Institut national supérieur du professorat et de l'éducation
Académie de Grenoble

Aléatoire en algorithmique

Emmanuel Beffara

M1 MEEF maths 2023-2024, UGA

Introduction

L’aléatoire, pour quoi faire?

Le module `random` de Python fournit des nombres aléatoires.

Pour quoi faire?

- Simuler des expériences aléatoires.
- Obtenir rapidement des résultats approchés à des problèmes difficiles.
- Obtenir des résultats exacts, peut-être plus rapidement.

Aléatoire ou pas?

Quand on fait de l'algorithmique avec utilisation de l'aléatoire, on suppose toujours qu'on a un générateur de nombres aléatoires, au minimum capable de choisir un entier uniformément dans un certain intervalle.

- En général les implémentations utilisent des générateurs *pseudo-aléatoires*, processus déterministes mais avec de bonnes propriétés.
- Quand on veut réellement de l'aléatoire, on peut utiliser des dispositifs physiques qui donnent un hasard de meilleure qualité...

Tout cela n'est pas le sujet de cette séance!

Algorithmes de Monte-Carlo

Principe

Un *algorithme de Monte-Carlo* est un algorithme qui

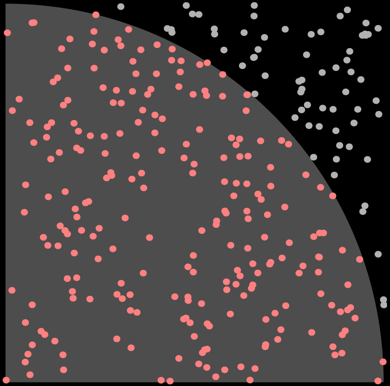
- utilise de l'aléatoire dans son exécution,
- a un temps d'exécution déterministe.

En général,

- le résultat rendu est potentiellement incorrect,
- le taux d'erreur est quantifiable précisément.

Calcul de π

On tire des points dans le carré de côté 1 de façon uniforme et on détermine s'ils sont ou non à distance 1 de l'origine:



Surface du quart de disque: $\pi/4$

- Programmer l'expérience qui calcule la proportion de points parmi n tirages qui arrivent dans le quart de cercle.
- Si X est la variable alétoire qui vaut 1 pour les points dans le quart de cercle et 0 pour les autres, quelle est l'espérance de X ?
  Quel est son écart-type ?
- Comment choisir n pour obtenir un écart-type donné ?

Savoir tester si un nombre entier est premier, c'est utile (par exemple: création de clés de chiffrement en cryptographie) mais pas facile à faire efficacement. Les méthodes probabilistes sont très utilisées pour cela.

> ***Théorème (Fermat).*** *Un entier p est premier si et seulement si pour tout a tel que $1 \leq a < p$, on a $a^{p-1} \equiv 1 \bmod p$.*

D'où le test suivant:

```
def test_primalite_fermat(N):
    a = randrange(1, N)
    return a ** (N-1) % N == 1
```

Si la fonction renvoie `False` alors `N` est assurément composé. Si elle renvoie `True`, alors il est peut-être premier...

Théorème. *Un entier p est premier si et seulement si les solutions de l'équation $x^2 \equiv 1 \mod p$ sont exactement 1 et -1.*

D'où le test probabiliste suivant, pour un entier n donné:

- Calculer la décomposition $n - 1 = 2^s \times q$ avec q impair.
- Tirer au hasard un entier a entre 2 et $n - 2$.
- Si $a^d \not\equiv 1$ et si pour tout $i \in [0, s-1]$ on a $a^{2^i q} \not\equiv -1$, alors n n'est pas premier.
- Sinon considérer que n est premier.

Quelques observations:

- Si on trouve que n n'est pas premier, c'est une certitude.
- Sinon, la probabilité qu'il soit composé est majorée par $1/4$.
- En répétant le test k fois, elle est majorée par $1/4^k$.

Algorithmes de Las Vegas

Un *algorithme de Las Vegas* est un algorithme qui

- utilise de l’aléatoire dans son exécution,
- donne toujours un résultat correct,
- a un temps d’exécution qui dépend des tirages aléatoires, dont on peut étudier la distribution de probabilité.

L’intérêt est d’utiliser l’aléatoire pour faire des choix qui seraient coûteux à faire de façon exacte.

Le célèbre algorithme Quicksort:

```
def tri_rapide(T):
    if len(T) <= 1:
        return
    i = choisir_indice(T)   # à préciser…
    p = partitionner(T, i)
    tri_rapide(T[:p])
    tri_rapide(T[p+1:])
```

où `partitionner(T,i)` déplace les éléments de `T` de sorte qu'il y ait d'abord les valeurs inférieures à `T[i]`, puis `T[i]` puis les valeurs supérieures, et renvoie la position où s'est retrouvé `T[i]`, en temps linéaire.

- Si le choix de `i` est mauvais, `tri_rapide` peut faire n^2 comparaisons (ex: prendre toujours le premier indice, si `T` est déjà trié au départ)

Tri rapide avec pivot aléatoire

L'algorithme Quicksort avec choix aléatoire du pivot:

```
def tri_rapide(T):
    if len(T) <= 1:
        return
    i = randrange(len(T))    # uniforme dans un intervalle
    p = partitionner(T, i)
    tri_rapide(T[:p])
    tri_rapide(T[p+1:])
```

Quel temps d'exécution?

— Intuitivement: l'espérance de `p` si `i` est choisi aléatoirement est la moitié de la longueur de `T`.
— On peut démontrer que l'espérance du nombre de comparaisons effectué est de l'ordre de $n \log n$.