Randomized Algorithms

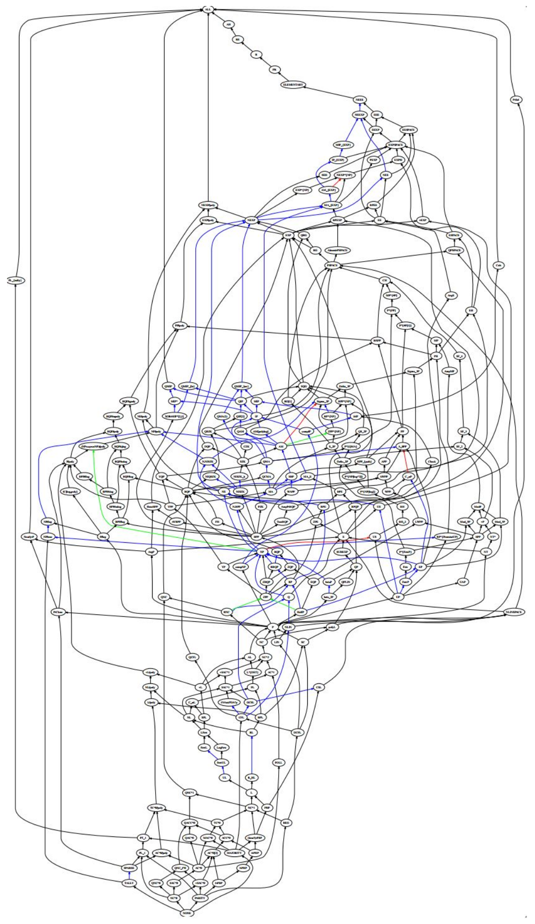
Ways to deal with NP-Hardness

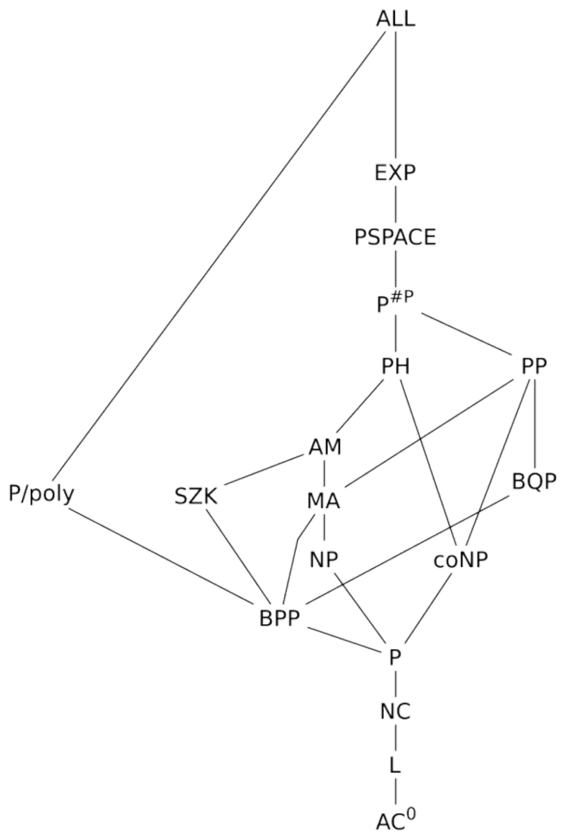
1. **Approximation algorithms**
2. Restrict to **special classes of inputs**
   - E.g. randomly-generated inputs, planar graphs, …
3. **Heuristics:** algorithms without provable guarantees that seem to work well in practice
   - SAT solvers sometimes do well in practice
4. If your **input is small**, sometimes you can afford to run an exponential-time algorithm

Goodbye Complexity…

The Complexity Zoo

(a database of 547 complexity classes)





The Petting Zoo

Scott Aaronson, zookeeper

Goodbye Complexity…

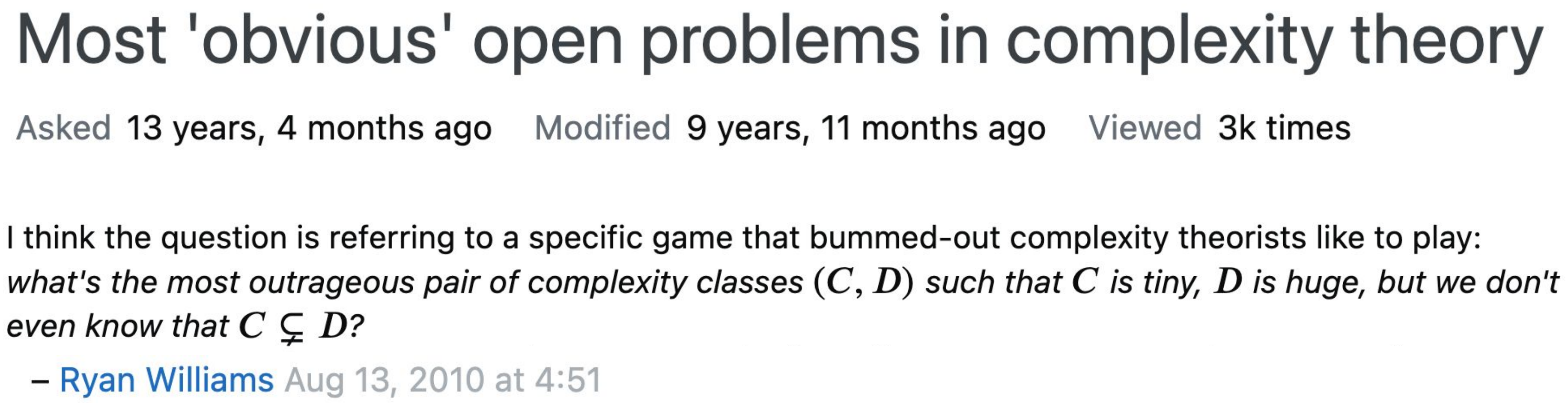
EECS 574: Computational Complexity

- more complexity classes
- hardness of approximation

EECS 477: Introduction to Algorithms

- more NP-hardness proofs
- more approximation algorithms

Open problems: Nearly everything



Most 'obvious' open problems in complexity theory

Asked 13 years, 4 months ago Modified 9 years, 11 months ago Viewed 3k times

I think the question is referring to a specific game that bummed-out complexity theorists like to play: *what's the most outrageous pair of complexity classes (C, D) such that C is tiny, D is huge, but we don't even know that $C \subsetneq D$?*

– Ryan Williams Aug 13, 2010 at 4:51

Techniques and Paradigms in this Course

- Divide-and-conquer, greed, dynamic programming, **the power of randomness**

  Problems that are **easy** for a computer

- Computability

  Problems that are **impossible** for a computer

- NP-completeness and approximation algorithms

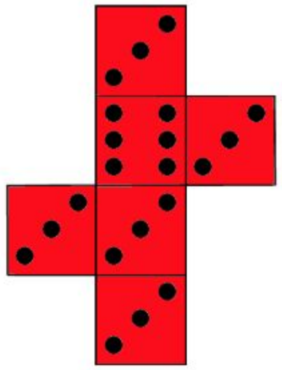
  Problems that are **"probably hard"** for a computer

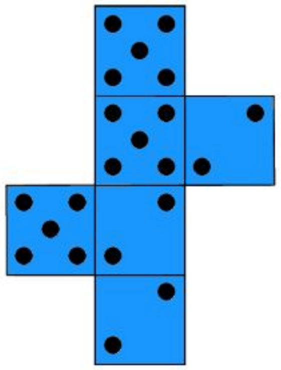
- Cryptography

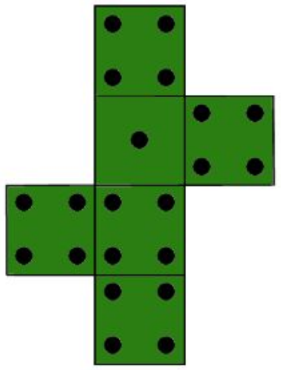
  Using "probably hard" problems for our benefit (hiding secrets)

Randomness can be unintuitive!

“Intransitive dice”







France, 1654

"Chevalier de Méré"
aka Antoine Gombaud
(a gambler)

Let's bet:

I will roll a die four times.
I win if I get a 1.

Hmm…

No one wants to take
this bet any more.

France, 1654

"Chevalier de Méré"
aka Antoine Gombaud
(a gambler)

New bet:

I will roll two dice, 24 times.
I win if I get double 1's.

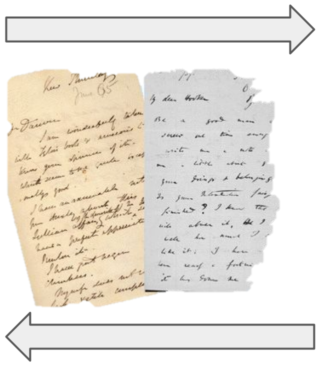
Hmm…

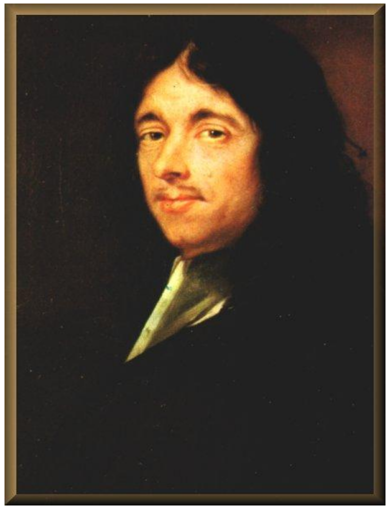
I keep losing money!

"It was only after gamblers invented intricate games of chance, sometimes carefully designed to trick us into bad choices, that mathematicians like Christiaan Huygens, Blaise Pascal and Pierre de Fermat found it necessary to develop what we call today probability theory."

- The Book of Why (Pearl and Mackenzie)

Pascal




Fermat

What do we mean by “randomized algorithm”?

Goal: Use randomization to get faster and/or more accurate algorithms (still for worst-case input).

We are NOT considering randomly chosen inputs.

E.g. Quicksort (next lecture):

“The expected running time is O(n log n)” means:

*“**For ANY given array**, in expectation (over the random choices made by the algorithm) the running time is O(n log n)”*

What do we mean by “randomized algorithm”?

Goal: Use randomization to get faster and/or more accurate algorithms (still for worst-case input).

We are NOT considering randomly chosen inputs.

In general, we will make statements like:

[In expectation] or [with probability at least p]

+

[runtime of algorithm is O(f(n))] or [algorithm returns correct answer]

Always imagine “for ANY given input” precedes such statements.

Why would you ever use a randomized algorithm?

…after all, there's no randomness in the problem statement

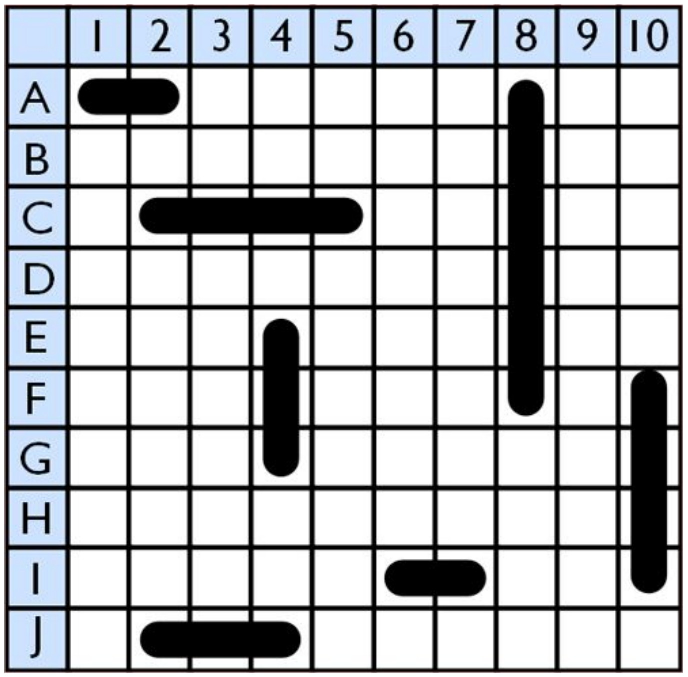
- Sometimes randomization is **provably necessary** ⟶
- Sometimes we **don't know a deterministic algorithm** (even though one may exist)
- Sometimes randomization is not necessary but leads to **simpler** or **faster-in-practice** algorithms

battleship!

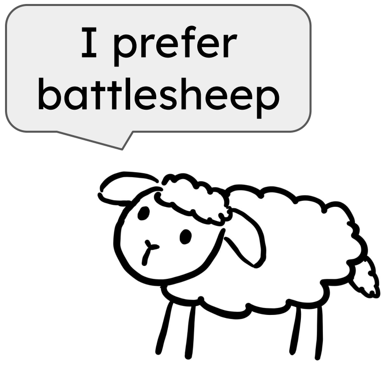
- However, for some applications we have to be careful: don't want an autonomous vehicle that fails 0.5% of the time

The Battleship Problem

Input: A hidden battleship board. When you guess a cell, its status (empty/occupied) is revealed.



Output: Any cell occupied by a ship.



1-Dimensional Battleship Problem

Input: Hidden 0/1 array **A** of length **n**, with exactly **n/4** 1's. When you guess a cell, its value is revealed.

0	0	1	0	0	0	1	1	0	1	0	0	0	0	0	0

Output: Any i such that A[i] = 1.

What algorithm would you use?

1-Dimensional Battleship Problem

Input: Hidden 0/1 array **A** of length **n**, with exactly **n/4** 1's.
When you guess a cell, its value is revealed.

0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1

Output: Any i such that A[i] = 1.

```
Linear scan algorithm:
  for i = 1… n:
    if A[i] = 1: return i
```

Worst-case number of guesses ≥ (3/4)n

1-Dimensional Battleship Problem

Input: Hidden 0/1 array **A** of length **n**, with exactly **n/4** 1's.
When you guess a cell, its value is revealed.

1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0

Output: Any i such that A[i] = 1.

```
Backwards linear scan:
    for i = n… 1:
        if A[i] = 1: return i
```

Worst-case number of guesses ≥ (3/4)n

1-Dimensional Battleship Problem

Input: Hidden 0/1 array **A** of length **n**, with exactly **n/4** 1's.
When you guess a cell, its value is revealed.

0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0

Output: Any i such that A[i] = 1.

```
Alternating linear scan:
    for i = 1, n, 2, n−2, 3, n−3, … :
        if A[i] = 1: return i
```

Worst-case number of guesses ≥ (3/4)n

1-Dimensional Battleship Problem

Input: Hidden 0/1 array **A** of length **n**, with exactly **n/4** 1's. When you guess a cell, its value is revealed.

0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0

Output: Any i such that A[i] = 1.

For **any** deterministic algorithm,
there **exists** an input
such the number of guesses is ≥ (3/4)n.

1-Dimensional Battleship Problem

Input: Hidden 0/1 array **A** of length **n**, with exactly **n/4** 1's.
When you guess a cell, its value is revealed.

0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0

Output: Any i such that A[i] = 1.

```
Random algorithm:
  while true:
    i = random integer in [1… n]
      if A[i] = 1: return i
```

For any given input, expected number of guesses = 4
(we won't analyze formally)

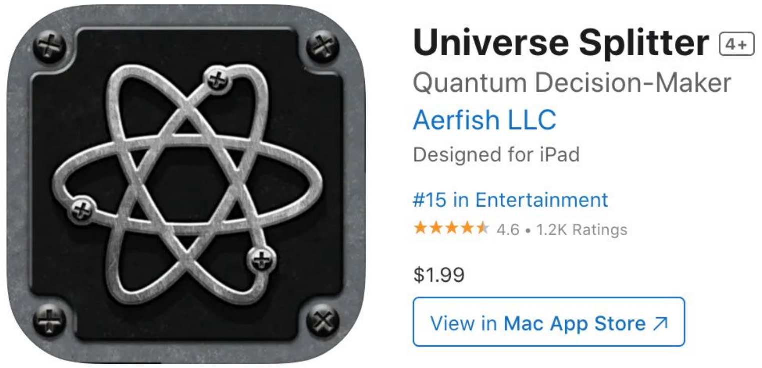
Where do random numbers come from?

In this class we assume access to a perfect random number generator.

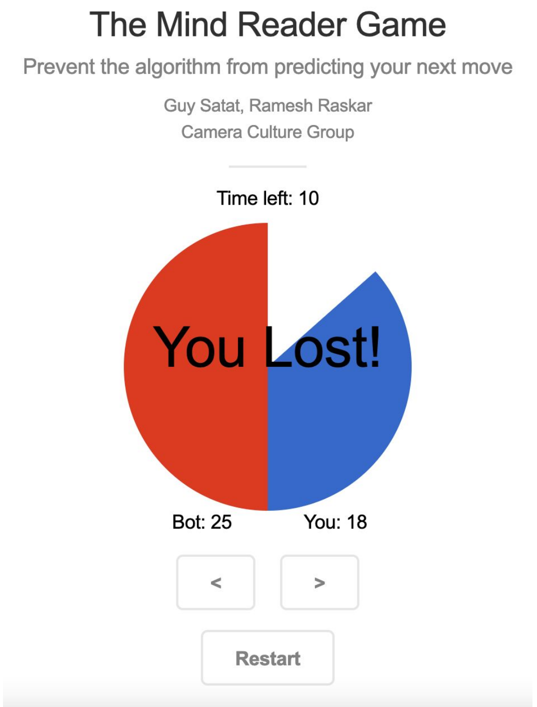
In practice, we use "good enough" random number generators.

If quantum physics produces true randomness, we can also use truly random numbers.

(Humans are generally not good at generating random numbers).



Universe Splitter 4+
Quantum Decision-Maker
Aerfish LLC
Designed for iPad
#15 in Entertainment
★★★★★ 4.6 • 1.2K Ratings
$1.99



The Mind Reader Game
Prevent the algorithm from predicting your next move
Guy Satat, Ramesh Raskar
Camera Culture Group

Time left: 10

You Lost!

Bot: 25
You: 18

Let's look at another toy example to review some concepts in probability:

Rock-Paper-Scissors

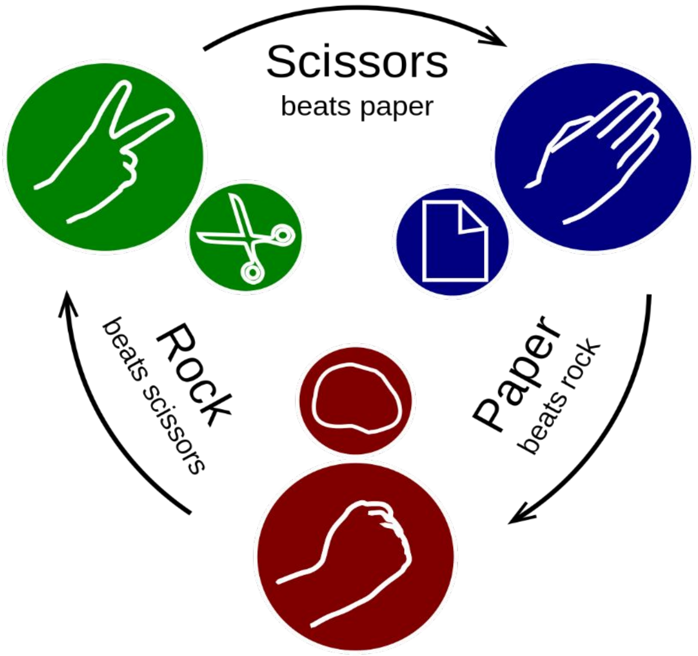
Goal: Minimize our probability of losing in a (best-of-one) game of rock-paper-scissors against an opponent who *knows our strategy*.

With any deterministic strategy, we lose (if opponent is logical). **Why?**

Strategy: Play uniformly at *random*.

Intuition: We lose with probability 1/3.

Let's prove it!



Rock-Paper-Scissors

Terminology Review

The ***sample space*** (set of all possible outcomes)

$$\begin{array}{lccc l} & (R,R) & (R,P) & (R,S) & \text{we play rock} \\ \text{they play rock} & (P,R) & (P,P) & (P,S) & \\ & (S,R) & (S,P) & (S,S) & \end{array}$$

Each outcome is of the form (what we play, what they play)

An ***event*** (subset of outcomes)

Each outcome has a ***probability*** and all of the probabilities add to 1.

The ***probability*** of an ***event*** is the sum of the probabilities of the outcomes in that event.

Rock-Paper-Scissors

The ***sample space*** (set of all possible outcomes)

probability they play rock depends on their strategy

$$\begin{array}{ccc} (R,R) & (R,P) & (R,S) \\ (P,R) & (P,P) & (P,S) \\ (S,R) & (S,P) & (S,S) \end{array}$$

probability we play rock = 1/3

Each outcome is of the form (what we play, what they play)

We can’t calculate the probability of (R,R) or any other outcome since we don’t know their strategy.

But we can still prove we lose with probability ⅓ using ***independence.***

203 Review: Independence

Intuitively: Two events are *independent* if the occurrence of one does not affect the probability of the other occurring.

Formal Definition: Events **A** and **B** are *independent* if

$$\Pr[A \cap B] = \Pr[A] \cdot \Pr[B].$$

(we won't prove the equivalence of the intuition and the formal defn)

E.g. we flip two fair coins:

A is the event that coin 1 is heads,

B is the event that coin 2 is heads.

A and B are independent events and the probability both are heads is: $\Pr[A \cap B] = \Pr[A] \cdot \Pr[B] = \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4}$.

Notice that our random rock-paper-scissors choice is *independent* of our opponent's choice

Rock-Paper-Scissors

Analysis

Claim. *Against any opponent*, by picking rock, paper, or scissors uniformly at random, we lose with probability ⅓.

Event A_x = we pick x; Event B_y = they pick y

The probability that we **lose** is:

$$\Pr(A_R \cap B_P) + \Pr(A_P \cap B_S) + \Pr(A_S \cap B_R)$$

$$= \frac{1}{3}(\Pr(B_P) + \Pr(B_S) + \Pr(B_R)) \text{ (independence)}$$

$$= \frac{1}{3} \quad \text{(since they either pick R, P, or S; covers all outcomes)}$$

France, 1654

"Chevalier de Méré"
aka Antoine Gombaud
(a gambler)

Let's bet:

I will roll a die four times.
I win if I get a 1.

Pr[he loses] = $(5/6)^4$ = 0.482…

New bet:

I will roll two dice, 24 times.
I win if I get double 1's.

Pr[he loses] = $(35/36)^{24}$ = 0.508…

Max-3SAT

3SAT is NP-hard but what if we try the easier goal of satisfying as many clauses as we can?

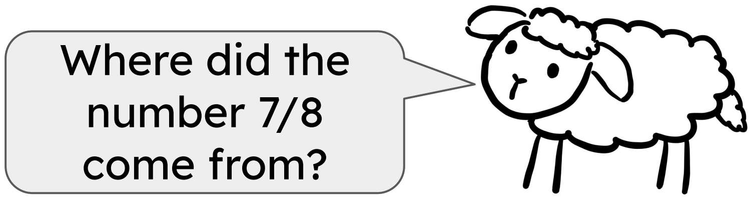
Input: 3CNF formula *where each clause has 3 distinct variables*.

E.g. $(x \lor y \lor z) \land (\neg x \lor y \lor z) \land (x \lor \neg y \lor z) \land (x \lor y \lor \neg z)$
$\land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z) \land (x \lor \neg y \lor \neg z) \land (\neg x \lor \neg y \lor \neg z)$

Our goal: Output an assignment that satisfies at least ⅞ of the clauses.

Wait, does such an assignment always exist? **Yes!** (we will show)

What about for a fraction bigger than ⅞? **No!** (above example)



We could try coming up with a deterministic algorithm… but that sounds complicated.

It turns out there's a very simple **randomized** algorithm: Pick an assignment <u>uniformly at random</u>!

We will show: *For any 3CNF formula*, the ***expected value*** of the fraction of satisfied clauses is $7/8$.

But first let's review the concept of expected value…

203 Review: Random Variables

A **random variable** is a quantity determined by the outcome of *a random experiment*.

E.g. Let **N** be the number of satisfied clauses of 3CNF formula ϕ when we *assign variables T/F independently and uniformly at random*.

$$\phi = \underbrace{(x \vee y \vee z)}_{c1} \wedge \underbrace{(\neg x \vee y \vee z)}_{c2} \wedge \underbrace{(\neg x \vee y \vee z)}_{c3}$$

(c2 and c3 are the same but that's ok)

x y z x y z …

Outcome	F F F	F F T	F T F	F T T	T F F	T F T	T T F	T T T

c1 c2 c3 c1 c2 c3 …

Sat?	N Y Y	Y Y Y	Y Y Y	Y Y Y	Y N N	Y Y Y	Y Y Y	Y Y Y

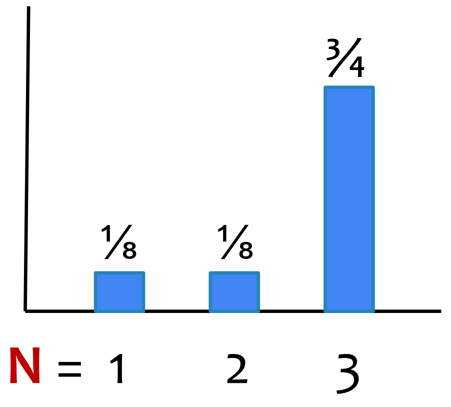
N	2	3	3	3	1	3	3	3

203 Review: Distribution of a Random Variable

The ***probability*** that a random variable equals some fixed value **v** is the sum of the probabilities of all outcomes that result in value **v**.

N	2	3	3	3	1	3	3	3
Pr[outcome]:	⅛	⅛	⅛	⅛	⅛	⅛	⅛	⅛

Pr[**N**=1] = ⅛, Pr[**N**=2] = ⅛, Pr[**N**=3] = ¾



203 Review: Expected Value of a Random Variable

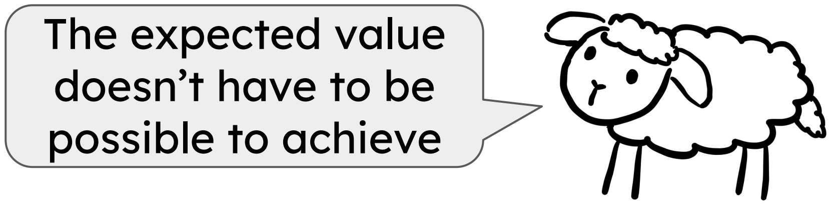
The ***expected value*** of a random variable *N* is the *weighted average* of its values:

$$\mathbb{E}[N] = \sum_{v} v \cdot \Pr[N = v]$$

E.g. Since Pr[**N**=1] = ⅛, Pr[**N**=2] = ⅛, Pr[**N**=3] = ¾,

E[**N**] = 1 · 1/8 + 2 · 1/8 + 3 · 3/4 = 2.625 clauses (which is a ⅞ fraction).

"*In expectation*, a random assignment satisfies ⅞ths of the clauses of ϕ."



We proved: "*In expectation*, a random assignment satisfies ⅞ths of the clauses of ϕ."

But ϕ was one specific 3CNF formula, and we want to prove this for **all** 3CNF formulas (where each clause has 3 distinct variables).

But how? The clauses are not independent of each other!

A very useful trick: indicator variables + linearity of expectation

203 Review: 0/1 Indicator Random Variables

An ***indicator*** random variable X has 2 possible outcomes: 0 and 1.

Expected value of an indicator r.v.: **E[X] = Pr[X=1].** Why?

E.g. For a 3CNF formula $C_1 \wedge C_2 \wedge \ldots \wedge C_m$, with a random assignment of variables, let $\mathbf{N_i}$ be the ***indicator*** random variable:

$$\begin{cases} 1 \text{ if clause } C_i \text{ is satisfied} \\ 0 \text{ otherwise} \end{cases}$$

Observation: The number of clauses satisfied by the assignment is $\mathbf{N = N_1 + N_2 + \ldots + N_m}$.

Review: Linearity of Expectation

Linearity of Expectation: For any (not necessarily independent!) random variables *N_i:*

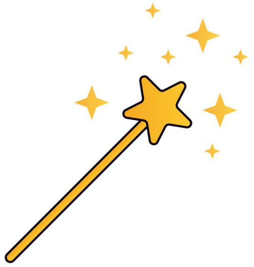
$$\mathbb{E}\Big[\sum_i N_i\Big] = \sum_i \mathbb{E}[N_i].$$

For example:

Let **X** be the outcome of rolling a 6-sided dice (expected value 3.5)

Let **Y** be the outcome of rolling a 12-sided dice (expected value 6.5)

Then the expected sum is: E[**X**+**Y**] = E[**X**] + E[**Y**] = 3.5+6.5 = 10.



Example to demonstrate the magic of indicator rv's + linearity of expectation

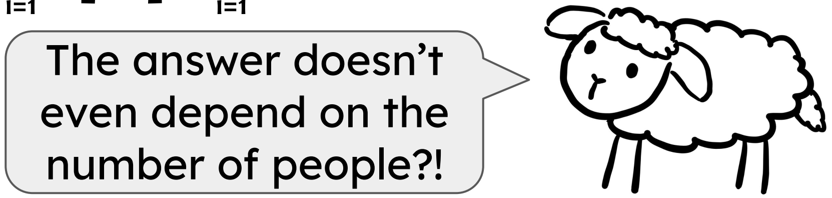
Suppose I have a name tag for all n people in this class, and I hand them out randomly. What is the expected number of people who receive their own name tag?

This sounds like a complicated problem until you learn this technique…

Let $\mathbf{X_i}$ be an ***indicator random variable*** for whether person i receives their own name tag. For all i, $E[\mathbf{X_i}] = \Pr[\mathbf{X_i} = 1] = 1/n$.

Observe that $\sum_{i=1}^{n} \mathbf{X_i}$ is the number of people who receive their own name tag. So our goal is to calculate $E[\sum_{i=1}^{n} \mathbf{X_i}]$.

By ***linearity of expectation***, $E[\sum_{i=1}^{n} \mathbf{X_i}] = \sum_{i=1}^{n} E[\mathbf{X_i}] = \sum_{i=1}^{n} 1/n = 1$.



Now we're ready to do what we set out to do…

Given a 3CNF formula $\phi = C_1 \wedge C_2 \wedge \ldots \wedge C_m$ where each clause has 3 distinct variables, we pick a random T/F assignment of the variables.

Your task: Calculate the E[**N**].

N is the number of clauses satisfied.

$\mathbf{N_i}$ is an indicator random variable for whether clause C_i is satisfied.

"I want all the students in our college to have grades above average."

– Anonymous Dean,
Graduation Ceremony

Fact: For any r.v. **X** there *exists* an outcome ≥ E[**X**].

Fact: For any r.v. **X** there exists an outcome ≤ E[**X**].

Thus, for **every** 3CNF (where each clause has 3 distinct variables), there **exists** an assignment satisfying at least ⅞ of the clauses.

This doesn't immediately say anything about the **probability** that our random assignment satisfies at least ⅞ of the clauses. **Why not?**

Markov's inequality can help us with that…

Markov's Inequality

"A <u>non-negative</u> random variable **X** is rarely way bigger than its expectation."

Markov's Inequality: If X is a *<u>non-negative</u>* random variable and $a > 0$, then $\Pr[X \geq a] \leq \mathbb{E}[X]/a$.

Proof: $\mathbb{E}[X] \geq a\Pr[X \geq a]$ by definition of expectation.

Divide by a.

Markov's Inequality

Let's use Markov's inequality to show it's "likely" that a random assignment satisfies at least **half** of the clauses (i.e. $\geq m/2$ clauses).

Let **N** be the number of satisfied clauses.
Let **N'** be the number of unsatisfied clauses.

Try Markov's inequality two ways:

$\Pr[\mathbf{N} \geq m/2] \leq$

$\Pr[\mathbf{N'} \geq m/2] \leq$

How would you modify the algorithm to achieve higher probability?

Derandomizing the Algorithm

using a technique called "*expectation maximization*"

Theorem: There is an efficient *deterministic* algorithm that outputs an assignment satisfying 7/8ths of the clauses.

Let $z_1,\dots,z_n$ be the variables in the formula.

Suppose I have already chosen an assignment for variables $z_1,\dots,z_i$.

Choose z_{i+1} = T or F to maximize the expected number of satisfied clauses if we choose $z_{i+2},\dots,z_n$ randomly.

(We can calculate this efficiently using linearity of expectation.)

Key idea for proving correctness: Each step, we fix one variable and keep the expected number of clauses satisfied $\geq 7m/8$.

When is Randomization Necessary?

my “bonus” lecture ↓ next week ↓

- In the areas of online algorithms / cryptography / games / etc. when the input is hidden, we can often **prove** randomization is necessary.
- In the “standard” setting (input given upfront), whether or not randomization is necessary is a **big open problem**. Most researchers seem to believe *every* randomized algorithm can be derandomized.
- When randomization isn’t necessary, it’s still useful for getting simpler and/or faster-in-practice algorithms, AND it provides inspiration for designing deterministic algorithms.
- Techniques you saw today:
    - Indicator r.v.’s + linearity of expectation (to calculate expectation)
    - Markov’s inequality (to bound the probability an algorithm fails)