Aléatoire en algorithmique

Emmanuel Beffara

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 Las Vegas

Principe

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.

Tri rapide

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 $n \log n$.

Algorithmes de Monte-Carlo

Principe

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

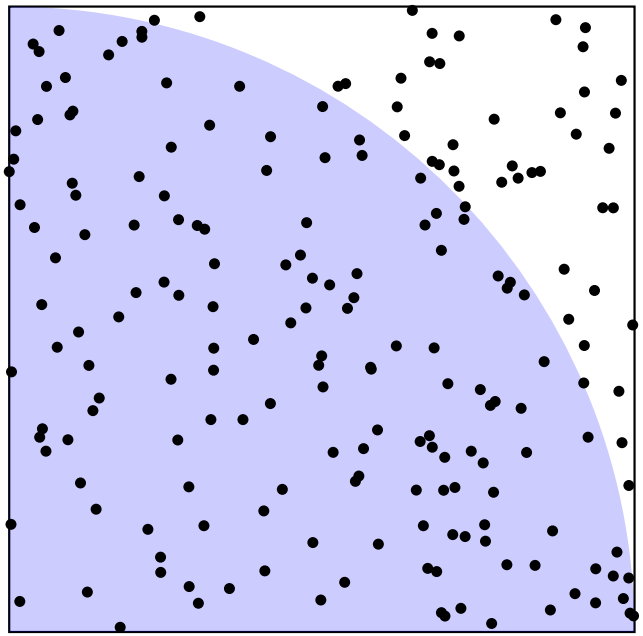
- utilise de l'aléatoire dans son exécution,
- a un temps d'exécution déterministe.

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: $\pi/4$

Calcul de π: le programme

```
from random import *
def pi_monte_carlo(nb_exp):
    succes = 0
    for i in range(nb_exp):
        x = random()     # flottants choisis uniformément
        y = random()     # dans l'intervalle [0, 1[
        if x * x + y * y <= 1:
            succes += 1
    return 4 * succes / nb_exp
```

- Le temps d'exécution est proportionnel à `nb_exp`.
- Si `random` est uniforme, l'espérance de `pi_monte_carlo(n)` est π.
- Quelle distribution de probabilité pour la valeur de `pi_monte_carlo(n)`?

Tests de primalité

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).*** *Si p est un nombre premier alors pour tout a tel que $1 \leq a < p$, on a $a^{p-1} \equiv 1 \bmod p$.*

```
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...

Tests de primalité: Miller-Rabin

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

```
def test_primalite_miller_rabin(n):
    s, q = decompose(n - 1)    # pour avoir n - 1 = 2**s * q
    a = randrange(1, n)
    if a ** q % n == 1:
        return True
    for i in range(s):
        if a ** (2**i * q) % n == n - 1:
            return True
    return False
```

- Si la réponse est `False`, alors `n` est composé.
- Si c'est `True`, la probabilité qu'il soit composé est majorée par $1/4$.
- En répétant le test k fois, elle est majorée par $1/4^k$.

Calcul de coupe minimale

À vous de jouer pour l'algorithme de Karger!