Algorithmique Probabiliste

Philippe Duchon

LaBRI - ENSEIRB-Matméca - Université de Bordeaux

2014-15

Le sujet

- Utilisation *explicite* de tirages aléatoires dans des algorithmes

Le sujet

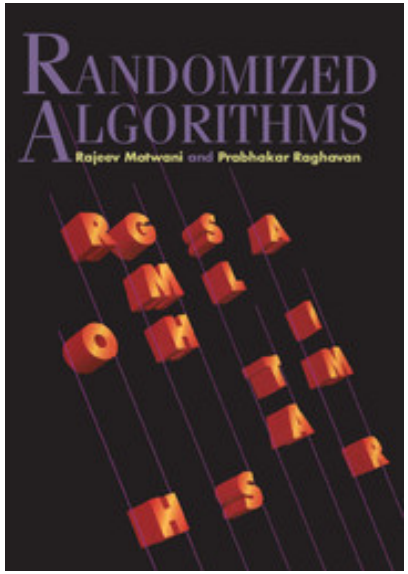
- Utilisation *explicite* de tirages aléatoires dans des algorithmes
- Analyse des complexités des algorithmes

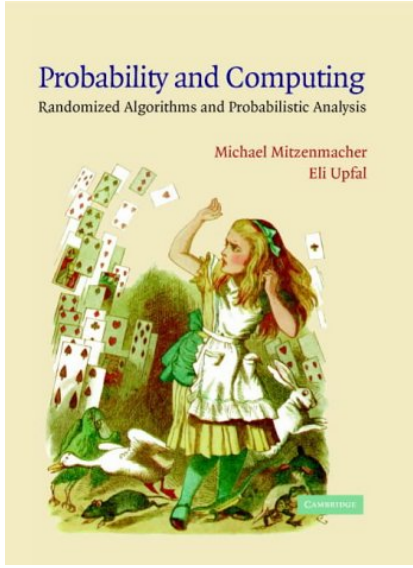
Le sujet

- Utilisation *explicite* de tirages aléatoires dans des algorithmes
- Analyse des complexités des algorithmes
- Quelques principes de conception d'algorithmes

Bibliographie expresse

- R. Motwani, P. Raghavan, *Randomized Algorithms*, Cambridge University Press, 1995.
- M. Mitzenmacher, E. Upfal, *Probability and Computing : Randomized Algorithms and Probabilistic Analysis* - Cambridge University Press, 2002.





Algorithme “classique”

- Un algorithme “classique” est **totalement déterministe**

Algorithme “classique”

- Un algorithme “classique” est **totalement déterministe**
- Étant donnés un algorithme A et une donnée x,

Algorithme “classique”

- Un algorithme “classique” est **totalement déterministe**
- Étant donnés un algorithme A et une donnée x,
  - le résultat $A(x)$ est entièrement déterminé ;

Algorithme “classique”

- Un algorithme “classique” est **totalement déterministe**
- Étant donnés un algorithme A et une donnée x,
  - le résultat $A(x)$ est entièrement déterminé ;
  - la séquence de calculs aussi ;

Algorithme “classique”

- Un algorithme “classique” est **totalement déterministe**
- Étant donnés un algorithme A et une donnée x,
  - le résultat $A(x)$ est entièrement déterminé ;
  - la séquence de calculs aussi ;
  - en particulier, *il n'y a aucun intérêt à relancer le même calcul*.

Algorithme “probabiliste”

(Synonyme : algorithme randomisé)

- L’algorithme “tire à pile ou face” **et peut agir différemment en fonction du résultat**

Algorithme “probabiliste”

(Synonyme : algorithme randomisé)

- L’algorithme “tire à pile ou face” **et peut agir différemment en fonction du résultat**
- En conséquence, pour un algorithme et une donnée fixés,

Algorithme “probabiliste”

(Synonyme : algorithme randomisé)

- L'algorithme “tire à pile ou face” **et peut agir différemment en fonction du résultat**
- En conséquence, pour un algorithme et une donnée fixés,
  - le résultat $A(x)$ est une *variable aléatoire*

Algorithme “probabiliste”

(Synonyme : algorithme randomisé)

- L'algorithme “tire à pile ou face” **et peut agir différemment en fonction du résultat**
- En conséquence, pour un algorithme et une donnée fixés,
  - le résultat $A(x)$ est une *variable aléatoire*
  - de même que la séquence de calculs, et toutes les grandeurs associées (temps, mémoire, etc)

Algorithme “probabiliste”

(Synonyme : algorithme randomisé)

- L'algorithme “tire à pile ou face” **et peut agir différemment en fonction du résultat**
- En conséquence, pour un algorithme et une donnée fixés,
  - le résultat $A(x)$ est une *variable aléatoire*
  - de même que la séquence de calculs, et toutes les grandeurs associées (temps, mémoire, etc)
  - en particulier, *il peut être intéressant de lancer plusieurs fois le même algorithme sur les mêmes données (et de “prendre le meilleur résultat”)*

Ça pose des quetions. . .

- Comment **décrire** des algorithmes probabilistes ? quel est le modèle ?

Ça pose des quetions. . .

- Comment **décrire** des algorithmes probabilistes ? quel est le modèle ?
- Comment **raisonner** sur de tels algorithmes ?

Ça pose des quetions. . .

- Comment **décrire** des algorithmes probabilistes ? quel est le modèle ?
- Comment **raisonner** sur de tels algorithmes ?
- C'est quoi, un "bon" algorithme probabiliste ?

Ça pose des quetions...

- Comment **décrire** des algorithmes probabilistes ? quel est le modèle ?
- Comment **raisonner** sur de tels algorithmes ?
- C'est quoi, un "bon" algorithme probabiliste ?
- Quelles sont les "bonnes" grandeurs à évaluer ?

Ça pose des quetions...

- Comment **décrire** des algorithmes probabilistes ? quel est le modèle ?
- Comment **raisonner** sur de tels algorithmes ?
- C'est quoi, un "bon" algorithme probabiliste ?
- Quelles sont les "bonnes" grandeurs à évaluer ?
- (par l'exemple)

Ça pose des quetions...

- Comment **décrire** des algorithmes probabilistes ? quel est le modèle ?
- Comment **raisonner** sur de tels algorithmes ?
- C'est quoi, un "bon" algorithme probabiliste ?
- Quelles sont les "bonnes" grandeurs à évaluer ?
- (par l'exemple)
- (ce qui implique de refaire un peu de probabilités)

Un exemple frappant : égalité de deux chaînes

- Deux machines M et M' détiennent chacune une chaîne binaire, s et s', de même longueur n ; on souhaite déterminer si oui ou non on a $s = s'$

Un exemple frappant : égalité de deux chaînes

- Deux machines M et M' détiennent chacune une chaîne binaire, s et s', de même longueur n ; on souhaite déterminer si oui ou non on a $s = s'$
- On s'intéresse à la *complexité de communication* : le temps de calcul n'importe pas, seule compte la longueur des messages envoyés

Un exemple frappant : égalité de deux chaînes

- Deux machines M et M' détiennent chacune une chaîne binaire, s et s', de même longueur n ; on souhaite déterminer si oui ou non on a $s = s'$
- On s'intéresse à la *complexité de communication* : le temps de calcul n'importe pas, seule compte la longueur des messages envoyés
- Pour simplifier, on suppose un protocole à 1 message :
  - M calcule : $m = f(s)$, et envoie m à M'
  - M' calcule : $r = g(m, s')$
  - le résultat est *correct* si r vaut `[s==s']`
  - la *complexité de communication* du protocole est la longueur du message envoyé, $C = \ell(m)$

Solution déterministe

- Un protocole (un choix de fonctions f et g) est correct si, pour tout n, et pour toutes chaînes s, s' de longueur n, le résultat r obtenu est correct

Solution déterministe

- Un protocole (un choix de fonctions f et g) est correct si, pour tout n, et pour toutes chaînes s, s' de longueur n, le résultat r obtenu est correct
- **Théorème :** pour tout protocole correct, pour tout n, il existe une instance (s, s') pour laquelle M envoie au moins n bits.

Solution déterministe

- Un protocole (un choix de fonctions f et g) est correct si, pour tout n, et pour toutes chaînes s, s' de longueur n, le résultat r obtenu est correct
- **Théorème :** pour tout protocole correct, pour tout n, il existe une instance (s, s') pour laquelle M envoie au moins n bits.
- **Preuve ?**

Solution déterministe

- Un protocole (un choix de fonctions f et g) est correct si, pour tout n, et pour toutes chaînes s, s' de longueur n, le résultat r obtenu est correct
- **Théorème :** pour tout protocole correct, pour tout n, il existe une instance (s, s') pour laquelle M envoie au moins n bits.
- **Preuve ?**
- En d'autres termes : en déterministe, *il n'y a pas de solution intrinsèquement plus intelligente que de demander à M d'envoyer sa chaîne s tout entière à M'*

Solution probabiliste : description

(On interprète les chaînes s et s' comme de grands entiers, de l'ordre de 2^n)

- M **choisit aléatoirement et uniformément** un nombre entier p, premier, compris entre $n^2/2$ et n^2
- M calcule $x = s \mod p$, et envoie $m = (x, p)$ à M'
- M' calcule $x' = s' \mod p$, et répond `x==x'`

Solution probabiliste : analyse

- x et p sont des entiers au plus égaux à n^2, donc de longueur au plus $2\log_2(n)$: **la complexité de communication est de** $4\log_2(n)$

Solution probabiliste : analyse

- x et p sont des entiers au plus égaux à n^2, donc de longueur au plus $2\log_2(n)$: **la complexité de communication est de** $4\log_2(n)$
- **On peut parfois avoir un résultat incorrect :** cela arrive exactement si $s \neq s'$, mais que le nombre entier p choisi est un **diviseur** de $|s - s'|$

Solution probabiliste : analyse

- x et p sont des entiers au plus égaux à n^2, donc de longueur au plus $2\log_2(n)$: **la complexité de communication est de** $4\log_2(n)$
- **On peut parfois avoir un résultat incorrect :** cela arrive exactement si $s \neq s'$, mais que le nombre entier p choisi est un **diviseur** de $|s - s'|$
- Si $s = s'$, la *probabilité de résultat incorrect* est d'exactement 0 ;

Solution probabiliste : analyse

- x et p sont des entiers au plus égaux à n^2, donc de longueur au plus $2\log_2(n)$: **la complexité de communication est de** $4\log_2(n)$
- **On peut parfois avoir un résultat incorrect :** cela arrive exactement si $s \neq s'$, mais que le nombre entier p choisi est un **diviseur** de $|s - s'|$
- Si $s = s'$, la *probabilité de résultat incorrect* est d'exactement 0 ;
- Si $s \neq s'$, la *probabilité de résultat incorrect* est exactement :

$$p_{s,s'} = \frac{\text{nombre de diviseurs premiers, entre } n^2/2 \text{ et } n^2, \text{ de } |s - s'|}{\text{nombre d'entiers premiers entre } n^2/2 \text{ et } n^2}$$

Solution probabiliste : analyse

- x et p sont des entiers au plus égaux à n^2, donc de longueur au plus $2\log_2(n)$: **la complexité de communication est de** $4\log_2(n)$
- **On peut parfois avoir un résultat incorrect :** cela arrive exactement si $s \neq s'$, mais que le nombre entier p choisi est un **diviseur** de $|s - s'|$
- Si $s = s'$, la *probabilité de résultat incorrect* est d'exactement 0 ;
- Si $s \neq s'$, la *probabilité de résultat incorrect* est exactement :

$$p_{s,s'} = \frac{\text{nombre de diviseurs premiers, entre } n^2/2 \text{ et } n^2, \text{ de } |s - s'|}{\text{nombre d'entiers premiers entre } n^2/2 \text{ et } n^2}$$

- Le numérateur est trivialement inférieur à n (et même à $n/2\log_2(n)$)

Solution probabiliste : analyse

- x et p sont des entiers au plus égaux à n^2, donc de longueur au plus $2\log_2(n)$: **la complexité de communication est de** $4\log_2(n)$
- **On peut parfois avoir un résultat incorrect :** cela arrive exactement si $s \neq s'$, mais que le nombre entier p choisi est un **diviseur** de $|s - s'|$
- Si $s = s'$, la *probabilité de résultat incorrect* est d'exactement 0 ;
- Si $s \neq s'$, la *probabilité de résultat incorrect* est exactement :

$$p_{s,s'} = \frac{\text{nombre de diviseurs premiers, entre } n^2/2 \text{ et } n^2, \text{ de } |s - s'|}{\text{nombre d'entiers premiers entre } n^2/2 \text{ et } n^2}$$

- Le numérateur est trivialement inférieur à n (et même à $n/2\log_2(n)$)
- Le dénominateur est (d'après le théorème des nombres premiers) de l'ordre de $\frac{n^2}{4\ln(n)}$

Solution probabiliste : analyse

- x et p sont des entiers au plus égaux à n^2, donc de longueur au plus $2\log_2(n)$: **la complexité de communication est de** $4\log_2(n)$
- **On peut parfois avoir un résultat incorrect :** cela arrive exactement si $s \neq s'$, mais que le nombre entier p choisi est un **diviseur** de $|s - s'|$
- Si $s = s'$, la *probabilité de résultat incorrect* est d'exactement 0 ;
- Si $s \neq s'$, la *probabilité de résultat incorrect* est exactement :

$$p_{s,s'} = \frac{\text{nombre de diviseurs premiers, entre } n^2/2 \text{ et } n^2, \text{ de } |s - s'|}{\text{nombre d'entiers premiers entre } n^2/2 \text{ et } n^2}$$

- Le numérateur est trivialement inférieur à n (et même à $n/2\log_2(n)$)
- Le dénominateur est (d'après le théorème des nombres premiers) de l'ordre de $\frac{n^2}{4\ln(n)}$
- **Conclusion :** le protocole probabiliste a, *pour toute instance* (s, s'), une probabilité inférieure à $2\ln(2)/n$ de se tromper.

Comparons. . .

On prend le cas $n = 10^9 \sim 2^{30}$: un gigabit

- **Déterministe :** on est absolument certain du résultat ; M doit envoyer 1 gigabit à M'.

Comparons. . .

On prend le cas $n = 10^9 \sim 2^{30}$: un gigabit

- **Déterministe :** on est absolument certain du résultat ; M doit envoyer 1 gigabit à M'.
- **Probabiliste :** M envoie 120 bits à M' ; en contrepartie, on accepte une probabilité d'erreur d'environ 1.4×10^{-9} (une chance sur 700 millions ; soit la probabilité d'être choisi si une personne résidant en Europe est choisie au hasard)

Comparons. . .

On prend le cas $n = 10^9 \sim 2^{30}$: un gigabit

- **Déterministe :** on est absolument certain du résultat ; M doit envoyer 1 gigabit à M'.
- **Probabiliste :** M envoie 120 bits à M' ; en contrepartie, on accepte une probabilité d'erreur d'environ 1.4×10^{-9} (une chance sur 700 millions ; soit la probabilité d'être choisi si une personne résidant en Europe est choisie au hasard)
- **En répétant deux fois le protocole probabiliste :** M envoie 240 bits à M', la probabilité d'erreur tombe à 2×10^{-18} (probabilité que le choix de **deux** personnes au hasard en Europe, vous désigne, vous **et** votre voisin de gauche)

Comparons...

On prend le cas $n = 10^9 \sim 2^{30}$: un gigabit

- **Déterministe :** on est absolument certain du résultat ; M doit envoyer 1 gigabit à M'.
- **Probabiliste :** M envoie 120 bits à M' ; en contrepartie, on accepte une probabilité d'erreur d'environ 1.4×10^{-9} (une chance sur 700 millions ; soit la probabilité d'être choisi si une personne résidant en Europe est choisie au hasard)
- **En répétant deux fois le protocole probabiliste :** M envoie 240 bits à M', la probabilité d'erreur tombe à 2×10^{-18} (probabilité que le choix de **deux** personnes au hasard en Europe, vous désigne, vous **et** votre voisin de gauche)
- (ce n'est rien d'autre qu'un schéma de hachage, explicitement aléatoire ; la propriété importante étant que, la fonction de hachage étant aléatoire, **on ne peut pas choisir deux chaînes différentes en sachant qu'elles ont la même valeur de hachage**)

L'exemple de **QuickSort**

Pour trier un tableau de n valeurs :

- Si $n \leq 1$, il est déjà trié
- Sinon :
  - choisir un *pivot* (premier élément du tableau) x
  - comparer x à chaque autre élément, et réordonner le tableau en "les plus petits que x", puis x, puis "les plus grands que x"
  - Trier récursivement "les plus petits que x" et "les plus grands que x" avec **QuickSort**

QuickSort en Python

```
def Partitionne(L,a,b,k):
    p = L[k]
    Echange(L,k,b)
    st = a
    for i in range(a,b):
        if L[i]<p:
            Echange(L,i,st)
            st = st+1
    Echange(L,st,b)
    return(st)

def QuickSortRec(L,a,b):
    if (a<b):
        k=Partitionne(L,a,b,a)
        QuickSortRec(L,a,k-1)
        QuickSortRec(L,k+1,b)
```

“Analyse” de **QuickSort**

- On regarde le **nombre de comparaisons de clés**

“Analyse” de **QuickSort**

- On regarde le **nombre de comparaisons de clés**
- “En moyenne” (si **le tableau initial est dans un ordre aléatoire uniforme**), on fait $2n\ln(n) + O(n)$ comparaisons

“Analyse” de **QuickSort**

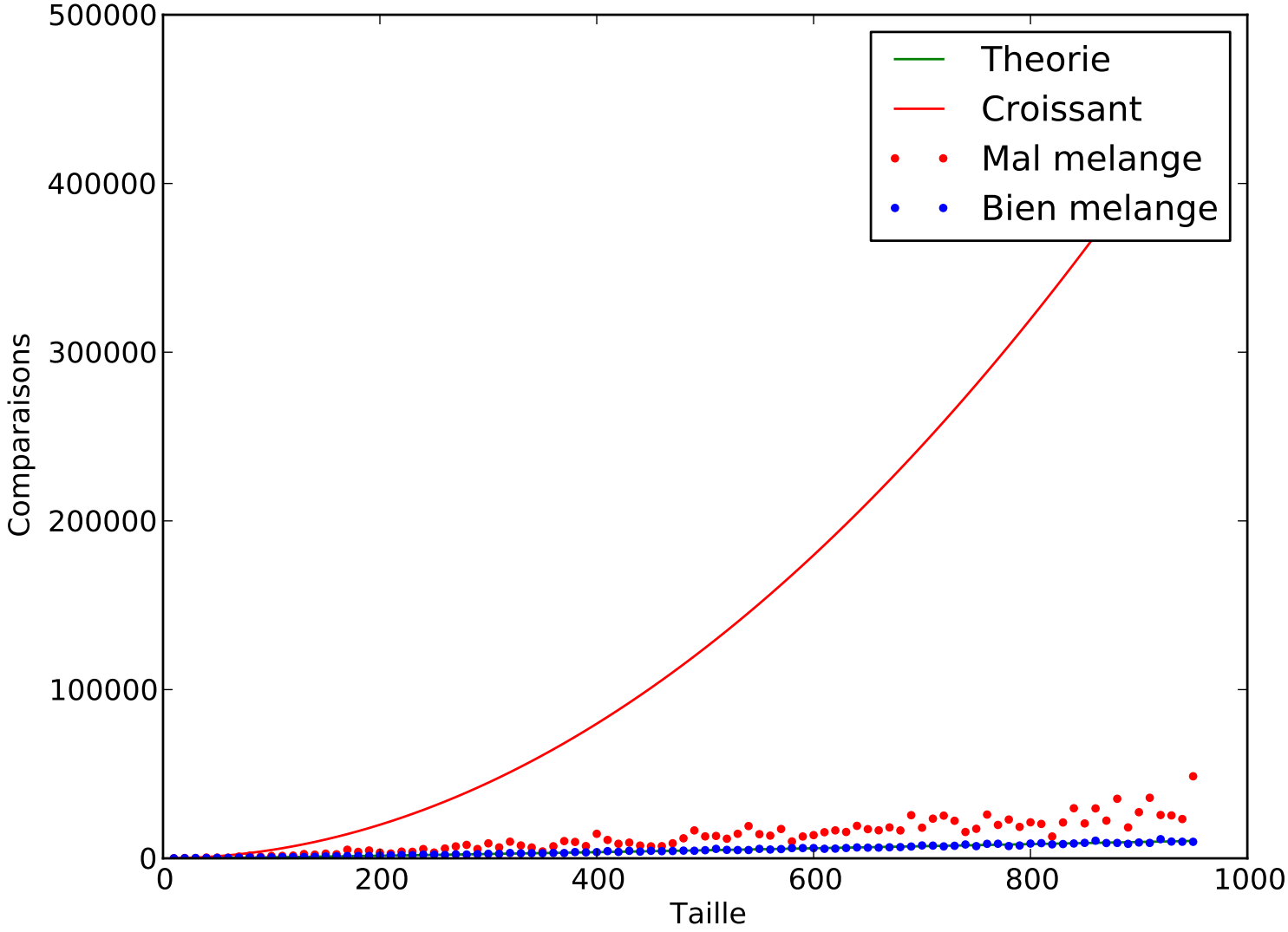
- On regarde le **nombre de comparaisons de clés**
- “En moyenne” (si **le tableau initial est dans un ordre aléatoire uniforme**), on fait $2n\ln(n) + O(n)$ comparaisons
- Mais le cas le pire est quadratique

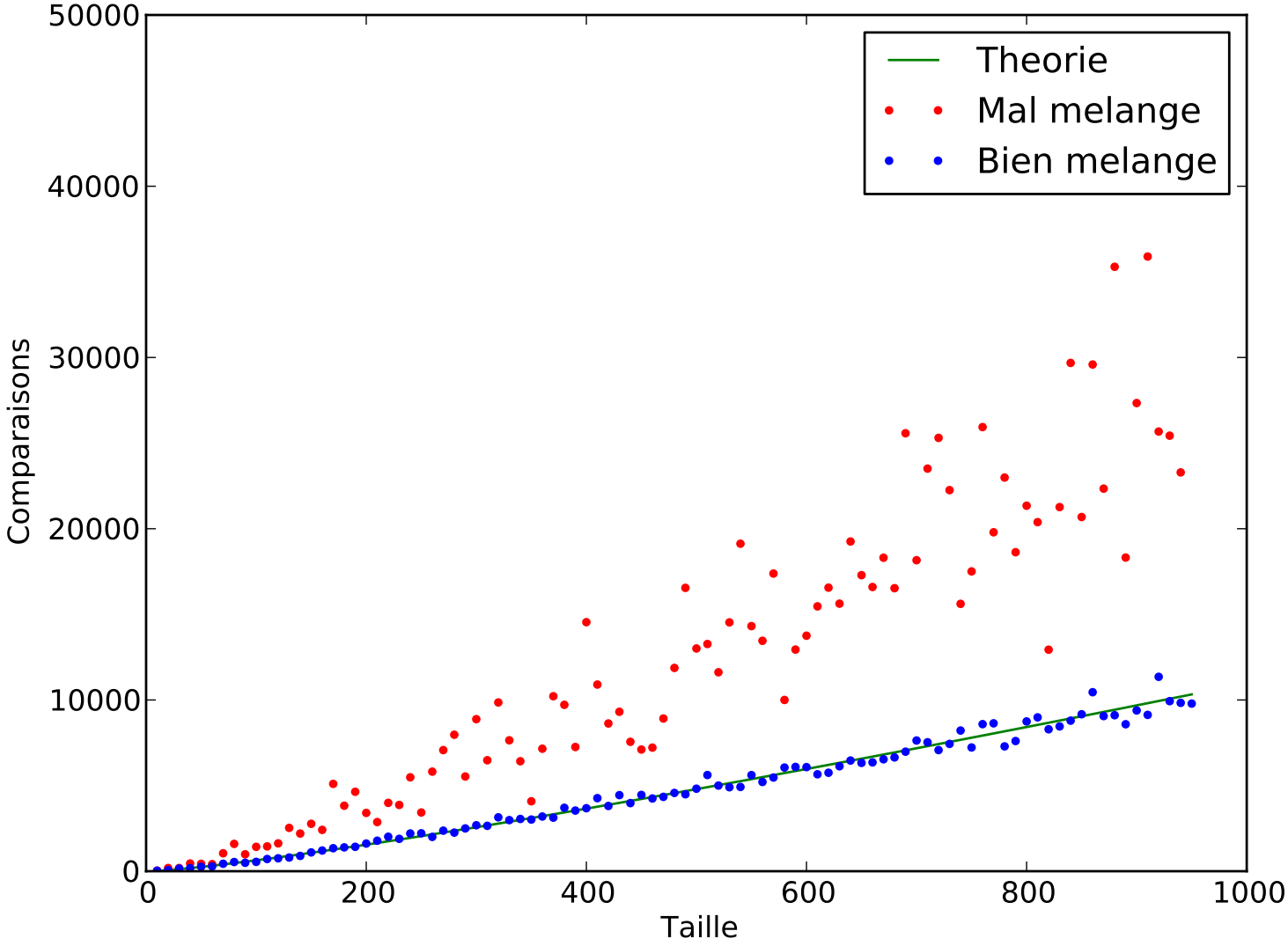
“Analyse” de **QuickSort**

- On regarde le **nombre de comparaisons de clés**
- “En moyenne” (si **le tableau initial est dans un ordre aléatoire uniforme**), on fait $2n\ln(n) + O(n)$ comparaisons
- Mais le cas le pire est quadratique
- **La mauvaise nouvelle :** un tableau **déjà trié** (ou “presque” déjà trié) est trié par **QuickSort** en temps quadratique

Quelques expériences

- Le code `Python` présenté (ou presque)
- On compte exactement les comparaisons
- Trois jeux de tests :
  - Liste triée par ordre croissant
  - Liste dans un ordre aléatoire
  - Liste croissante, “mal” mélangée ($n/10$ échanges de paires prises au hasard)


Theorie
Croissant
Mal melange
Bien melange
Comparaisons
Taille
0
100000
200000
300000
400000
500000
0
200
400
600
800
1000


Theorie
Mal melange
Bien melange
Comparaisons
Taille
0
10000
20000
30000
40000
50000
0
200
400
600
800
1000

Interprétons

- Sur des listes aléatoires, l'algorithme a vraiment l'air de se comporter souvent en $\Theta(n \ln(n))$

Interprétons

- Sur des listes aléatoires, l'algorithme a vraiment l'air de se comporter souvent en $\Theta(n \ln(n))$
- Sur des listes croissantes, il est catastrophique

Interprétons

- Sur des listes aléatoires, l'algorithme a vraiment l'air de se comporter souvent en $\Theta(n \ln(n))$
- Sur des listes croissantes, il est catastrophique
- Sur des listes "un peu mal mélangées", les performances se dégradent assez vite, et semblent assez aléatoires (grosse incertitude)

Interprétons

- Sur des listes aléatoires, l'algorithme a vraiment l'air de se comporter souvent en $\Theta(n \ln(n))$
- Sur des listes croissantes, il est catastrophique
- Sur des listes "un peu mal mélangées", les performances se dégradent assez vite, et semblent assez aléatoires (grosse incertitude)
- **Malheureusement, on ne peut pas faire confiance à l'utilisateur de l'algorithme pour ne nous faire trier que des listes aléatoires**

Et si. . .

- Idée toute bête : *et si au lieu de choisir un pivot déterministe, on le choisissait au hasard dans le tableau ?*

Et si. . .

- Idée toute bête : *et si au lieu de choisir un pivot déterministe, on le choisissait au hasard dans le tableau ?*
- On obtient une **version randomisée** de l'algorithme : à chaque exécution, on peut avoir un comportement différent

Et si. . .

- Idée toute bête : *et si au lieu de choisir un pivot déterministe, on le choisissait au hasard dans le tableau ?*
- On obtient une **version randomisée** de l'algorithme : à chaque exécution, on peut avoir un comportement différent
- (Le résultat, lui, n'est pas aléatoire : on trie toujours dans l'ordre croissant)

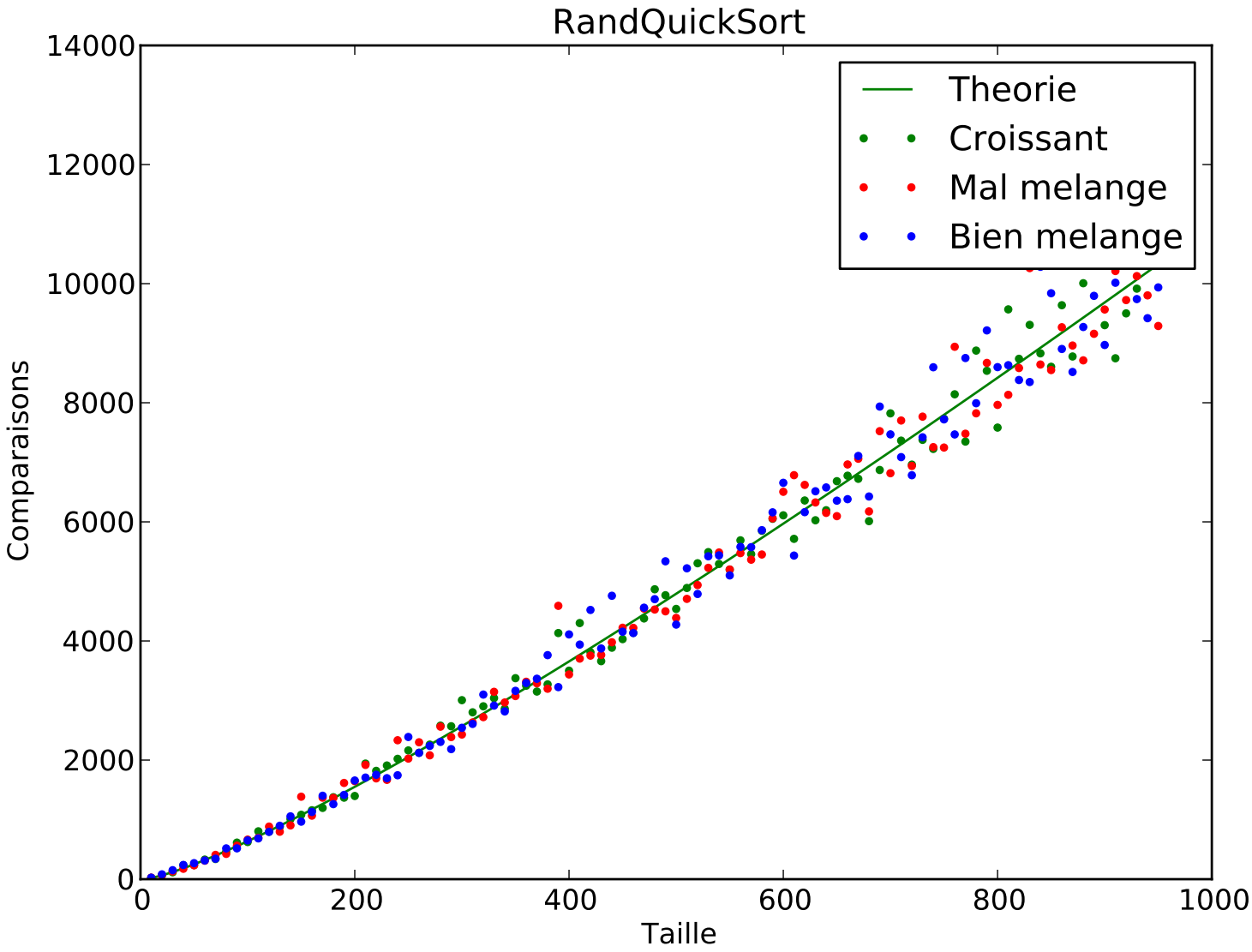
Et si. . .

- Idée toute bête : *et si au lieu de choisir un pivot déterministe, on le choisissait au hasard dans le tableau ?*
- On obtient une **version randomisée** de l'algorithme : à chaque exécution, on peut avoir un comportement différent
- (Le résultat, lui, n'est pas aléatoire : on trie toujours dans l'ordre croissant)
- Quelques lignes de code à changer

RandQuickSort en Python

```
def RandQuickSortRec(L,a,b):
    if (a<b):
        k=Partitionne(L,a,b,random.randint(a,b))
        RandQuickSortRec(L,a,k-1)
        RandQuickSortRec(L,k+1,b)

def RandQuickSort(L):
    RandQuickSortRec(L,0,len(L)-1)
```


RandQuickSort
Theorie
Croissant
Mal melange
Bien melange
Comparaisons
Taille
0
2000
4000
6000
8000
10000
12000
14000
0
200
400
600
800
1000

Ré-intreprétons. . .

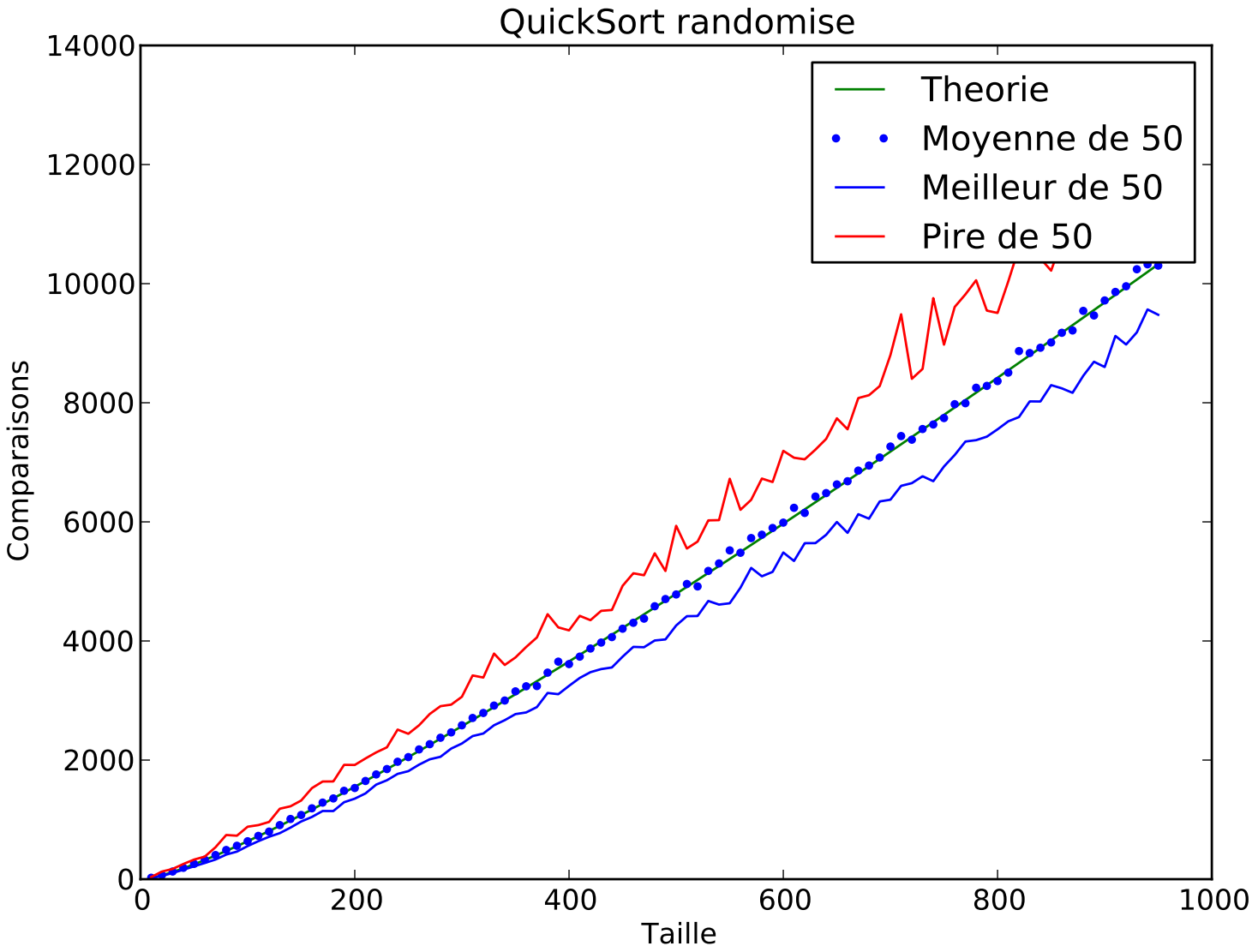
- **RandQuickSort** n'a pas l'air très sensible (statistiquement) au caractère plus ou moins bien mélangé de la liste de départ

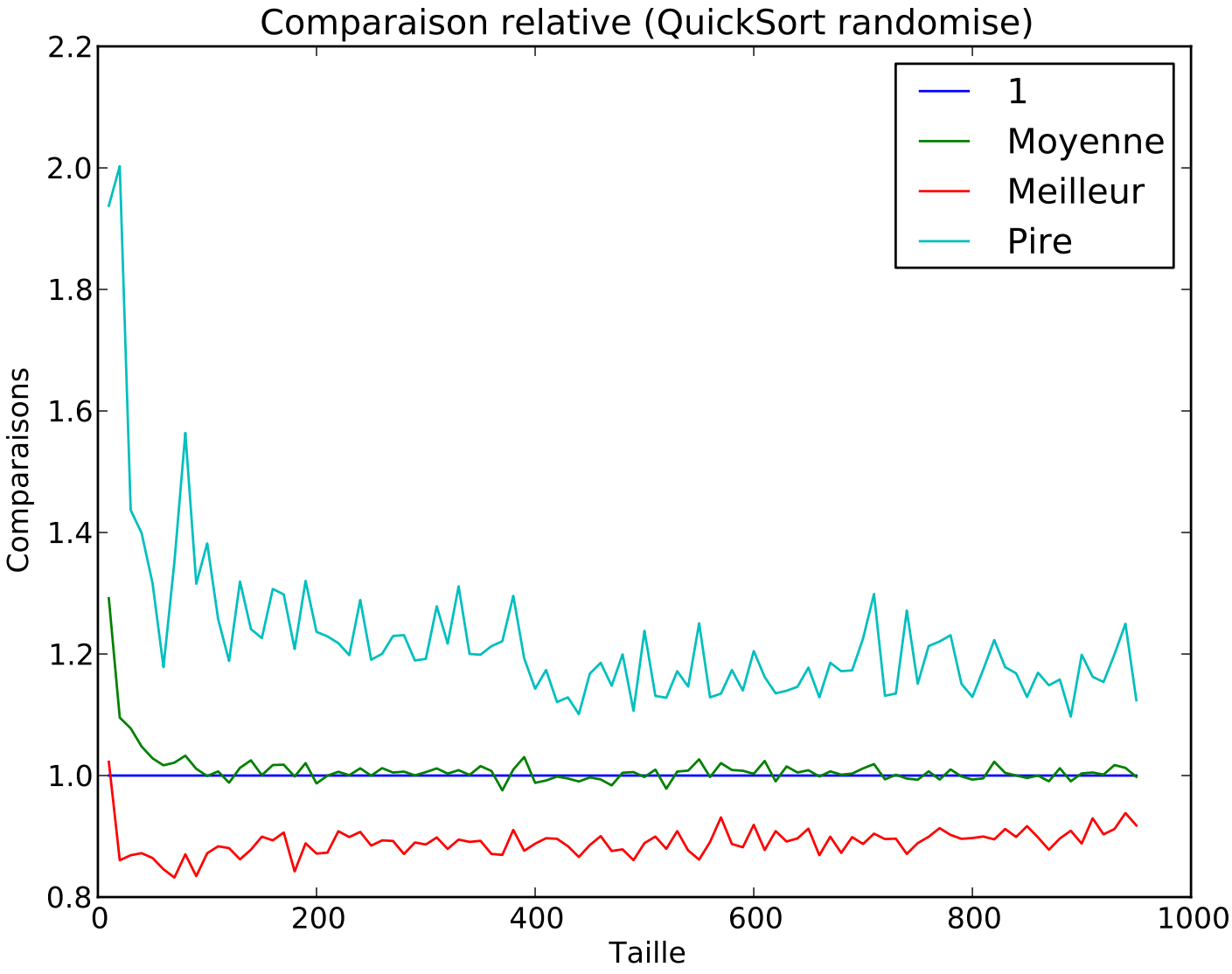
Ré-intreprétons. . .

- **RandQuickSort** n'a pas l'air très sensible (statistiquement) au caractère plus ou moins bien mélangé de la liste de départ
- Il a l'air de *toujours* se comporter comme **QuickSort** sur des listes aléatoires

Ré-intreprétons. . .

- **RandQuickSort** n'a pas l'air très sensible (statistiquement) au caractère plus ou moins bien mélangé de la liste de départ
- Il a l'air de *toujours* se comporter comme **QuickSort** sur des listes aléatoires
- Les variations par rapport à la moyenne semblent assez faibles


QuickSort randomise
Theorie
Moyenne de 50
Meilleur de 50
Pire de 50
Comparaisons
Taille
0
2000
4000
6000
8000
10000
12000
14000
0
200
400
600
800
1000


Comparaison relative (QuickSort randomise)
1
Moyenne
Meilleur
Pire
Comparaisons
Taille
2.2
2.0
1.8
1.6
1.4
1.2
1.0
0.8
0
200
400
600
800
1000

Et maintenant ?

- On va *prouver* ce qu'on vient d'observer
- Mais on a besoin de préciser ce qu'on suppose vrai des tirages aléatoires

Générateur aléatoire

La fonction `randint(a,b)` de `Python` (module `random`) est censée

- nous renvoyer un nombre entier aléatoire compris entre a et b (inclus), **uniforme** (chaque entier de $[[a, b]]$ a probabilité $1/(b - a + 1)$)

Générateur aléatoire

La fonction `randint(a,b)` de `Python` (module `random`) est censée

- nous renvoyer un nombre entier aléatoire compris entre a et b (inclus), **uniforme** (chaque entier de $[[a, b]]$ a probabilité $1/(b - a + 1)$)
- faire en sorte que le résultat de **chaque appel** soit **indépendant de tous ceux qui l'ont précédé**

Générateur aléatoire

La fonction `randint(a,b)` de `Python` (module `random`) est censée

- nous renvoyer un nombre entier aléatoire compris entre a et b (inclus), **uniforme** (chaque entier de $[[a, b]]$ a probabilité $1/(b - a + 1)$)
- faire en sorte que le résultat de **chaque appel** soit **indépendant de tous ceux qui l'ont précédé**

Variantes :

Générateur aléatoire

La fonction `randint(a,b)` de `Python` (module `random`) est censée

- nous renvoyer un nombre entier aléatoire compris entre a et b (inclus), **uniforme** (chaque entier de $[[a, b]]$ a probabilité $1/(b - a + 1)$)
- faire en sorte que le résultat de **chaque appel** soit **indépendant de tous ceux qui l'ont précédé**

Variantes :

- `flip()` : équivalent à `randint(0,1)` ("pile ou face" avec une pièce équilibrée)

Générateur aléatoire

La fonction `randint(a,b)` de `Python` (module `random`) est censée

- nous renvoyer un nombre entier aléatoire compris entre a et b (inclus), **uniforme** (chaque entier de $[[a, b]]$ a probabilité $1/(b - a + 1)$)
- faire en sorte que le résultat de **chaque appel** soit **indépendant de tous ceux qui l'ont précédé**

Variantes :

- `flip()` : équivalent à `randint(0,1)` ("pile ou face" avec une pièce équilibrée)
- `random()` : renvoie un nombre réel uniforme sur $[0, 1]$

Générateur aléatoire

La fonction `randint(a,b)` de `Python` (module `random`) est censée

- nous renvoyer un nombre entier aléatoire compris entre a et b (inclus), **uniforme** (chaque entier de $[[a, b]]$ a probabilité $1/(b - a + 1)$)
- faire en sorte que le résultat de **chaque appel** soit **indépendant de tous ceux qui l'ont précédé**

Variantes :

- `flip()` : équivalent à `randint(0,1)` ("pile ou face" avec une pièce équilibrée)
- `random()` : renvoie un nombre réel uniforme sur $[0, 1]$
- `Bernoulli(p)` : renvoie 1 avec probabilité p, 0 avec probabilité $1 - p$ (pour $0 < p < 1$)

Générateur aléatoire

La fonction `randint(a,b)` de `Python` (module `random`) est censée

- nous renvoyer un nombre entier aléatoire compris entre a et b (inclus), **uniforme** (chaque entier de $[[a, b]]$ a probabilité $1/(b - a + 1)$)
- faire en sorte que le résultat de **chaque appel** soit **indépendant de tous ceux qui l'ont précédé**

Variantes :

- `flip()` : équivalent à `randint(0,1)` ("pile ou face" avec une pièce équilibrée)
- `random()` : renvoie un nombre réel uniforme sur $[0, 1]$
- `Bernoulli(p)` : renvoie 1 avec probabilité p, 0 avec probabilité $1 - p$ (pour $0 < p < 1$)
- **toujours** en supposant que les appels sont **indépendants**

Analyse de **[Rand]QuickSort**

Théorème

Soit $n \geq 1$, $\sigma \in S_n$ une permutation quelconque de $[[1, n]]$, et soit

- QS_n, la variable aléatoire qui décrit le nombre de comparaisons de clés dans une exécution de l'algorithme déterministe **QuickSort** sur un *tableau de n valeurs distinctes, rangées initialement dans un ordre aléatoire uniforme* ;
- $RQS_n(\sigma)$, la variable aléatoire qui décrit le nombre de comparaisons de clés dans une exécution de l'algorithme randomisé **RandQuickSort** sur un *tableau de n valeurs distinctes, rangées initialement selon l'ordre* σ.

Alors QS_n et $RQS_n(\sigma)$ ont la **même loi** : pour tout k,

$$\mathbb{P}(QS_n = k) = \mathbb{P}(RQS_n(\sigma) = k).$$

Analyse de **[Rand]QuickSort**

Théorème

Soit $n \geq 1$, $\sigma \in S_n$ une permutation quelconque de $[[1, n]]$, et soit

- QS_n, la variable aléatoire qui décrit le nombre de comparaisons de clés dans une exécution de l'algorithme déterministe **QuickSort** sur un *tableau de n valeurs distinctes, rangées initialement dans un ordre aléatoire uniforme* ;
- $RQS_n(\sigma)$, la variable aléatoire qui décrit le nombre de comparaisons de clés dans une exécution de l'algorithme randomisé **RandQuickSort** sur un *tableau de n valeurs distinctes, rangées initialement selon l'ordre* σ.

Alors QS_n et $RQS_n(\sigma)$ ont la **même loi** : pour tout k,

$$\mathbb{P}(QS_n = k) = \mathbb{P}(RQS_n(\sigma) = k).$$

Corollaire 1 : même espérance, même variance. . .

Analyse de **[Rand]QuickSort**

Théorème

Soit $n \geq 1$, $\sigma \in S_n$ une permutation quelconque de $[[1, n]]$, et soit

- QS_n, la variable aléatoire qui décrit le nombre de comparaisons de clés dans une exécution de l'algorithme déterministe **QuickSort** sur un *tableau de n valeurs distinctes, rangées initialement dans un ordre aléatoire uniforme* ;
- $RQS_n(\sigma)$, la variable aléatoire qui décrit le nombre de comparaisons de clés dans une exécution de l'algorithme randomisé **RandQuickSort** sur un *tableau de n valeurs distinctes, rangées initialement selon l'ordre* σ.

Alors QS_n et $RQS_n(\sigma)$ ont la **même loi** : pour tout k,

$$\mathbb{P}(QS_n = k) = \mathbb{P}(RQS_n(\sigma) = k).$$

Corollaire 1 : même espérance, même variance. . .
Corollaire 2 : pour **RandQuickSort**, il n'y a pas de tableau plus mauvais qu'un autre