Ecole Polytechnique, Cycle Ingénieur (2A)
MAP471A - Problem solving en mathématiques appliquées
Enseignants : Lucas Gerin (mail) (mailto:lucas.gerin@polytechnique.edu) Teddy Pichard (mail) (mailto:teddy.pichard@polytechnique.edu) Nicole Spillane (mail) (mailto:nicole.spillane@polytechnique.edu)

Algorithmes randomisés : autour de Quicksort

```
# css style
from IPython.core.display import HTML
def css_styling():
    styles = open("./style/custom2.css").read()
    return HTML(styles)
css_styling()
```

```
# load the libraries
import matplotlib.pyplot as plt # 2D plotting library
import numpy as np              # package for scientific computing
import random
%matplotlib inline
```

Table des matières

Introduction

L'objectif de ce TP est d'illustrer la puissance des algorithmes probabilistes (ou *randomisés*) pour résoudre des problèmes déterministes. Il existe deux grandes classes d'algorithmes randomisés :

- Les algorithmes de Las Vegas ont un temps d'exécution aléatoire mais donnent une réponse exacte ;
- Les algorithmes de Monte-Carlo ont un temps d'éxecution qui est déterministe (ou aléatoire mais borné) et donne une réponse approchée.

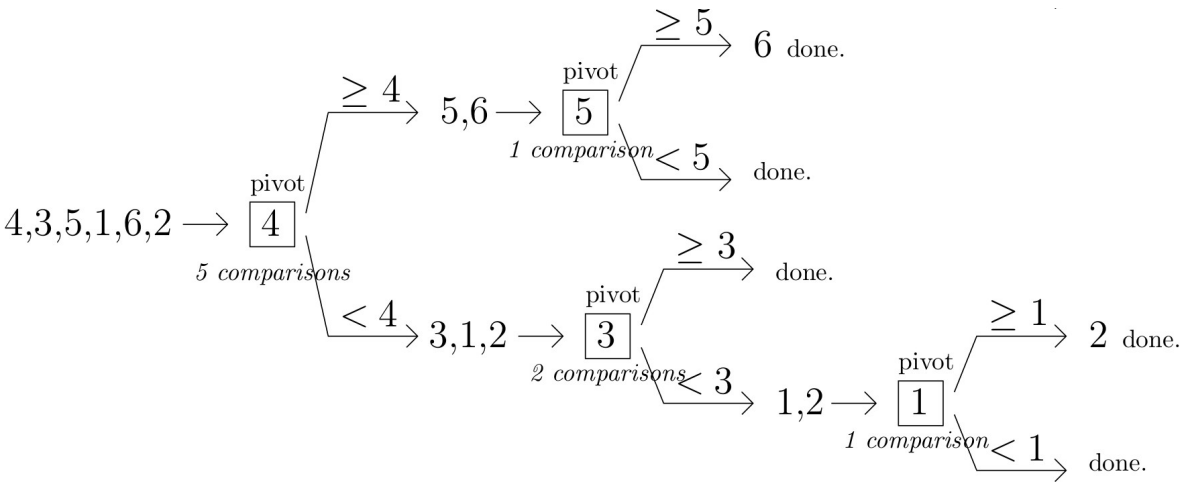
Quicksort

`Quicksort` est l'un des algorithmes les plus populaires pour trier une liste de nombres $L = x_1, \dots, x_n$. L'objectif de cette section est d'analyser le comportement de `Quicksort` (et de ses variantes) lorsque les x_i sont aléatoires et i.i.d.

Définissons formellement la sortie $\text{Quicksort}(L)$ de l'algorithme : c'est la liste L triée par ordre croissant. L'idée de QuickSort est d'utiliser la stratégie *Diviser-pour-régner* et se décompose récursivement de la façon suivante :

1. Si L est vide ou réduite à un seul élément, $\text{Quicksort}(L) = L$.
2. Sinon le 1er élément x_1 est appelé *pivot* de la liste. On compare chacun des $n-1$ autre éléments à x_1 pour obtenir deux sous-listes
$$L_< = \{x_j ; j \geq 2, x_j < x_1\},$$
$$L_\geq = \{x_j ; j \geq 2, x_j \geq x_1\}.$$
3. On applique récursivement l'algorithme à $L_<$ et $L_\geq$: on renvoie la concaténation des trois listes
$$\text{Quicksort}(L_<), \qquad [x_1], \qquad \text{Quicksort}(L_\geq).$$

Voici schématiquement l'exécution de Quicksort pour $L = [4, 3, 5, 1, 6, 2]$:



On note $C(x_1, x_2, \dots, x_n)$ le nombre de comparaisons nécessaires pour trier la liste $x_1, \dots, x_n$. L'exemple ci-dessus démontre que
$$C(4, 3, 5, 1, 6, 2) = 5 + 1 + 2 + 1 = 9.$$

> **Remark.** La complexité *dans le pire cas* de Quicksort est quadratique : il existe $c, C > 0$ telles que
> $$cn^2 \leq \max_{x_1,\dots,x_n} C(x_1, x_2, \dots, x_n) \leq Cn^2.$$
> *(Ici le seul coût considéré dans la complexité est le nombre de comparaisons de deux réels. On néglige donc les coûts d'accès et écriture en mémoire.)*

Preuve : Lorsque les $x_1, x_2, \dots, x_n$ sont déjà triés l'algorithme effectue

$$(n-1) + (n-2) + \cdots + 1 \sim \tfrac{1}{2}n^2 \text{ comparaisons,}$$

d'où l'inégalité de gauche.

Pour l'inégalité de droite, montrons par récurrence que

$$\max_{x_1,\dots,x_n} C(x_1, x_2, \dots, x_n) \leq 2n^2.$$

Pour $n = 1$ c'est ok. Ensuite

$$\begin{aligned} C(x_1, x_2, \dots, x_n) &= (n-1) + C(\{x_j \text{ tels que } x_j < x_1\}) + C(\{x_j \text{ tels que } x_j \geq x_1\}) \\ &\leq (n-1) + \max_{0\leq r\leq n-1} \left\{ \max_{y_1,\dots,y_r} C(y_1, \dots, y_r) + \max_{y_{r+1},\dots,y_{n-1}} C(y_{r+1}, \dots, y_{n-1} \right. \\ &\leq (n-1) + \max_{0\leq r\leq n-1} 2r^2 + 2(n-1-r)^2 \\ &\leq (n-1) + 2(n-1)^2/2 \leq 2n^2. \end{aligned}$$

L'objectif est d'évaluer la variable aléatoire $C_n := C(X_1, \dots, X_n)$ lorsque les données sont des variables aléatoires $X_1, \dots, X_n$ i.i.d. uniformes sur $[0, 1]$. C'est donc l'analyse en complexité moyenne de `Quicksort` .

On peut démontrer les choses suivantes :

1. On a l'identité en loi :
$$\text{card}(L_<) \overset{\text{(loi)}}{=} \text{card}(L_\geq) \overset{\text{(loi)}}{=} \text{Uniforme dans } \{0, 1, \dots, n-1\}.$$
2. Conditionnellement à X_1, les X_j dans $L_<$ sont répartis uniformément dans l'intervalle $[0, X_1]$ et sont indépendants. (On bien sûr le même énoncé pour $L_\geq$.)

On en déduit donc que le nombre aléatoire de comparaisons C_n pour une liste de taille n vérifie

$$\begin{aligned} C_0 &= 0, \\ C_1 &= 0, \\ \text{Pour tout } n \geq 2, \quad C_n &\overset{\text{(loi)}}{=} n - 1 + C'_{U_n-1} + C''_{n-U_n}, \qquad (\star) \end{aligned}$$

où U_n est une uniforme sur $\{1, \dots, n\}$ et pour tout k, C_k, C'_k, C''_k ont même loi, sont indépendantes entre elles et indépendantes de U_n.

Do it yourself.

1. Ecrire une fonction `Uniform(a,b)` qui renvoie un tirage d'une v.a. uniforme dans $\{a, a+1, \dots, b\}$.
   (Rappel : `np.random.rand()` *renvoie un réel uniforme dans* $[0, 1]$.
2. Utiliser la relation $(\star)$ pour simuler des variables aléatoires C_n pour plusieurs valeurs de n, et tracer un graphique représentant les résultats $n \mapsto C_n$. (Prendre n entre 1 et 2000.)
   *(**Attention!** Il n'y a pas besoin de réellement implémenter Quicksort, simplement de simuler* C_n*.)*
3. On peut démontrer (*) que
$$\mathbb{E}[C_n] = 2n\log(n) - 2.85n + o(n).$$
   Comparer avec vos résultats.
   Pour visualiser on vous suggère de tracer une grande quantité de points (n, C_n) *sur le même graphique, ainsi que* $n \mapsto 2n\log(n) - 2.85n$*. Essayer jusqu'à au moins*

$n = 2000$.

(*) Référence: Voir par exemple p.37 dans M.Mitzenmacher, E.Upfal (2005). Probability and Computing. *Cambridge University Press*.

Remark. La conséquence des cette analyse est que le comportement moyen en $2n\log(n)$ est bien inférieur au comportement dans le pire cas en n^2. C'est pourquoi dans certaines libraires l'algorithme Quicksort est implémenté avec une étape préliminaire consistant à mélanger uniformément la liste $x_1, \dots, x_n$.
Concrètement, il suffit simplement de choisir le pivot uniformément dans la liste à chaque itération de l'algorithme.

Une amélioration : Median-of-Three Quicksort

Intuitivement, Quicksort est plus efficace lorsque $L_<$ et $L_\geq$ sont de taille équivalente. Pour exploiter cette idée, une variante assez utilisée dans la pratique consiste à prendre la médiane de X_1, X_2, X_3 comme pivot au lieu de X_1. C'est *Median-of-Three Quicksort*.
L'objectif est d'illustrer l'efficacité de cette variante.

Do it yourself. 1. Les variables aléatoires X_1, X_2, X_3 sont indépendantes et uniformes sur $[0,1]$. Quelle est la loi du nombre de comparaisons nécessaires pour trouver la médiane de X_1, X_2, X_3? On note $\mathcal{M}$ une réalisation de cette variable, écrire une fonction qui simule $\mathcal{M}$.

2. Déterminer l'analogue de $(\star)$ pour Median-of-Three Quicksort. En déduire des simulations des variables aléatoires D_n définies par le nombre de comparaisons pour Median-of-Three Quicksort.
   (Attention : ne pas oublier le coût de calcul de la médiane.)
3. Comparer vos simulations avec Quicksort classique et avec le résultat en moyenne : Pour Median-of-Three Quicksort on a
   $$\mathbb{E}[D_n] = \tfrac{12}{7} n\log(n) - 1.827n + o(n).$$
   (voir Th.3.5 dans Sedgewick, R., & Flajolet, P. (2013). An introduction to the analysis of algorithms.)

Answers.
1.
2.

Encore une variante : Diviser en 3-pour-régner

On va chercher à évaluer l'efficacité de Quicksort si on divise la liste en 3 listes au lieu de 2, en utilisant les pivots X_1, X_2. Pour cela on note (I_n, J_n) une paire uniforme parmi les $\binom{n}{2}$ paires de la forme $\{1 \leq i < j \leq n\}$.

Do it yourself.

1. Ecrire une fonction `PaireIJ` qui tire au sort (I_n, J_n).
2. Ecrire une fonction `NbComparaisons(i,j,n)` qui simule la variable aléatoire suivante : (i, j étant fixés) le coût pour trier i, j, X_3 lorsque X_3 est une uniforme sur $\{1, \dots, n\}$. On note $\mathcal{N}_n(i, j)$ une réalisation de cette v.a.
3. Ecrire l'analogue de $(\star)$ pour la stratégie diviser en 3-pour-régner. En déduire des simulations des variables aléatoires E_n. Comparer avec Quicksort classique et median-of-three.

Answers.

2.
3.

Do it yourself. Quelle est votre conclusion pour Quicksort avec la stratégie "Diviser en 3"

Answers.

Implémentation

On va comparer les simulations avec des implémentations réelles de Quicksort.

Do it yourself.

1. Ecrire une fonction `QuicksortEnVrai()` qui trie une liste selon Quicksort, et qui renvoie la liste triée ainsi que le nombre de comparaisons effectuées. Par exemple :

```
>QuicksortEnVrai([5,2,3,10,100,4])
[[2, 3, 4, 5, 10, 100], 9]
```

2. Vérifier sur des simulations que les nombres de comparaisons sont comparables avec vos simulations précédentes.
3. Mêmes questions avec "Diviser en 3-pour-régner"

Algorithmes randomisés pour la médiane

Un algorithme naïf pour trouver la médiane d'une liste $\{X_1, \dots, X_n\}$ consiste à trier la liste et à lire ensuite le $\lfloor n/2 \rfloor$-ème élément. D'après ce qui précède cela coûte $\text{constante} \times n \log(n)$ comparaisons.

Nous allons utiliser la stratégie randomisée pour trouver la médiane avec un nombre linéaire de comparaisons.

QuickMedian

Do it yourself.

1. En s'inspirant de la stratégie QuickSort, décrire un algorithme efficace QuickMedian qui renvoie le k-ème plus petit élément d'une liste $\{X_1, \dots, X_n\}$ (sans trier toute la liste!).
2. Ecrire la relation de récurrence pour le nombre de comparaisons $S_{n,k}$ effectuées par QuickMedian.

Answers.

Do it yourself. 1. Ecrire une fonction qui simule la variable $S_{n,k}$.

2. Afficher plusieurs simulations pour $k = \lfloor n/2 \rfloor$ (calcul de la médiane). On devrait avoir
$$\mathbb{E}[S_{n,\lfloor n/2 \rfloor}] = n\,(2 + 2\log(2)) + o(n) = n \times 3.3863... + o(n)$$
(Voir ce lien. (https://11011110.github.io/blog/2007/10/09/blum-style-analysis-of.html))