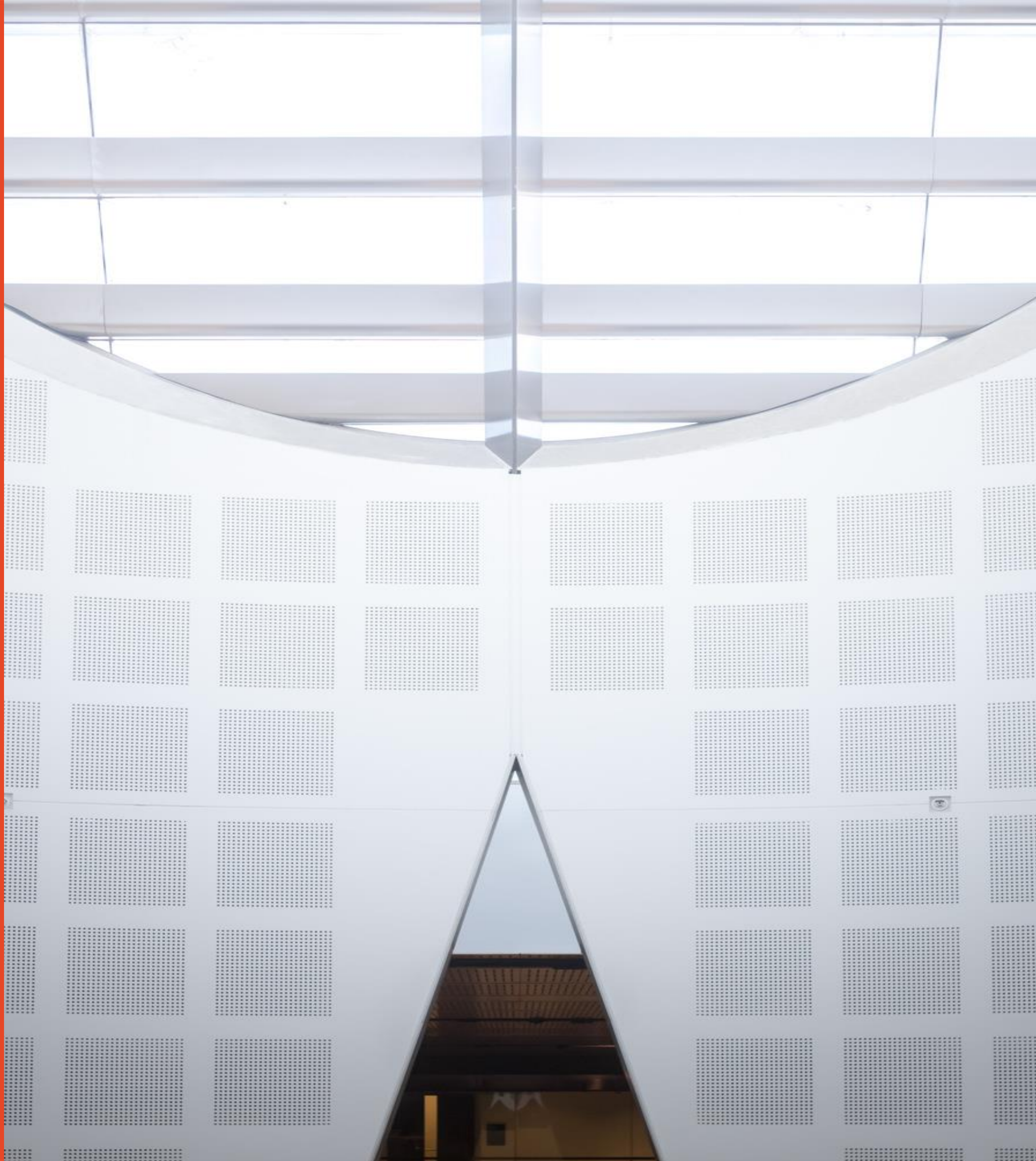
COMPx270: Randomised and Advanced Algorithms
Lecture 1: Randomness, Probability, and Algorithms

Clément Canonne

School of Computer Science

THE UNIVERSITY OF SYDNEY



An experiment 🃏

Shuffle a deck of cards: then go through them in order. How many times do two consecutive cards have the same suit?

An experiment 🃏

Shuffle a deck of cards: then go through them in order. How many times do two consecutive cards have the same suit?

An experiment 🃏

Shuffle a deck of cards: then go through them in order. How many times do two consecutive cards have the same suit?

4♡, 3♡, 8♣, 2♣, 3♠, 10♡, 8◊, 7♠, K♡, 5◊, 8♡, J♡, 9♣, 5♣, J♠, 2♡, Q♠, 2♠, 10♠, 6♠, 6♣, 5♡, 4♣, 9♠, Q◊, 8♠, 6◊, 10◊, 7♣, J♣, K♣, 4◊, K◊, K♠, A◊, A♠, A♣, 4♠, A♡, 3♣, 9◊, 3◊, J◊, 9♡, Q♡, Q♣, 2◊, 10♣, 5♠, 7◊, 6♡, 7♡

An experiment 🃏

Shuffle a deck of cards: then go through them in order. How many times do two consecutive cards have the same suit in expectation?

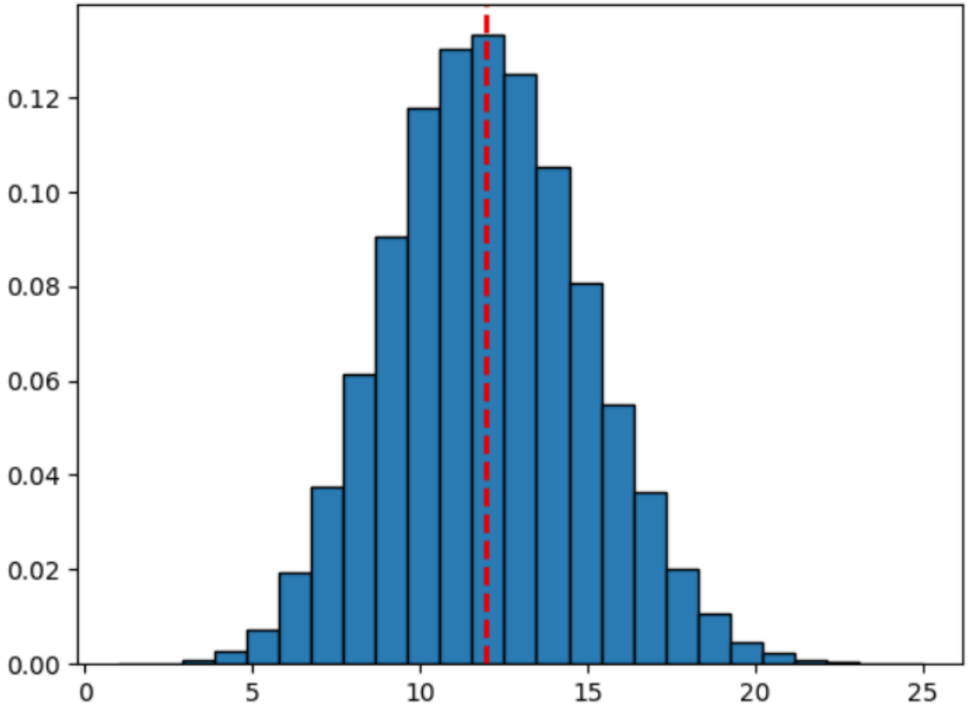
An experiment 🃏

Shuffle a deck of cards: then go through them in order. How many times do two consecutive cards have the same suit in expectation?

```
import numpy as np
import random
deck = 13*[’S’, ’H’, ’D’, ’C’]
consecutives = []
for _ in range(50000):
    shuffled_deck = random.sample(deck, len(deck));
    consecutives += [np.sum([shuffled_deck[i] == shuffled_deck[i+1] for i
     in range(len(deck)-1)])]
print("Empirical mean: %f" % np.mean(consecutives))
```

An experiment 🃏

Shuffle a deck of cards: then go through them in order. How many times do two consecutive cards have the same suit in expectation?



An experiment 🃏

Shuffle a deck of cards: then go through them in order. How many times do two consecutive cards have the same suit in expectation?

Can we prove it?

An experiment 🃏

Shuffle a deck of cards: then go through them in order. How many times do two consecutive cards have the same suit in expectation?

Theorem (Linearity of expectation).

$$\mathbb{E}[X+Y] = \mathbb{E}[X] + \mathbb{E}[Y]$$

No assumption of independence, or anything. Surprisingly useful!

An experiment 🃏

Shuffle a deck of cards: then go through them in order. How many times do two consecutive cards have the same suit in expectation?

Randomised algorithms

Standard algorithms: “recipes.” Input = ingredients, output = 🍰 .

Given input, follow steps, get 🍰.

Given same ingredients, get same 🍰.

Randomised algorithms

Standard algorithms: “recipes.” Input = ingredients, output = cake .

- Given input, follow steps, get 🍰.
- Given same ingredients, get same 🍰.

Randomised algorithms: “recipes with randomness” Input = ingredients, output = cake , randomness = unpredictable oven

- Given input, follow steps, get 🍰.
- Given same ingredients, get 🎂.

Randomised algorithms

Randomized algorithms are algorithms where the behaviour doesn’t depend solely on the input. It also depends (in part) on random choices or the values of a number of random bits. 🎲

Important distinctions: what is (and isn’t) a randomized algo

- the input is assumed to be “random” ❌
- we average the time complexity over many calls to the algo ❌
- the input is worst-case, but the algo makes random choices ✅

Randomised algorithms

(cartoon definition)

Randomised algorithms, Monte Carlo, Las Vegas

(details)

Why randomisation? 🎲

- Avoid pathological corner cases
- Get approximate result very fast
- Avoid predictable outcomes
- Get faster, simpler algorithms
- Break ties or bypass “impossibility results”
- Cryptography! Privacy!

Why not randomisation? 🎲

- Randomness is not always good or desirable
- Random bits don’t grow on trees!
- Bad random bits? Bad outputs.

`secrets` — Generate secure random numbers for managing secrets¶

Added in version 3.6.

Source code: Lib/secrets.py

The `secrets` module is used for generating cryptographically strong random numbers suitable for managing data such as passwords, account authentication, security tokens, and related secrets.

In particular, `secrets` should be used in preference to the default pseudo-random number generator in the `random` module, which is designed for modelling and simulation, not security or cryptography.

Get faster, simpler algorithms? (An example)

Given an n-bit integer, decide whether it is a prime number.

Get faster, simpler algorithms? (An example)

Given an n-bit integer, decide whether it is a prime number.

There exists a polynomial-time algorithm! Since 2002 (AKS).
Runs in time $\tilde{O}(n^{12})$. Improved to $\tilde{O}(n^6)$.

Get faster, simpler algorithms? (An example)

Given an n-bit integer, decide whether it is a prime number.

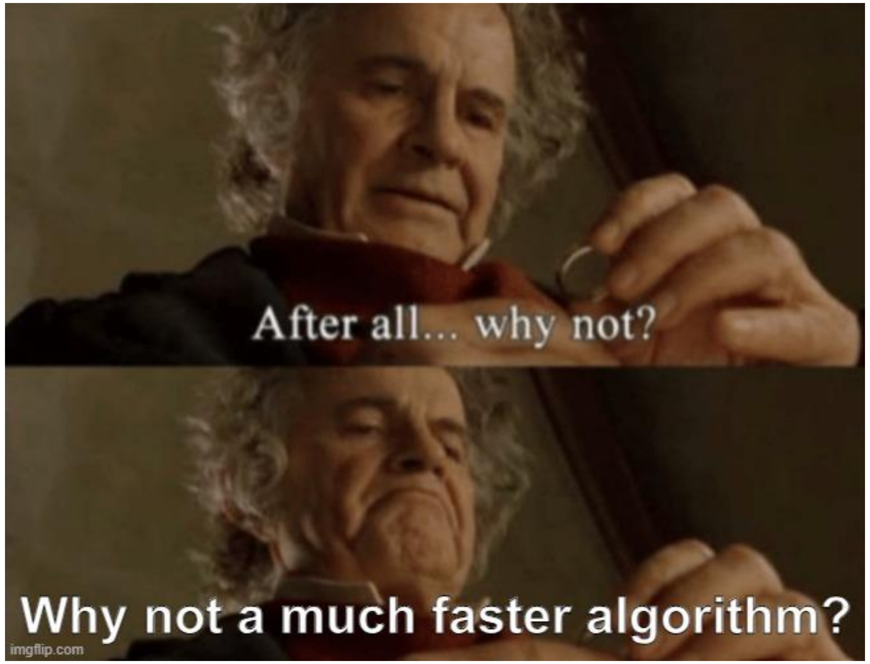
There exists a polynomial-time algorithm! Since 2002 (AKS).
Runs in time $\tilde{O}(n^{12})$. Improved to $\tilde{O}(n^{6})$.

The algorithm was the first one which is able to determine in polynomial time, whether a given number is prime or composite and this without relying on mathematical conjectures such as the generalized Riemann hypothesis. [...] In 2006 the authors received both the Gödel Prize and Fulkerson Prize for their work. (Wikipedia)

Get faster, simpler algorithms? (An example)

Given an n-bit integer, decide whether it is a prime number.

There exists a randomised algorithm! Since 1980 (Miller-Rabin). 🎲
Runs in time $\tilde{O}(n^2)$.



Now, QuickSort

Given an array A of n distinct numbers, sort A.

Theorem. There are deterministic sorting algorithms with running time $O(n \log n)$.

Now, QuickSort

Given an array A of n distinct numbers, sort A.

Theorem. There are deterministic sorting algorithms with running time $O(n \log n)$.

Theorem. Every (comparison-based) sorting algorithm must have worst-case running time $\Omega(n \log n)$.

Now, QuickSort

Require: Input array A of size n

if $n \leq 1$ **then return** A

Select an index $1 \leq i \leq n$, and let $p \leftarrow A[i]$ be the *pivot*

Partition A into 3 subarrays: A_1 (elements smaller than p), A_2 (equal to p), and A_3 (greater than p) $\triangleright$ $O(n)$ time

Recursively call QuickSort on A_1 and A_3 to sort them

Merge the (sorted) A_1, A_2, A_3 into A $\triangleright$ $O(n)$ time

return A

Now, QuickSort

Given an array A of n distinct numbers, sort A.

Theorem. QuickSort is a deterministic sorting algorithm with running time $O(n^2)$.

(But it is simple, and nice, and does well in practice.)

Randomised QuickSort

Require: Input array A of size n

if $n \leq 1$ **then return** A

Select an index $1 \leq i \leq n$, and let $p \leftarrow A[i]$ be the *pivot*

Partition A into 3 subarrays: A_1 (elements smaller than p), A_2 (equal to p), and A_3 (greater than p) $\triangleright$ $O(n)$ time

Recursively call QuickSort on A_1 and A_3 to sort them

Merge the (sorted) A_1, A_2, A_3 into A $\triangleright$ $O(n)$ time

return A

Now, QuickSort

Given an array A of n distinct numbers, sort A.

Theorem. Randomised QuickSort is a sorting algorithm with *expected* running time $O(n \log n)$.

(And it is simple, and still nice, and still does well in practice.)

Now, QuickSort

(proof)

(proof)

Recap, and looking forward

- Randomised, linearity of expectation, applications
- Concentration bounds, probability amplification, median trick
- Coupon Collector, Load Balancing, Power of Two Choices
- Derandomisation: Max-Cut, Method of Conditional Expectations
- Randomized Min-Cut (Karger's algorithm)
- Probabilistic data structures I: Hashing and Bloom filters
- Probabilistic data structures II: Johnson-Lindenstrauss, LSH
- Streaming and Sketching I: definitions, examples, frequency estimation
- Streaming and Sketching II: CountSketch, Count–min Sketch
- Linear Programming and Randomised Rounding
- Embeddings: FRT algorithm, and applications
- Sampling and Counting

To conclude: something completely different!

If X is a non-negative integer-valued random variable, then

$$\mathbb{E}[X] = \sum_{n=0}^{\infty} n \Pr[\,X = n\,] = \sum_{n=1}^{\infty} \Pr[\,X \geq n\,]$$

(This is useful!) See tutorial.