Introduction to programming

Lecture 13:

Lecturers: Giovanni Casini (giovanni.casini@uni.lu) and Xavier Parent (xavier.parent@uni.lu)

Revised version of material from Clément Guérin

Randomness

Random variables

Informally, a random variable is a variable whose values depend on outcomes of a random phenomenon, that is, a phenomenon apparently lacking any form of pattern or predictability.

A more formal definition:

Let Ω be the **set of outcomes** (also known as the **universe**) i.e. a space with a measure $\mathbb{P}$ of total weight 1 on it and E be a **measurable space**. We say that a map X from Ω to E is a **random variable** if it is measurable.

A fundamental example is the coin flips with a fixed bias of $0 \leq p \leq 1$. In this case $\Omega = \{head, tail\}$, $\mathbb{P} = p\delta_{head} + (1-p)\delta_{tail}$ and X is defined by :

$$X(\omega) := \begin{cases} 1 \text{ if } \omega = tail \\ 0 \text{ else} \end{cases}$$

A balanced coin means that you have $p = \frac{1}{2}$.

A random variable is said to be **discrete** if $X(\Omega)$ is finite or countably infinite (typically $\{0, 1\}$ or $\mathbb{N}$). We will only be interested in discrete random variables here.

What we are most interested in is the **probability law** of a random variable X. By definition, this is $\mathbb{P}_X$ where for any measurable subset S of E we have :

$$\mathbb{P}_X(S) := \mathbb{P}\left(X^{-1}(S)\right)$$

There is a wide list of discrete probability laws.

https://en.wikipedia.org/wiki/List_of_probability_distributions (https://en.wikipedia.org/wiki/List_of_probability_distributions)

The most important for us is the **uniform** one. In this case (remember that we are in the discrete case), the probability is defined on a non-empty finite set A of cardinality $n \geq 1$ and the probability of any subset B in A is defined as $\frac{|B|}{|A|}$.

For instance, the probability law defined by a random variable coming from the tossing of a balanced coin is a uniform probability on $\{0, 1\}$.

Computer science translation

A **randomized** function on a computer can be defined by saying that this is a function that, whenever it is executed twice by the computer, does not do the same computations or does not output the same value, even though the whole setup is completely the same and the argument of the functions are the same.

Use of randomness

Generate examples

In general, when you want to show that some function that you have made actually works, it is a good idea to **try it on some examples**.

Of course, if you choose your own examples, well you make sure that everything work quite well. So, what is a better idea is to try it with randomly chosen arguments.

Even though you don't need to convince someone that your code is working, you should always try to run your functions with randomly chosen arguments several times. If everything works as you would expect then your function is "generically" correct (generically should be understood as defined previously). There might still be some exceptional cases to take care of but overall your code is working.

Unpredictability of the generated data

There are situations where you want to avoid the use of predictable data.

For instance, **in cryptography** you really want to have as much random as possible.

Generate randomness on a computer

A priori, a computer is a completely **deterministic** object. The word **deterministic** means, in this context, that the outcome of any process should always be predictable provided that you are given the initial data. Unfortunately, generating randomness on a computer requires the exact opposite of it. We shall see two ways to overcome this difficulty.

Pseudo-random numbers

The principle here is very simple. You recursively create a **deterministic** sequence of numbers that has a very big period. Let us say you want to generate random numbers between 0 and $M-1$.

- You first need to find a data set X along with a natural map $\phi : X \to \mathbb{Z}\%M$.
- Fix a **seed** x_0 and an **iterating map** $f : X \to X$.
- Every time you need a new random number, compute $x_0 = f(x_0)$ and return $\phi(x_0)$.

If you don't generate "too many" numbers, the numbers you are generating will seem randomly chosen.

In [ ]:

```
# An example of random generation of bits
N=1000000
a=2713
b=189647
#This are the parameters of the iterator

#This is the iterator
f=lambda x:(a*x+b)%N

#This is the seed
seed=100


#Here the function phi(x) returns the sum of the digits of x mod 10
# Therefore, we have a seemingly random generation of integers between 0 and 9
def phi(x):
    sx=str(x)
    res=0
    for c in sx:
        res=(res+(int(c)%10))%10
    return res


for x in range(0,1000):
    print(phi(seed),end=" ")

    seed=f(seed)
```

In Python, every generation of random numbers relies on the **Mersenne Twister algorithm**. The original article is there :

http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/ARTICLES/mt.ps (http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/ARTICLES/mt.ps)

It is a very quick algorithm having a maximal period of $2^{19937} - 1$.

- Being a deterministic process, the whole generation can be forecasted, provided that you know the seed and the parameters of the function. This might be a problem, especially if you want to generate random data for security reasons (e.g. a token).
- Apart from this, not every pseudo-random generation leads to a 'satisfactory' pseudo-random behavior. There is a big litterature about testing the pseudo-randomness of a pseudo-random generators.
- They are usually very fast.

Environment-based random (a.k.a. truly random generators)

These methods are completely different and relies on the environment of the computer. Like before you will have a set of data X but instead of having an iterator of the form $X \to X$, you will just have $f : \Omega \to X$ where Ω is the 'environment' of the computer.

Then you need a function ϕ to obtain the random value from your given X.

Take any natural process that is supposed to be random and use a captor to create a number out of it (the weather, any thermodynamical process, some light sensitive captor, the hardware of the computer, the time...).

In [ ]:

```
# Here is an example with the time

#A module that allows you to access the time
import time

# A big integer out of
time.time()
```

In [ ]:

```
# frand returns the phi (like before)
def frand():
    return phi(int(1000000*time.time()))


for x in range(0,1000):
     print(frand(),end=" ")
```

In Python, you have (in the **secure** module) this kind of truly random generators. It is based on a mix of different data related to the current state of the hardware (memory use, time, heat,...).

- In general these are completely unforecastable generator (unless you choose a very bad way to create a data out of it).
- You really need to test the actual randomness of the process you are using to create your random data.
- It is in general slower than a pseudo-random generator process.
- It is more difficult to actually generate uniform random variables out of this process. The general idea being that you have absolutely no control on the raw data.

Pseudo-random generators vs environment-based generators

If you need a big sample of random numbers independently chosen to test an algorithm, you want **pseudo-random generators**.

If you just need a very big number that is supposed to be the beginning of a cryptosystem, you should consider using **environment-based generators**.

Las Vegas vs. Montecarlo algorithms

There are different kinds of algorithms using randomness. Among these, two of them are usually distinguished. The difference between both is regarding the ouptut you get.

A **Las Vegas algorithm** is an algorithm which gives the correct answer to the problem (if it exists), but in doing so it uses some random values in the procedure.

A **Monte-Carlo algorithm** returns answers with a random amount of error. Obviously, the degree of error in Monte Carlo system decreases with the increase in resources such as data or computation models.

Las Vegas algorithms are widely used either to get an average complexity (quicksort), or use specific theorems using random variables (Pollard rho algorithm).

Monte-Carlo algorithms are used to drastically reduce the complexity of algorithms. Instead of having a heavy deterministic algorithm answering a question with a 100% certitude, you would rather have a very small probability of mistake but a very light algorithm. One of the main applications of Monte Carlo methods in learning systems is to draw samples from some probability distribution that represents a dataset. This is known as Monte Carlo sampling and has been widely used to solve highly complex data estimation problems.

Examples of randomized algorithms

Quicksort

The quicksort algorithm is a divide and conquer type of algorithm and is used to sort lists. We already saw it before.

- First you chose a pivot x in L
- You divide the list L in three sublists, the list L_m of elements of L which are below x, the list L_e of elements of L which are equal to x and the list L_g of elements of L which are greater than L
- Recursively sort L_m and L_g
- Return $sort(L_m) + L_e + sort(L_g)$.

The more you equally divide your list the faster is the algorithm. One thing that is still not said is "how do you choose the pivot?". If you chose the first element of the list then your worst case might appear more often than you think since it is sorted lists. Therefore you want to use the randomness to just do **as if** the list was randomly chosen. In this case this means that you don't chose the first element nor the last but a random one inside the list.

This is a **Las-Vegas algorithm**.

In [ ]:

```
count=0
def quicksort(L):
    global count
    if len(L)<=1:
        return L
    else:
        x=L[0]
        Lm=[]
        Le=[]
        Lg=[]
        for u in L:
            count+=1
            if u<x:
                Lm.append(u)
            elif u>x:
                Lg.append(u)
            else:
                Le.append(u)

        return quicksort(Lm)+Le+quicksort(Lg)


countr=0
def quicksortr(L):
    global countr
    if len(L)<=1:
        return L
    else:
        x=L[random.randrange(0,len(L))]
        Lm=[]
        Le=[]
        Lg=[]
        for u in L:
            countr+=1
            if u<x:
                Lm.append(u)
            elif u>x:
                Lg.append(u)
            else:
                Le.append(u)
        return quicksortr(Lm)+Le+quicksortr(Lg)
```

In [ ]:

```
# L is sorted
L=[i for i in range(0,100)]
count=0
countr=0
quicksort(L)
quicksortr(L)
print("quicksort deterministic pivot complexity : ",count)
print("quicksort random pivot complexity : ",countr)
```

Montecarlo method to compute area

It relies on the fact that you can always compute area using the law of big numbers :

$$\frac{\text{card}\left\{1 \leq n \leq N \mid X_{B,n} \in A\right\}}{N} \rightarrow_N \frac{Area(A)}{Area(B)}$$

The method is quite simple, you want to compute the area of a set A. Identify a bigger subset B containing A whose you know the area and randomly (and **independently**) chose some elements of B. When you have enough random elements of B, it is probable that the number of times these elements are in A divided by the number of random elements is close to $\frac{Area(A)}{Area(B)}$.

In [ ]:

```
import math
```

In [ ]:

```
#Here is an example where I compute the area of a disk of radius 1 (i.e. pi) usi
ng Monte-Carlo method
# The area of the disk is the quantity to compute.
# B will be the square [-1,1]x[-1,1] of area 4.


piapprox=0 #Will contain the number of points being in the circle at each step
N=2000 #Number of total elements of B we are looking at

#Just drawing a frame here
reset()
hideturtle()
speed(0)
tracer(1000)
penup()
setposition(-200,-200)
pendown()
setposition(200,-200)
setposition(200,200)
setposition(-200,200)
setposition(-200,-200)
penup()
tracer(1)




for i in range(0,N):
    tracer(100000)
    x=2*random.random()-1 #This is a random element in [-1,1]
    y=2*random.random()-1 #This is a random element in [-1,1]
    if x*x+y*y<=1: #Check if (x,y) is in the disk
        piapprox+=1  #Add 1 to the number of elements being in the circle
        drawpoint((4*x,4*y)) #Add the point to the drawing
    else:
        pencolor('red')
        drawpoint((4*x,4*y))#Add the point to the drawing but in red
        pencolor('black')
    if i%100==0:
        print("with {} points we get the following approximation of pi : {}".for
mat(i+1,4*piapprox/(i+1)))
        tracer(1)
        tracer(100000)
```