randomly integers $x_1, \ldots, x_m$ such that $1 \leq x_j < n$;

PRIMES RECOGNITION

The fastest known sequential deterministic algorithm to decide whether a given integer n is prime has complexity $O\left((\lg n)^{14}\right)$

A simple randomized Rabin-Miller's Monte Carlo algorithm for prime recognition is based on the following result from the number theory.

Lemma Let $n \in \mathbf{N}$, $n = 2^s d + 1, d$ is odd. Denote, for $1 \leq x < n$, by $C(x)$ the condition: $x^d \not\equiv 1 \pmod{n}$ **and** $x^{2^r d} \not\equiv -1 \pmod{n}$ **for all** $1 < r < s$ **Key fact:** If $C(x)$ holds for some $1 \leq x < n$, then n is not prime (and x is a witness for compositness of n). If n is not prime, then $C(x)$ holds for at least half of x between 1 and n.

In other words most of the numbers between 1 and n are witnesses for composability of n.

Rabin-Miller algorithm

- Choose randomly integers $x_1, \ldots, x_m$ such that $1 \leq x_j < n$;
- For each x_j determine whether $C(x_j)$ holds;

PRIMES RECOGNITION

The fastest known sequential deterministic algorithm to decide whether a given integer n is prime has complexity $O\left((\lg n)^{14}\right)$

A simple randomized Rabin-Miller's Monte Carlo algorithm for prime recognition is based on the following result from the number theory.

Lemma Let $n \in \mathbf{N}$, $n = 2^s d + 1, d$ is odd. Denote, for $1 \leq x < n$, by $C(x)$ the condition: $x^d \not\equiv 1 \pmod{n}$ **and** $x^{2^r d} \not\equiv -1 \pmod{n}$ **for all** $1 < r < s$ **Key fact:** If $C(x)$ holds for some $1 \leq x < n$, then n is not prime (and x is a witness for compositness of n). If n is not prime, then $C(x)$ holds for at least half of x between 1 and n.

In other words most of the numbers between 1 and n are witnesses for composability of n.

Rabin-Miller algorithm

- Choose randomly integers $x_1, \ldots, x_m$ such that $1 \leq x_j < n$;
- For each x_j determine whether $C(x_j)$ holds;
- **if** $C(x_j)$ holds for some x_j;

PRIMES RECOGNITION

The fastest known sequential deterministic algorithm to decide whether a given integer n is prime has complexity $O\left((\lg n)^{14}\right)$

A simple randomized Rabin-Miller's Monte Carlo algorithm for prime recognition is based on the following result from the number theory.

Lemma Let $n \in \mathbf{N}$, $n = 2^s d + 1, d$ is odd. Denote, for $1 \leq x < n$, by $C(x)$ the condition: $x^d \not\equiv 1 \pmod{n}$ **and** $x^{2^r d} \not\equiv -1 \pmod{n}$ **for all** $1 < r < s$ **Key fact:** If $C(x)$ holds for some $1 \leq x < n$, then n is not prime (and x is a witness for compositness of n). If n is not prime, then $C(x)$ holds for at least half of x between 1 and n.

In other words most of the numbers between 1 and n are witnesses for composability of n.

Rabin-Miller algorithm

- Choose randomly integers $x_1, \ldots, x_m$ such that $1 \leq x_j < n$;
- For each x_j determine whether $C(x_j)$ holds;
- **if** $C(x_j)$ holds for some x_j;

  then n is not prime

PRIMES RECOGNITION

The fastest known sequential deterministic algorithm to decide whether a given integer n is prime has complexity $O\left((\lg n)^{14}\right)$

A simple randomized Rabin-Miller's Monte Carlo algorithm for prime recognition is based on the following result from the number theory.

Lemma Let $n \in \mathbf{N}$, $n = 2^s d + 1, d$ is odd. Denote, for $1 \leq x < n$, by $C(x)$ the condition: $x^d \not\equiv 1 \pmod{n}$ **and** $x^{2^r d} \not\equiv -1 \pmod{n}$ **for all** $1 < r < s$ **Key fact:** If $C(x)$ holds for some $1 \leq x < n$, then n is not prime (and x is a witness for compositness of n). If n is not prime, then $C(x)$ holds for at least half of x between 1 and n.

In other words most of the numbers between 1 and n are witnesses for composability of n.

Rabin-Miller algorithm

- Choose randomly integers $x_1, \ldots, x_m$ such that $1 \leq x_j < n$;
- For each x_j determine whether $C(x_j)$ holds;
- **if** $C(x_j)$ holds for some x_j;

  then n is not prime

  else n is prime, with probability of error 2^{-m}

LARGEST PRIME - I.

On February 3, 2016 C. Cooper from university Missouri announced a new (Mersenne) prime

$$2^{74207181} - 1$$

that has 5 millions more digits as previously known largest prime.

LARGEST PRIME - II.

On December 29, 2017 people from the project GIMPS (Great Internet Mersenne Prime Search a new (Mersenne) prime

$$2^{77232917} - 1$$

announced that has 2 millions more digits as previously known largest prime. It has 23, 249,425 digits.

LARGEST PRIME - II.

On December 29, 2017 people from the project GIMPS (Great Internet Mersenne Prime Search a new (Mersenne) prime

$$2^{77232917} - 1$$

announced that has 2 millions more digits as previously known largest prime. It has 23, 249,425 digits.

Four research groups over the world verified after the announcement for three days that the number claimed to be a new largest prime is indeed a prime.

In 2008 a 100.000 $ price was given for first 10 millions digit primes.

In 2008 a 100.000 $ price was given for first 10 millions digit primes.

A special price is offered for first 100 millions of digits prime.

In 2008 a 100.000 $ price was given for first 10 millions digit primes.

A special price is offered for first 100 millions of digits prime.

Percentage of 512 bits numbers that are primes is 0.006...

RANDOMIZED COMPLEXITY CLASSES

COMPLEXITY CLASSES for DETERMINISTIC COMPUTATIONS

- **P** is the class of problems (languages) that can be solved (accepted) by deterministic algorithms running in polynomial time. (**Or P is class of problems solvable in polynomial time on deterministic Turing machines.**)
- **NP** is the class of problems solution of which can be verified in polynomial time. (**Or NP is the class of problems that can be solved in polynomial time on nondeterministic Turing machines.**)
- **co-NP** is the class of languages that are complements of languages in **NP**.
- **PSPACE** is the class of problems (languages) that can be solved (accepted) by algorithms using only polynomially large space/memory.
- **EXP** is the class of problems (languages) solvable in exponential time.

RANDOMIZED COMPLEXITY CLASSES

A way how to model random steps formally, and to study power of randomization, is to consider probabilistic algorithms as nondeterministic Turing machines (NTM), that have in each configuration exactly two choices to make and for each input all computations have the same length.

RANDOMIZED COMPLEXITY CLASSES

A way how to model random steps formally, and to study power of randomization, is to consider probabilistic algorithms as nondeterministic Turing machines (NTM), that have in each configuration exactly two choices to make and for each input all computations have the same length. In order to define different complexity classes for randomized computations, one then just needs to consider different acceptance modes.

RANDOMIZED COMPLEXITY CLASSES

A way how to model random steps formally, and to study power of randomization, is to consider probabilistic algorithms as nondeterministic Turing machines (NTM), that have in each configuration exactly two choices to make and for each input all computations have the same length. In order to define different complexity classes for randomized computations, one then just needs to consider different acceptance modes.

RP: A language L is in randomized complexity class **RP** (**R**andom **P**olynomial time) if there is a polynomial NTM such that:

RANDOMIZED COMPLEXITY CLASSES

A way how to model random steps formally, and to study power of randomization, is to consider probabilistic algorithms as nondeterministic Turing machines (NTM), that have in each configuration exactly two choices to make and for each input all computations have the same length. In order to define different complexity classes for randomized computations, one then just needs to consider different acceptance modes.

RP: A language L is in randomized complexity class **RP** (**R**andom **P**olynomial time) if there is a polynomial NTM such that:

- if $x \in L$, then *at least half* of all computations of M on x terminate in an accepting state;

RANDOMIZED COMPLEXITY CLASSES

A way how to model random steps formally, and to study power of randomization, is to consider probabilistic algorithms as nondeterministic Turing machines (NTM), that have in each configuration exactly two choices to make and for each input all computations have the same length. In order to define different complexity classes for randomized computations, one then just needs to consider different acceptance modes.

RP: A language L is in randomized complexity class **RP** (**R**andom **P**olynomial time) if there is a polynomial NTM such that:

- if $x \in L$, then *at least half* of all computations of M on x terminate in an accepting state;
- if $x \notin L$, then *all* computations of M terminate in rejecting states. (So called *Monte Carlo* acceptance or *one-sided Monte Carlo* acceptance).

RANDOMIZED COMPLEXITY CLASSES

A way how to model random steps formally, and to study power of randomization, is to consider probabilistic algorithms as nondeterministic Turing machines (NTM), that have in each configuration exactly two choices to make and for each input all computations have the same length. In order to define different complexity classes for randomized computations, one then just needs to consider different acceptance modes.

RP: A language L is in randomized complexity class **RP** (**R**andom **P**olynomial time) if there is a polynomial NTM such that:

- if $x \in L$, then *at least half* of all computations of M on x terminate in an accepting state;
- if $x \notin L$, then *all* computations of M terminate in rejecting states. (So called *Monte Carlo* acceptance or *one-sided Monte Carlo* acceptance).

ZPP: A language L is in **ZPP** (**Z**ero error **P**robabilistic **P**olynomial time) (it is also called *Las Vegas acceptance* if.) $L \in \mathbf{ZPP} = \mathbf{RP} \wedge \mathbf{coRP}$.

RANDOMIZED COMPLEXITY CLASSES

A way how to model random steps formally, and to study power of randomization, is to consider probabilistic algorithms as nondeterministic Turing machines (NTM), that have in each configuration exactly two choices to make and for each input all computations have the same length. In order to define different complexity classes for randomized computations, one then just needs to consider different acceptance modes.

RP: A language L is in randomized complexity class **RP** (**R**andom **P**olynomial time) if there is a polynomial NTM such that:

- if $x \in L$, then *at least half* of all computations of M on x terminate in an accepting state;
- if $x \notin L$, then *all* computations of M terminate in rejecting states. (So called *Monte Carlo* acceptance or *one-sided Monte Carlo* acceptance).

ZPP: A language L is in **ZPP** (**Z**ero error **P**robabilistic **P**olynomial time) (it is also called *Las Vegas acceptance* if.) $L \in \mathbf{ZPP} = \mathbf{RP} \wedge \mathbf{coRP}$.

PP: A language L is in **PP** (**P**robabilistic **P**olynomial time) if there is a polynomial NTM such that: $x \in L$ iff *more than half* of computations of M on x terminate in accepting states. (So called *acceptance by majority*.)

BPP and OTHER VIEW of COMPLEXITY CLASSES

BPP and OTHER VIEW of COMPLEXITY CLASSES

BPP: A language is in **BPP** (**B**ounded error away from $\frac{1}{2}$ **P**robabilistic **P**olynomial time), if there is a polynomial NTM M such that:

- If $x \in L$, then at least $\frac{3}{4}$ computations of M on x terminate in accepting states.
- If $x \notin L$, then at least $\frac{3}{4}$ of computations of M on x terminate in rejecting states.

BPP and OTHER VIEW of COMPLEXITY CLASSES

BPP: A language is in **BPP** (**B**ounded error away from $\frac{1}{2}$ **P**robabilistic **P**olynomial time), if there is a polynomial NTM M such that:

- If $x \in L$, then at least $\frac{3}{4}$ computations of M on x terminate in accepting states.
- If $x \notin L$, then at least $\frac{3}{4}$ of computations of M on x terminate in rejecting states.

Less formally, classes **RP**, **PP** and **BPP** can be defined as classes of problems (languages) for which there is a randomized algorithm A with the following property:

BPP and OTHER VIEW of COMPLEXITY CLASSES

BPP: A language is in **BPP** (**B**ounded error away from $\frac{1}{2}$ **P**robabilistic **P**olynomial time), if there is a polynomial NTM M such that:

- If $x \in L$, then at least $\frac{3}{4}$ computations of M on x terminate in accepting states.
- If $x \notin L$, then at least $\frac{3}{4}$ of computations of M on x terminate in rejecting states.

Less formally, classes **RP**, **PP** and **BPP** can be defined as classes of problems (languages) for which there is a randomized algorithm A with the following property:

- **RP:**
  - $x \in L \Rightarrow PR(A(x) \text{ accepts}) \geq \frac{1}{2}$;
  - $x \notin L \Rightarrow PR(A(x) \text{ accepts}) = 0$

BPP and OTHER VIEW of COMPLEXITY CLASSES

BPP: A language is in **BPP** (**B**ounded error away from $\frac{1}{2}$ **P**robabilistic **P**olynomial time), if there is a polynomial NTM M such that:

- If $x \in L$, then at least $\frac{3}{4}$ computations of M on x terminate in accepting states.
- If $x \notin L$, then at least $\frac{3}{4}$ of computations of M on x terminate in rejecting states.

Less formally, classes **RP**, **PP** and **BPP** can be defined as classes of problems (languages) for which there is a randomized algorithm A with the following property:

- **RP:**
  - $x \in L \Rightarrow PR(A(x) \text{ accepts}) \geq \frac{1}{2}$;
  - $x \notin L \Rightarrow PR(A(x) \text{ accepts}) = 0$
- **PP:**
  - $x \in L \Rightarrow PR(A(x) \text{ accepts}) > \frac{1}{2}$;
  - $x \notin L \Rightarrow PR(A(x) \text{ accepts}) \leq \frac{1}{2}$.

BPP and OTHER VIEW of COMPLEXITY CLASSES

BPP: A language is in **BPP** (**B**ounded error away from $\frac{1}{2}$ **P**robabilistic **P**olynomial time), if there is a polynomial NTM M such that:

- If $x \in L$, then at least $\frac{3}{4}$ computations of M on x terminate in accepting states.
- If $x \notin L$, then at least $\frac{3}{4}$ of computations of M on x terminate in rejecting states.

Less formally, classes **RP**, **PP** and **BPP** can be defined as classes of problems (languages) for which there is a randomized algorithm A with the following property:

- **RP:**
  - $x \in L \Rightarrow PR(A(x) \text{ accepts}) \geq \frac{1}{2}$;
  - $x \notin L \Rightarrow PR(A(x) \text{ accepts}) = 0$
- **PP:**
  - $x \in L \Rightarrow PR(A(x) \text{ accepts}) > \frac{1}{2}$;
  - $x \notin L \Rightarrow PR(A(x) \text{ accepts}) \leq \frac{1}{2}$.
- **BPP:**
  - $x \in L \Rightarrow PR(A(x) \text{ accepts}) \geq \frac{3}{4}$;
  - $x \notin L \Rightarrow PR(A(x) \text{ accepts}) \leq \frac{1}{4}$

PP class - some observations

PP class - some observations

- Definition of the class **PP** seems to be very natural. However, in reality this class is not realistic.

PP class - some observations

- Definition of the class **PP** seems to be very natural. However, in reality this class is not realistic.
- An example of a **PP** problem: Given a Boolean formula ϕ with n variables, do at least half of the 2^n possible assignments of variables make the formula to evaluate to TRUE?

PP class - some observations

- Definition of the class **PP** seems to be very natural. However, in reality this class is not realistic.
- An example of a **PP** problem: Given a Boolean formula ϕ with n variables, do at least half of the 2^n possible assignments of variables make the formula to evaluate to TRUE?
- Just like the problem to decide whether there exists a satisfying assignment for a Boolean formula is **NP**-complete, so this majority-vote variant of the above decision problem can be shown to be **PP**-complete;

PP class - some observations

- Definition of the class **PP** seems to be very natural. However, in reality this class is not realistic.
- An example of a **PP** problem: Given a Boolean formula ϕ with n variables, do at least half of the 2^n possible assignments of variables make the formula to evaluate to TRUE?
- Just like the problem to decide whether there exists a satisfying assignment for a Boolean formula is **NP**-complete, so this majority-vote variant of the above decision problem can be shown to be **PP**-complete; that is, any other **PP**-complete problem is efficiently reducible to it.

PP class - some observations

- Definition of the class **PP** seems to be very natural. However, in reality this class is not realistic.
- An example of a **PP** problem: Given a Boolean formula ϕ with n variables, do at least half of the 2^n possible assignments of variables make the formula to evaluate to TRUE?
- Just like the problem to decide whether there exists a satisfying assignment for a Boolean formula is **NP**-complete, so this majority-vote variant of the above decision problem can be shown to be **PP**-complete; that is, any other **PP**-complete problem is efficiently reducible to it.
- Problems: a **PP**-algorithm is free to accept with probability $1/2 + 2^{-n}$ if the answer is yes and probability $1/2 - 2^n$ if the answer is no. However how can a mortal distinguish these two cases if, for example, $n = 5000$?

INCLUSIONS between MAIN COMPLEXITY CLASSES

INCLUSIONS between MAIN COMPLEXITY CLASSES

Theorem

$$\mathbf{P} \subseteq \mathbf{ZPP} \subseteq \mathbf{RP} \subseteq \mathbf{NP} \subseteq \mathbf{PP} \subseteq \mathbf{PSPACE}$$

Proof: Since relations $\mathbf{P} \subseteq \mathbf{ZPP} \subseteq \mathbf{RP}$ are obvious, we show first

$$\mathbf{RP} \subseteq \mathbf{NP}$$

If $L \in \mathbf{RP}$ then there is a NTM M accepting L with Monte Carlo acceptance. Hence, $L \in \mathbf{NP}$.

INCLUSIONS between MAIN COMPLEXITY CLASSES

Theorem

$$\mathbf{P} \subseteq \mathbf{ZPP} \subseteq \mathbf{RP} \subseteq \mathbf{NP} \subseteq \mathbf{PP} \subseteq \mathbf{PSPACE}$$

Proof: Since relations $\mathbf{P} \subseteq \mathbf{ZPP} \subseteq \mathbf{RP}$ are obvious, we show first

$$\mathbf{RP} \subseteq \mathbf{NP}$$

If $L \in \mathbf{RP}$ then there is a NTM M accepting L with Monte Carlo acceptance. Hence, $L \in \mathbf{NP}$. Now we show:

$$\mathbf{NP} \subseteq \mathbf{PP}$$

INCLUSIONS between MAIN COMPLEXITY CLASSES

Theorem

$$\mathbf{P} \subseteq \mathbf{ZPP} \subseteq \mathbf{RP} \subseteq \mathbf{NP} \subseteq \mathbf{PP} \subseteq \mathbf{PSPACE}$$

Proof: Since relations $\mathbf{P} \subseteq \mathbf{ZPP} \subseteq \mathbf{RP}$ are obvious, we show first

$$\mathbf{RP} \subseteq \mathbf{NP}$$

If $L \in \mathbf{RP}$ then there is a NTM M accepting L with Monte Carlo acceptance. Hence, $L \in \mathbf{NP}$. Now we show:

$$\mathbf{NP} \subseteq \mathbf{PP}$$

Let $L \in \mathbf{NP}$ and M be a polynomial NTM for L.

INCLUSIONS between MAIN COMPLEXITY CLASSES

Theorem

$$\mathbf{P} \subseteq \mathbf{ZPP} \subseteq \mathbf{RP} \subseteq \mathbf{NP} \subseteq \mathbf{PP} \subseteq \mathbf{PSPACE}$$

Proof: Since relations $\mathbf{P} \subseteq \mathbf{ZPP} \subseteq \mathbf{RP}$ are obvious, we show first

$$\mathbf{RP} \subseteq \mathbf{NP}$$

If $L \in \mathbf{RP}$ then there is a NTM M accepting L with Monte Carlo acceptance. Hence, $L \in \mathbf{NP}$. Now we show:

$$\mathbf{NP} \subseteq \mathbf{PP}$$

Let $L \in \mathbf{NP}$ and M be a polynomial NTM for L. Design a NTM M' that for f an input w nondeterministically chooses and performs one of two steps:

1. (1) M' accepts (2) M' simulates M on the input w.

INCLUSIONS between MAIN COMPLEXITY CLASSES

Theorem

$$\mathbf{P} \subseteq \mathbf{ZPP} \subseteq \mathbf{RP} \subseteq \mathbf{NP} \subseteq \mathbf{PP} \subseteq \mathbf{PSPACE}$$

Proof: Since relations $\mathbf{P} \subseteq \mathbf{ZPP} \subseteq \mathbf{RP}$ are obvious, we show first

$$\mathbf{RP} \subseteq \mathbf{NP}$$

If $L \in \mathbf{RP}$ then there is a NTM M accepting L with Monte Carlo acceptance. Hence, $L \in \mathbf{NP}$. Now we show:

$$\mathbf{NP} \subseteq \mathbf{PP}$$

Let $L \in \mathbf{NP}$ and M be a polynomial NTM for L. Design a NTM M' that for f an input w nondeterministically chooses and performs one of two steps:

1. (1) M' accepts (2) M' simulates M on the input w.

M' can be transformed into an equivalent NTM M'' that always have two choices and all its computations on w have the same length.

INCLUSIONS between MAIN COMPLEXITY CLASSES

Theorem

$$\mathbf{P} \subseteq \mathbf{ZPP} \subseteq \mathbf{RP} \subseteq \mathbf{NP} \subseteq \mathbf{PP} \subseteq \mathbf{PSPACE}$$

Proof: Since relations $\mathbf{P} \subseteq \mathbf{ZPP} \subseteq \mathbf{RP}$ are obvious, we show first

$$\mathbf{RP} \subseteq \mathbf{NP}$$

If $L \in \mathbf{RP}$ then there is a NTM M accepting L with Monte Carlo acceptance. Hence, $L \in \mathbf{NP}$. Now we show:

$$\mathbf{NP} \subseteq \mathbf{PP}$$

Let $L \in \mathbf{NP}$ and M be a polynomial NTM for L. Design a NTM M' that for f an input w nondeterministically chooses and performs one of two steps:

1. (1) M' accepts (2) M' simulates M on the input w.

M' can be transformed into an equivalent NTM M'' that always have two choices and all its computations on w have the same length. M'' therefore accepts L by majority what implies: $L \in \mathbf{PP}$.

INCLUSIONS between MAIN COMPLEXITY CLASSES

Theorem

$$\mathbf{P} \subseteq \mathbf{ZPP} \subseteq \mathbf{RP} \subseteq \mathbf{NP} \subseteq \mathbf{PP} \subseteq \mathbf{PSPACE}$$

Proof: Since relations $\mathbf{P} \subseteq \mathbf{ZPP} \subseteq \mathbf{RP}$ are obvious, we show first

$$\mathbf{RP} \subseteq \mathbf{NP}$$

If $L \in \mathbf{RP}$ then there is a NTM M accepting L with Monte Carlo acceptance. Hence, $L \in \mathbf{NP}$. Now we show:

$$\mathbf{NP} \subseteq \mathbf{PP}$$

Let $L \in \mathbf{NP}$ and M be a polynomial NTM for L. Design a NTM M' that for f an input w nondeterministically chooses and performs one of two steps:

1. (1) M' accepts (2) M' simulates M on the input w.

M' can be transformed into an equivalent NTM M'' that always have two choices and all its computations on w have the same length. M'' therefore accepts L by majority what implies: $L \in \mathbf{PP}$. Indeed: If $w \notin L$, then exactly half of computations accept – those corresponding to step 1.
If $w \in L$, then there is at least one computation of M that accepts $w \Rightarrow$ more than half of computations of M'' accept. In addition, it holds $\mathbf{PP} \subseteq \mathbf{PSPACE}$.

COMPLEXITY CLASS BPP

COMPLEXITY CLASS BPP

- Acceptance by clear majority seems to be the most important concept of the randomized computing.

COMPLEXITY CLASS BPP

- Acceptance by clear majority seems to be the most important concept of the randomized computing.
- The number $\frac{3}{4}$ used in the definition of the class **BPP** can be replaced by any number larger than $\frac{1}{2}$.

COMPLEXITY CLASS BPP

- Acceptance by clear majority seems to be the most important concept of the randomized computing.
- The number $\frac{3}{4}$ used in the definition of the class **BPP** can be replaced by any number larger than $\frac{1}{2}$. In other words, for any $\varepsilon < \frac{1}{2}$ we can say that an BPP-algorithm accepts (rejects) any word from (not from) the underlying language with the probability at least $1 - \varepsilon$

COMPLEXITY CLASS BPP

- Acceptance by clear majority seems to be the most important concept of the randomized computing.
- The number $\frac{3}{4}$ used in the definition of the class **BPP** can be replaced by any number larger than $\frac{1}{2}$. In other words, for any $\varepsilon < \frac{1}{2}$ we can say that an BPP-algorithm accepts (rejects) any word from (not from) the underlying language with the probability at least $1 - \varepsilon$
- **BPP**–algorithms allow to diminish, by a repeated application, the probability of error as much as needed.

COMPLEXITY CLASS BPP

- Acceptance by clear majority seems to be the most important concept of the randomized computing.
- The number $\frac{3}{4}$ used in the definition of the class **BPP** can be replaced by any number larger than $\frac{1}{2}$. In other words, for any $\varepsilon < \frac{1}{2}$ we can say that an BPP-algorithm accepts (rejects) any word from (not from) the underlying language with the probability at least $1 - \varepsilon$
- **BPP**–algorithms allow to diminish, by a repeated application, the probability of error as much as needed.
- It *seems* that **P** $\subsetneq$ **BPP** $\subsetneq$ **NP** and therefore the class **BPP** seems to be a reasonable extension of the class **P** and as a class of *feasible problems*.

COMPLEXITY CLASS BPP

- Acceptance by clear majority seems to be the most important concept of the randomized computing.
- The number $\frac{3}{4}$ used in the definition of the class **BPP** can be replaced by any number larger than $\frac{1}{2}$. In other words, for any $\varepsilon < \frac{1}{2}$ we can say that an BPP-algorithm accepts (rejects) any word from (not from) the underlying language with the probability at least $1 - \varepsilon$
- **BPP**–algorithms allow to diminish, by a repeated application, the probability of error as much as needed.
- It *seems* that $\mathbf{P} \subsetneqq \mathbf{BPP} \subsetneqq \mathbf{NP}$ and therefore the class **BPP** seems to be a reasonable extension of the class **P** and as a class of *feasible problems*.

Theorem All languages in **BPP** have polynomial size Boolean circuits.

COMPLEXITY CLASS BPP

- Acceptance by clear majority seems to be the most important concept of the randomized computing.
- The number $\frac{3}{4}$ used in the definition of the class **BPP** can be replaced by any number larger than $\frac{1}{2}$. In other words, for any $\varepsilon < \frac{1}{2}$ we can say that an BPP-algorithm accepts (rejects) any word from (not from) the underlying language with the probability at least $1 - \varepsilon$
- **BPP**–algorithms allow to diminish, by a repeated application, the probability of error as much as needed.
- It *seems* that $\mathbf{P} \subsetneqq \mathbf{BPP} \subsetneqq \mathbf{NP}$ and therefore the class **BPP** seems to be a reasonable extension of the class **P** and as a class of *feasible problems*.

Theorem All languages in **BPP** have polynomial size Boolean circuits.

Definition A language $L \subseteq \{0,1\}^\star$ has polynomial size Boolean circuits if there is a family of Boolean circuits $G = \{C_i\}_{i=1}^{\infty}$ and a polynomial p such that size of C_n is bounded by $p(n)$, C_n has n inputs and $x \in L$ iff the output of $C_{|x|}$ is 1 if its input is x.

AMPLIFICATION of PROBABILITIES

AMPLIFICATION of PROBABILITIES

Let a PTM $\mathcal{M}$ have a probability of error at solving a decision problem at most $\varepsilon < \frac{1}{2}$.

AMPLIFICATION of PROBABILITIES

Let a PTM $\mathcal{M}$ have a probability of error at solving a decision problem at most $\varepsilon < \frac{1}{2}$. Let us run $\mathcal{M}$ for the same input k times and take as the output the majority one (in other words apply so called **majority voting**).

AMPLIFICATION of PROBABILITIES

Let a PTM $\mathcal{M}$ have a probability of error at solving a decision problem at most $\varepsilon < \frac{1}{2}$. Let us run $\mathcal{M}$ for the same input k times and take as the output the majority one (in other words apply so called **majority voting**).
In order to determine how wrong may be such majority voting, observe that for any subset $S \subseteq \{1, \ldots, k\}, |S| \leq k/2$ the probability that majority voting provided by outcomes at such a set of runs is erroneous is smaller than $(1-\varepsilon)^{|S|}\varepsilon^{k-|S|}$.

AMPLIFICATION of PROBABILITIES

Let a PTM $\mathcal{M}$ have a probability of error at solving a decision problem at most $\varepsilon < \frac{1}{2}$. Let us run $\mathcal{M}$ for the same input k times and take as the output the majority one (in other words apply so called **majority voting**).
In order to determine how wrong may be such majority voting, observe that for any subset $S \subseteq \{1, \ldots, k\}, |S| \leq k/2$ the probability that majority voting provided by outcomes at such a set of runs is erroneous is smaller than $(1-\varepsilon)^{|S|}\varepsilon^{k-|S|}$.
Such a majority voting will therefore be wrong with probability

$$p_{err} \leq \sum_{S \subseteq \{1,\ldots,k\}, |S| \leq k/2} (1-\varepsilon)^{|S|}\varepsilon^{k-|S|} \tag{1}$$

$$=$$

AMPLIFICATION of PROBABILITIES

Let a PTM $\mathcal{M}$ have a probability of error at solving a decision problem at most $\varepsilon < \frac{1}{2}$. Let us run $\mathcal{M}$ for the same input k times and take as the output the majority one (in other words apply so called **majority voting**).

In order to determine how wrong may be such majority voting, observe that for any subset $S \subseteq \{1, \ldots, k\}, |S| \leq k/2$ the probability that majority voting provided by outcomes at such a set of runs is erroneous is smaller than $(1-\varepsilon)^{|S|}\varepsilon^{k-|S|}$.

Such a majority voting will therefore be wrong with probability

$$p_{err} \leq \sum_{S \subseteq \{1,\ldots,k\}, |S| \leq k/2} (1-\varepsilon)^{|S|}\varepsilon^{k-|S|} \quad (1)$$

$$= ((1-\varepsilon)\varepsilon)^{k/2} \sum_{S \subseteq \{1,\ldots,k\}, |S| \leq k/2} \left(\frac{\varepsilon}{1-\varepsilon}\right)^{k/2-|S|} \quad (2)$$

$$< ($$

AMPLIFICATION of PROBABILITIES

Let a PTM $\mathcal{M}$ have a probability of error at solving a decision problem at most $\varepsilon < \frac{1}{2}$. Let us run $\mathcal{M}$ for the same input k times and take as the output the majority one (in other words apply so called **majority voting**).
In order to determine how wrong may be such majority voting, observe that for any subset $S \subseteq \{1, \dots, k\}, |S| \leq k/2$ the probability that majority voting provided by outcomes at such a set of runs is erroneous is smaller than $(1-\varepsilon)^{|S|}\varepsilon^{k-|S|}$.
Such a majority voting will therefore be wrong with probability

$$p_{err} \leq \sum_{S\subseteq\{1,\dots,k\},|S|\leq k/2} (1-\varepsilon)^{|S|}\varepsilon^{k-|S|} \quad (1)$$

$$= ((1-\varepsilon)\varepsilon)^{k/2} \sum_{S\subseteq\{1,\dots,k\},|S|\leq k/2} \left(\frac{\varepsilon}{1-\varepsilon}\right)^{k/2-|S|} \quad (2)$$

$$< (\sqrt{\varepsilon(1-\varepsilon)})^k 2^k = \lambda^k, \quad (3)$$

AMPLIFICATION of PROBABILITIES

Let a PTM $\mathcal{M}$ have a probability of error at solving a decision problem at most $\varepsilon < \frac{1}{2}$. Let us run $\mathcal{M}$ for the same input k times and take as the output the majority one (in other words apply so called **majority voting**).
In order to determine how wrong may be such majority voting, observe that for any subset $S \subseteq \{1, \ldots, k\}, |S| \leq k/2$ the probability that majority voting provided by outcomes at such a set of runs is erroneous is smaller than $(1-\varepsilon)^{|S|}\varepsilon^{k-|S|}$.
Such a majority voting will therefore be wrong with probability

$$p_{err} \leq \sum_{S\subseteq\{1,\ldots,k\},|S|\leq k/2} (1-\varepsilon)^{|S|}\varepsilon^{k-|S|} \tag{1}$$

$$= ((1-\varepsilon)\varepsilon)^{k/2} \sum_{S\subseteq\{1,\ldots,k\},|S|\leq k/2} \left(\frac{\varepsilon}{1-\varepsilon}\right)^{k/2-|S|} \tag{2}$$

$$< (\sqrt{\varepsilon(1-\varepsilon)})^k 2^k = \lambda^k, \tag{3}$$

where $\lambda = 2\sqrt{\varepsilon(1-\varepsilon)} < 1$, because the above sum is $\leq 2^k$, since $\frac{\varepsilon}{1-\varepsilon} \leq 1$.

AMPLIFICATION of PROBABILITIES

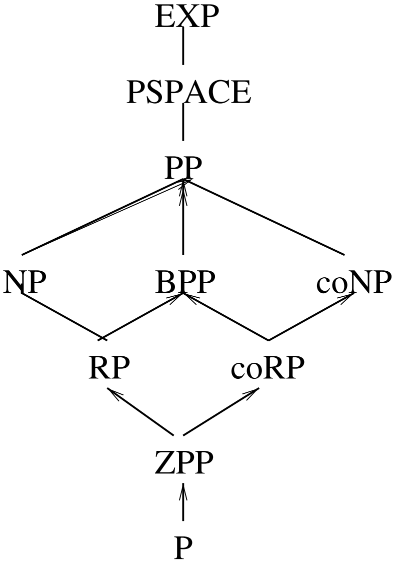
Let a PTM $\mathcal{M}$ have a probability of error at solving a decision problem at most $\varepsilon < \frac{1}{2}$. Let us run $\mathcal{M}$ for the same input k times and take as the output the majority one (in other words apply so called **majority voting**).
In order to determine how wrong may be such majority voting, observe that for any subset $S \subseteq \{1, \ldots, k\}, |S| \leq k/2$ the probability that majority voting provided by outcomes at such a set of runs is erroneous is smaller than $(1-\varepsilon)^{|S|}\varepsilon^{k-|S|}$.
Such a majority voting will therefore be wrong with probability

$$
\begin{aligned}
p_{err} &\leq \sum_{S\subseteq\{1,\ldots,k\},|S|\leq k/2} (1-\varepsilon)^{|S|}\varepsilon^{k-|S|} && (1)\\
&= ((1-\varepsilon)\varepsilon)^{k/2} \sum_{S\subseteq\{1,\ldots,k\},|S|\leq k/2} \left(\frac{\varepsilon}{1-\varepsilon}\right)^{k/2-|S|} && (2)\\
&< (\sqrt{\varepsilon(1-\varepsilon)})^k 2^k = \lambda^k, && (3)
\end{aligned}
$$

where $\lambda = 2\sqrt{\varepsilon(1-\varepsilon)} < 1$, because the above sum is $\leq 2^k$, since $\frac{\varepsilon}{1-\varepsilon} \leq 1$.

In case k is big enough, the effective error probability will be as small as we wish. This process is called **amplification of probability**.

HIERARCHY of COMPLEXITY CLASSES



CLASS MA

CLASS MA

The class **BPP** can be seen as a randomized version of the class **P**. In a similar way the class **MA** (Marlin-Arthur), defined bellow, can be seen as a randomized version of the class **NP**.

CLASS MA

The class **BPP** can be seen as a randomized version of the class **P**. In a similar way the class **MA** (Marlin-Arthur), defined bellow, can be seen as a randomized version of the class **NP**.

MA is the class of decision problems solvable by a Merlin-Arthur protocol, which goes as follows: Merlin, who has unbounded computational resources, sends Arthur a polynomial-size to-be-proof that the answer to the problem is "yes". Arthur must verify the proof in BPP, so that if the answer to the decision problem is

CLASS MA

The class **BPP** can be seen as a randomized version of the class **P**. In a similar way the class **MA** (Marlin-Arthur), defined bellow, can be seen as a randomized version of the class **NP**.

MA is the class of decision problems solvable by a Merlin-Arthur protocol, which goes as follows: Merlin, who has unbounded computational resources, sends Arthur a polynomial-size to-be-proof that the answer to the problem is "yes". Arthur must verify the proof in BPP, so that if the answer to the decision problem is

- "yes", then there exists a proof which Arthur accepts with probability at least $\frac{2}{3}$.

CLASS MA

The class **BPP** can be seen as a randomized version of the class **P**. In a similar way the class **MA** (Marlin-Arthur), defined bellow, can be seen as a randomized version of the class **NP**.

MA is the class of decision problems solvable by a Merlin-Arthur protocol, which goes as follows: Merlin, who has unbounded computational resources, sends Arthur a polynomial-size to-be-proof that the answer to the problem is "yes". Arthur must verify the proof in BPP, so that if the answer to the decision problem is

- "yes", then there exists a proof which Arthur accepts with probability at least $\frac{2}{3}$.
- "no", then Arthur accepts any "to-be-proof" with probability at most $\frac{1}{3}$.

CLASS MA

The class **BPP** can be seen as a randomized version of the class **P**. In a similar way the class **MA** (Marlin-Arthur), defined bellow, can be seen as a randomized version of the class **NP**.

MA is the class of decision problems solvable by a Merlin-Arthur protocol, which goes as follows: Merlin, who has unbounded computational resources, sends Arthur a polynomial-size to-be-proof that the answer to the problem is "yes". Arthur must verify the proof in BPP, so that if the answer to the decision problem is

- "yes", then there exists a proof which Arthur accepts with probability at least $\frac{2}{3}$.
- "no", then Arthur accepts any "to-be-proof" with probability at most $\frac{1}{3}$.

An alternative definition requires that if the answer is "yes", then there exists a proof that Arthur accepts with certainty.

CLASS MA

The class **BPP** can be seen as a randomized version of the class **P**. In a similar way the class **MA** (Marlin-Arthur), defined bellow, can be seen as a randomized version of the class **NP**.

MA is the class of decision problems solvable by a Merlin-Arthur protocol, which goes as follows: Merlin, who has unbounded computational resources, sends Arthur a polynomial-size to-be-proof that the answer to the problem is "yes". Arthur must verify the proof in BPP, so that if the answer to the decision problem is

- "yes", then there exists a proof which Arthur accepts with probability at least $\frac{2}{3}$.
- "no", then Arthur accepts any "to-be-proof" with probability at most $\frac{1}{3}$.

An alternative definition requires that if the answer is "yes", then there exists a proof that Arthur accepts with certainty.

It can be shown that if $\mathbf{P} = \mathbf{BPP}$, then **MA=NP**.

HOW IMPORTANT is RANDOMNESS for DESIGN of ALGORITHMS

HOW IMPORTANT is RANDOMNESS for DESIGN of ALGORITHMS

- The answer depends much on how we define when an algorithm is "efficient".

HOW IMPORTANT is RANDOMNESS for DESIGN of ALGORITHMS

- The answer depends much on how we define when an algorithm is "efficient".
- If constant factors are of importance, then randomization is clearly of large importance.

HOW IMPORTANT is RANDOMNESS for DESIGN of ALGORITHMS

- The answer depends much on how we define when an algorithm is "efficient".
- If constant factors are of importance, then randomization is clearly of large importance.
- If we consider $\mathcal{O}(n), \mathcal{O}(n^2)$ and also $\mathcal{O}(n^3)$ algorithms as still efficient, but already $\mathcal{O}(n^4)$ algorithms as not, then randomness is still of importance for some problems.

HOW IMPORTANT is RANDOMNESS for DESIGN of ALGORITHMS

- The answer depends much on how we define when an algorithm is "efficient".
- If constant factors are of importance, then randomization is clearly of large importance.
- If we consider $\mathcal{O}(n), \mathcal{O}(n^2)$ and also $\mathcal{O}(n^3)$ algorithms as still efficient, but already $\mathcal{O}(n^4)$ algorithms as not, then randomness is still of importance for some problems.
- If "polynomial-time computability" is used for efficiency criterion, we do not know answer yet but we maybe able to claim that randomness is not essential.

HOW IMPORTANT is RANDOMNESS for DESIGN of ALGORITHMS

- The answer depends much on how we define when an algorithm is "efficient".
- If constant factors are of importance, then randomization is clearly of large importance.
- If we consider $\mathcal{O}(n), \mathcal{O}(n^2)$ and also $\mathcal{O}(n^3)$ algorithms as still efficient, but already $\mathcal{O}(n^4)$ algorithms as not, then randomness is still of importance for some problems.
- If "polynomial-time computability" is used for efficiency criterion, we do not know answer yet but we maybe able to claim that randomness is not essential. **Why**

HOW IMPORTANT is RANDOMNESS for DESIGN of ALGORITHMS

- The answer depends much on how we define when an algorithm is "efficient".
- If constant factors are of importance, then randomization is clearly of large importance.
- If we consider $\mathcal{O}(n), \mathcal{O}(n^2)$ and also $\mathcal{O}(n^3)$ algorithms as still efficient, but already $\mathcal{O}(n^4)$ algorithms as not, then randomness is still of importance for some problems.
- If "polynomial-time computability" is used for efficiency criterion, we do not know answer yet but we maybe able to claim that randomness is not essential. **Why**
- There is a strong evidence that $\mathbf{P} = \mathbf{BPP}$.

HOW IMPORTANT is RANDOMNESS for DESIGN of ALGORITHMS

- The answer depends much on how we define when an algorithm is "efficient".
- If constant factors are of importance, then randomization is clearly of large importance.
- If we consider $\mathcal{O}(n), \mathcal{O}(n^2)$ and also $\mathcal{O}(n^3)$ algorithms as still efficient, but already $\mathcal{O}(n^4)$ algorithms as not, then randomness is still of importance for some problems.
- If "polynomial-time computability" is used for efficiency criterion, we do not know answer yet but we maybe able to claim that randomness is not essential. **Why**
- There is a strong evidence that $\mathbf{P} = \mathbf{BPP}$.
- Such assumption is based on results showing that computational hardness of some problems can be used to generate pseudorandom sequences that look random to all polynomial time algorithms.

HOW IMPORTANT is RANDOMNESS for DESIGN of ALGORITHMS

- The answer depends much on how we define when an algorithm is "efficient".
- If constant factors are of importance, then randomization is clearly of large importance.
- If we consider $\mathcal{O}(n), \mathcal{O}(n^2)$ and also $\mathcal{O}(n^3)$ algorithms as still efficient, but already $\mathcal{O}(n^4)$ algorithms as not, then randomness is still of importance for some problems.
- If "polynomial-time computability" is used for efficiency criterion, we do not know answer yet but we maybe able to claim that randomness is not essential. **Why**
- There is a strong evidence that **P = BPP**.
- Such assumption is based on results showing that computational hardness of some problems can be used to generate pseudorandom sequences that look random to all polynomial time algorithms.
- Using such techniques Widgerson and Impagliazo showed that **P=BPP** if there is a problem computable in an exponential time that requires circuits of exponential size.

WHAT is PROBABILITY- of an EVENT?

WHAT is PROBABILITY- of an EVENT?

Intuitively, **probability of an elementary event e in a finite set of events E** is the ratio between the number of e-favorable elementary events in E to the total number of all possible elementary events involved in E.

WHAT is PROBABILITY- of an EVENT?

Intuitively, **probability of an elementary event e in a finite set of events E** is the ratio between the number of e-favorable elementary events in E to the total number of all possible elementary events involved in E.

$$Pr(e \in E) = \frac{\text{number of favorable e-events in } E}{\text{number of all possible elementary events in } E}$$

Example When tossing a perfect dice with it sides labeled by 1, 2,3, 4, 5 6, then the probability that the outcome of a perfectly random tossing of such a dice is 3 is

WHAT is PROBABILITY- of an EVENT?

Intuitively, **probability of an elementary event e in a finite set of events E** is the ratio between the number of e-favorable elementary events in E to the total number of all possible elementary events involved in E.

$$Pr(e \in E) = \frac{\text{number of favorable e-events in } E}{\text{number of all possible elementary events in } E}$$

Example When tossing a perfect dice with it sides labeled by 1, 2,3, 4, 5 6, then the probability that the outcome of a perfectly random tossing of such a dice is 3 is

$$\frac{1}{6}$$

PUZZLE

In case the set of elementary events E is infinite situation is much more complex as the following example discuss in lecture 3 illustrates.

BERTRAND's PROBLEM - PARADOX

BERTRAND's PROBLEM - PARADOX

The following problem has at least three very different (and correct) solutions,

BERTRAND's PROBLEM - PARADOX

The following problem has at least three very different (and correct) solutions, with different outcomes.

BERTRAND's PROBLEM - PARADOX

The following problem has at least three very different (and correct) solutions, with different outcomes. This indicates how tricky are concepts of probability and randomness.

BERTRAND's PROBLEM - PARADOX

The following problem has at least three very different (and correct) solutions, with different outcomes. This indicates how tricky are concepts of probability and randomness.

Problem See the next figure.

BERTRAND's PROBLEM - PARADOX

The following problem has at least three very different (and correct) solutions, with different outcomes. This indicates how tricky are concepts of probability and randomness.

Problem See the next figure. Fix a circle of radius 1. Draw in the circle equilateral triangle and denote l its length.

BERTRAND's PROBLEM - PARADOX

The following problem has at least three very different (and correct) solutions, with different outcomes. This indicates how tricky are concepts of probability and randomness.

Problem See the next figure. Fix a circle of radius 1. Draw in the circle equilateral triangle and denote l its length. Choose randomly a chord d (and denote m its length) of the circle. What is the probability that $m \geq l$?