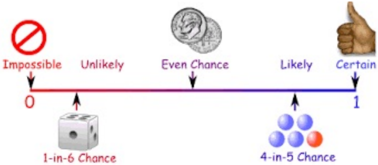
Probabilistic Tools in Algorithms

RA-MIRI QT Curs 2020-2021

What is probability?

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



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

What is probability?

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

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

In the same way, in todays computers, the *random generator* functions are *deterministic* programs, which simulates randomness. What is denoted pseudorandom generators.

Probability and computers

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

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

for a, b constants and m a large integer.

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

A computer deterministically generates pseudorandom numbers.

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

for $(i = 0; i < n; i++)$ **do**
    values[i]=rand() % 2;
    printf("%d", values[i]);

Some applications of probability in CS

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

Some applications of probability in CS

- Data science: To design efficient algorithm for huge data set, usually we do keep a relevant sample, rather than keep all the data.
- Cryptography: Randomness and number theory, are essential for cryptography and crypto-hashing.
- Data compression: improving data compression algorithms passes through analysing and modelling the underlying probability distribution of the data, and evaluating its information-theoretic contets.
- Modelling and analysing the spread of particular infections: Probabilistic ad-hoc graph models and techniques, have play an important role in helping to stop or mitigated massive infections, including e-infections.

Randomization and algorithmis: Probabilistic analysis

Given a deterministic algorithm, it happens that a few "instances" may bias the complexity outcome of the algorithm, which for most of the instances seem to work well, for ex. Quicksort.
In this cases, we can perform a probabilistic analysis of the deterministic algorithm as follows:
Fix a probability distribution on the set of inputs, parametrized by input size. Often the distribution is the uniform, but not always.
We see the number of steps as a random variable $T(n)$ and compute its expected value $\mu = \mathbf{E}[T(n)]$.
We also need to prove concentration, i.e. with high probability, for most of the imputs, $T(n)$ is near μ.

Randomization and algorithmis: Randomized algorithms

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

In this case, we may have to perform a probabilistic analysis of the complexity.

There are two main types of probabilistic algorithms:

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

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

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

A randomized sorting algorithm

What do you know about QuickSort?

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

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

A randomized sorting algorithm

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

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

Generating a permutation uniformly at random

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

The number of different permutations is $n!$.

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

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

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

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

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

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

$\text{Rand}(i)$ provides a random number in $[0, i)$.

Fisher-Yates algorithm

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

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

- That is, each permutation is equally likely.

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