DIU Enseigner l'informatique au lycée

Notes de cours

Jean-Christophe Filliâtre

Jean-Christophe.Filliatre@lri.fr

Apprentissage

définition

Un algorithme traditionnel donne une réponse correcte à un problème précisément défini (par ex. « trier des entiers »).

Par opposition, un algorithme d'apprentissage apporte une réponse plausible, mais pas nécessairement exacte, à un problème auquel il est difficile d'appliquer un algorithme traditionnel.

C'est un domaine en forte expansion depuis les années 2010.

Ceci s'explique par la disponibilité de données massives.

applications

- Il serait trop coûteux de donner une réponse exacte.
  - Exemple : jouer au Go.
- Il n'y a pas de définition précise du problème.
  - Exemple : traduire une langue étrangère.
- Les données sont incomplètes/imprécises.
  - Exemple : la publicité ciblée.

principe

L'apprentissage procède (en général) en deux temps :

1. La phase d'apprentissage proprement dit, par exemple sur la base de données connues.

2. Le calcul de la réponse pour une nouvelle donnée.

classification

On s'intéresse ici au problème particulier de la classification :

- chaque donnée appartient à une classe ;
- pour une nouvelle donnée, on cherche à deviner sa classe.

On suppose par ailleurs que l'on connaît la classe des données utilisées dans la phase d'apprentissage.

On parle d'apprentissage supervisé.

classification

Plus précisément, on se donne

- un ensemble E d'éléments,
- un ensemble C de classes,
- un ensemble $T = \{(e_1, c_1), (e_2, c_2), \ldots, (e_n, c_n)\}$ d'éléments e_i dont la classe c_i est donnée.

On cherche à construire une fonction $c : E \to C$ qui estime la classe d'un élément quelconque.

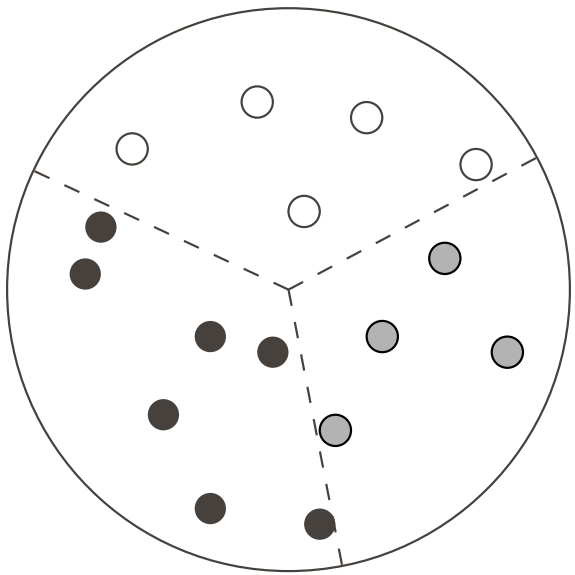
exemples

- Iris de Fischer
  - $C =$ trois espèces d'Iris (setosa, virginica, versicolor)
  - $E =$ ensemble de quatre caractéristiques d'une fleur (longueur/largeur sépale, longueur/largeur pétale)
  - $T =$ 150 échantillons

- Articles de presse
  - $C =$ catégories : culture, politique, sport, etc.
  - $E =$ ensemble d'articles (des textes)
  - $T =$ un sous-ensemble d'articles déjà catégorisés

autre exemple

- $C = \{\text{blanc}, \text{gris}, \text{noir}\}$
- $E =$ les points situés dans un cercle
- $T =$ 16 points dont on connaît la couleur



un algorithme

L'algorithme des k plus proches voisins est un algorithme d'apprentissage supervisé.

Au-delà de C, E et T, on suppose donnés

- une notion de distance sur E,
- un paramètre $k \in \mathbb{N}^\star$.

Principe de l'algorithme :

1. on reçoit en entrée un élément e
2. on sélectionne dans T les k éléments ayant les distances à e les plus faibles
3. on renvoie la classe majoritaire parmi les classes de ces k éléments

cas d'égalité

Il peut y avoir des cas d'égalité

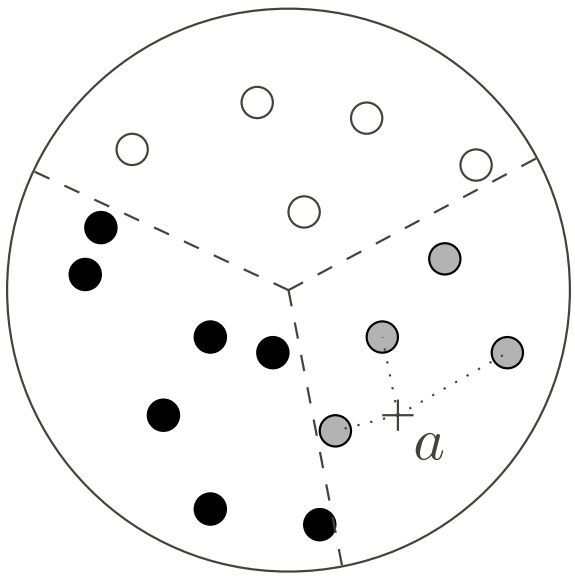
- dans les distances calculées,
- dans les classes les plus représentées.

On peut alors choisir de trancher arbitrairement.

illustration

On prend

- la distance euclidienne
- le paramètre $k = 3$



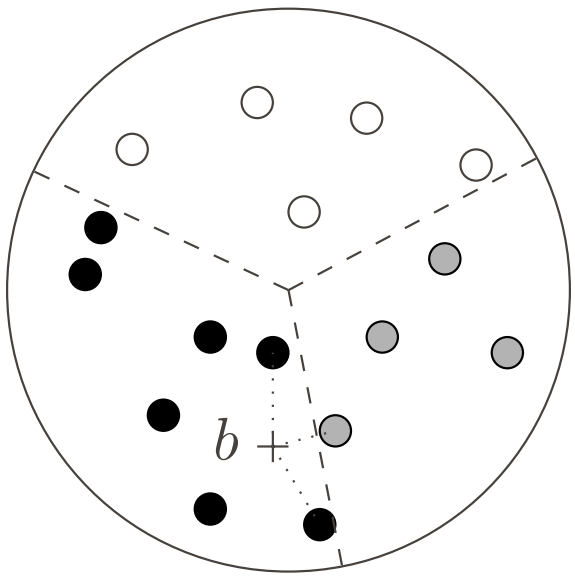
3 voisins : ●●●

résultat : ●

illustration

On prend

- la distance euclidienne
- le paramètre $k = 3$



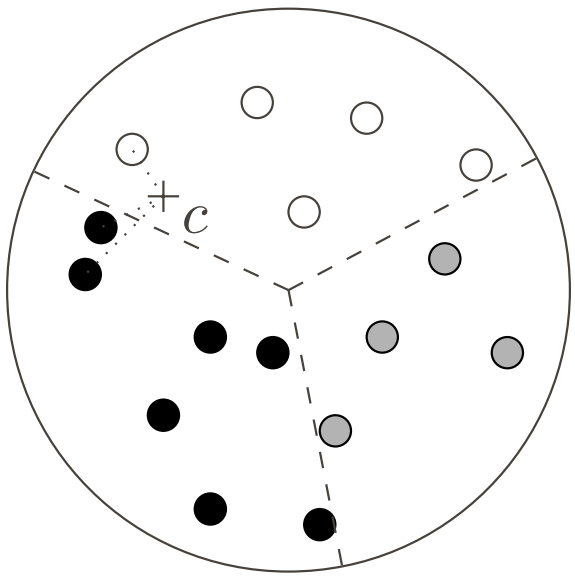
3 voisins : ●●●

résultat : ●

illustration

On prend

- la distance euclidienne
- le paramètre $k = 3$



3 voisins : ○●●

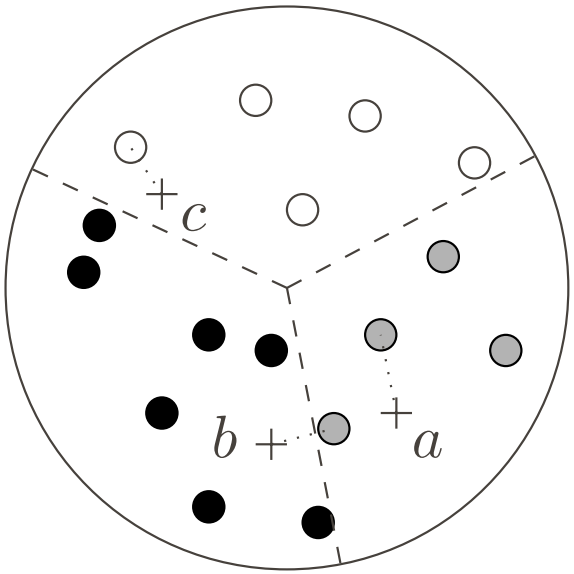
résultat : ●

remarque

Si l'élément reçu en entrée est l'un des éléments e_i de T, l'algorithme des k plus proches voisins ne donnera pas forcément la réponse c_i !

On peut choisir que, dans ce cas, la réponse doit être c_i et non celle calculée par l'algorithme.

influence du paramètre k

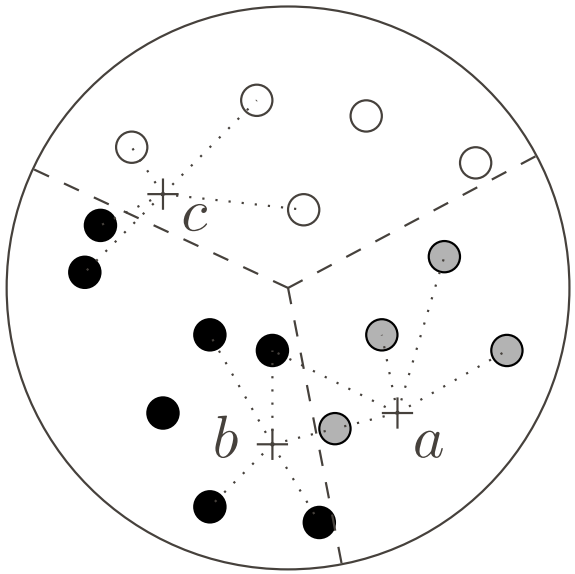


$k = 1$

a :

b :

c :



$k = 5$

a :

b :

c :

influence du paramètre k

- Un paramètre k trop petit peut donner de mauvais résultats.
- Mais un paramètre k trop grand également !

Comment choisir le paramètre k ?

influence du paramètre k

Il n'y a pas de moyen systématique de choisir la valeur de k.

La meilleure valeur possible de k dépend du problème considéré et du jeu de données.

On peut déterminer k empiriquement avec une validation croisée.

validation croisée

On utilise le tableau T lui-même comme base d'évaluation :

1. on retire de T un certain nombre d'éléments,
2. on utilise l'algorithme sur les éléments qui ont été retirés,
3. on compare les réponses de l'algorithme avec les réponses attendues.

On recommence plusieurs fois, en retirant des éléments différents à chaque fois et avec plusieurs valeurs de k.

(C'est une méthode d'apprentissage.)

programmation

Programmons ensemble l'algorithme des k plus proches voisins dans le cas des Iris de Fischer.

On dispose d'un fichier CSV :

```
sepal_length,sepal_width,petal_length,petal_width,species
5.1,3.5,1.4,0.2,setosa
7.0,3.2,4.7,1.4,versicolor
6.7,3.1,5.6,2.4,virginica
...
```

On cherche à écrire un programme Python de ce type :

```
Entrer la longueur de pétale : 2.5
Entrer la largeur  de pétale : 0.75
k=3 donne setosa
```

notion de distance

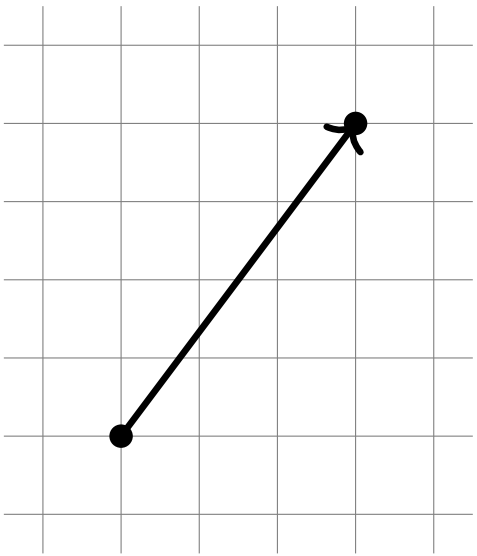
Si nos éléments ont n caractéristiques réelles (des flottants en Python), ce sont des points de $E = \mathbb{R}^n$.

- Exemple : pour les Iris de Fischer, on a $n = 4$ dans le fichier et $n = 2$ dans le code qu'on a écrit.

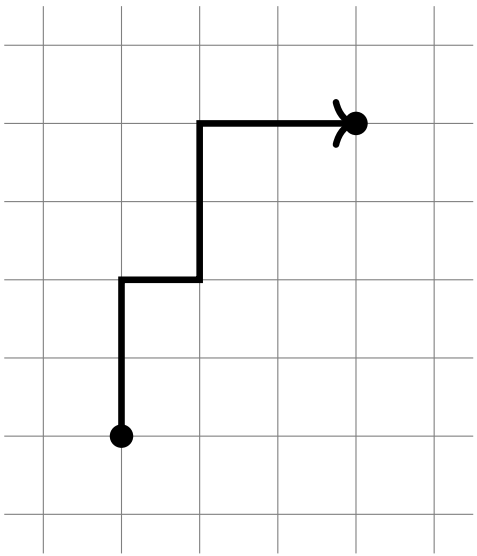
Comme distance entre deux points $(x_1, \ldots, x_n)$ et $(y_1, \ldots, y_n)$, on a de nombreux choix :

- distance euclidienne : $d = \sqrt{(\sum_{i=1}^{n}(x_i - y_i)^2)}$
- distance de Manhattan : $d = \sum_{i=1}^{n} |x_i - y_i|$
- distance de Tchebychev : $d = \max_{1 \leq i \leq n} |x_i - y_i|$
- etc.

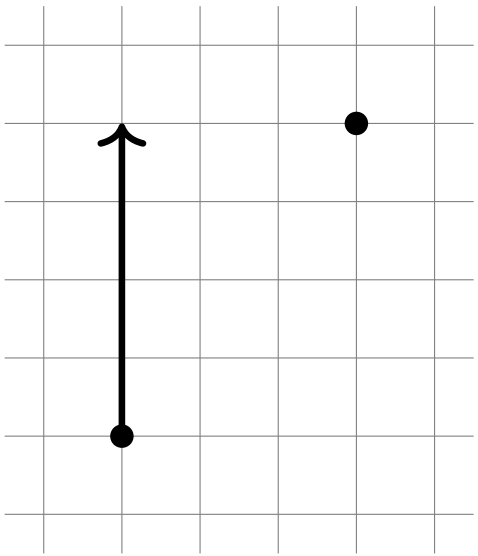
notion de distance



Euclidienne : 5



Manhattan : 7



Tchebychev : 4

distance entre chaînes de caractères

L'apprentissage sur des textes (classification, traduction, etc.) amène à définir une distance sur les chaînes de caractères.

On cherche à capturer une notion de proximité.

Deux grands classiques :

- la distance de Hamming ;
- la distance d'édition.

la distance de Hamming

La distance de Hamming compte le nombre de caractères à une même position qui diffèrent.

Chaînes	Dist.
`Coliot-Jurie` `Joliot-Curie`	2
`art` `arbre`	3

(C'est un petit exercice de programmation très facile.)

la distance de Hamming

En pratique, cependant, la distance de Hamming n'est pas la notion de proximité la plus naturelle.

Chaînes	Dist.
`canard` `cannard`	4
`amer` `ramer`	5

Une simple faute de frappe donne tout de suite une grande distance.

une meilleure distance

La distance d'édition entre deux chaînes de caractères est le nombre minimal d'ajouts, de suppressions et de modifications permettant de transformer l'une en l'autre.

Chaînes	Dist.
`canard` `cafards`	2
`amer` `ramer`	1

(Calculer la distance d'édition est un bon exercice de programmation dynamique, qui n'est pas sans rapport avec le problème de l'alignement de séquence.)

Exercices

Algorithmes probabilistes

définition

Un algorithme probabiliste, ou algorithme randomisé, est un algorithme qui utilise une source de hasard.

exemple

Pour mélanger les éléments d'un tableau, on peut utiliser le mélange de Knuth :

```
def melange(t):
    for i in range(1, len(t)):
        j = randint(0, i)
        t[i], t[j] = t[j], t[i]
```

le hasard dans un ordinateur

Une fonction comme `randint` implémente un générateur de nombres pseudo-aléatoires.

Ce n'est qu'un algorithme qui construit une séquence de nombres de façon déterministe à partir d'une donnée initiale (qu'on appelle la graine).

Cette séquence n'a que l'apparence du hasard, avec de plus ou moins bonnes propriétés.

exemples de générateurs

- 1958 : congruence linéaire (historique, mais pas terrible)

$$X_{n+1} = (aX_n + c) \mod m$$

- 1965 : registre à décalage à rétroaction linéaire (bien meilleur)
- ...
- 1998 : Mersenne Twister (c'est celui de Python, notamment)
  Il a une période $2^{19937} - 1$.
- ...

algorithmes probabilistes

Il y a différents types d'algorithmes probabilistes :

- Le résultat est toujours correct. Le temps de calcul est déterministe. C'est la distribution des résultats possibles qui nous intéresse.
- Monte-Carlo : Le résultat n'est pas toujours correct. Le temps de calcul est déterministe. C'est la probabilité de la correction qui nous intéresse.
- Las Vegas : Le résultat est toujours correct. Le hasard influence le temps de calcul (il est petit avec une forte probabilité).

retour sur le mélange de Knuth

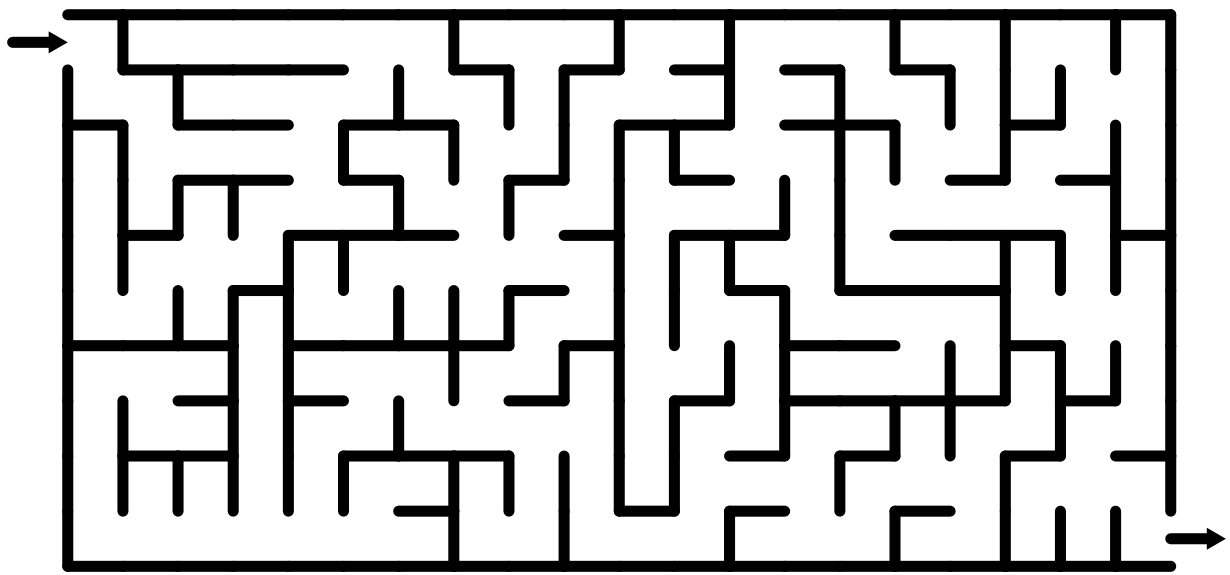
Le mélange de Knuth

```
def melange(t):
    for i in range(1, len(t)):
        j = randint(0, i)
        t[i], t[j] = t[j], t[i]
```

est un bon mélange : chaque permutation est choisie avec la même probabilité, à savoir $1/n!$ pour un tableau de taille n.

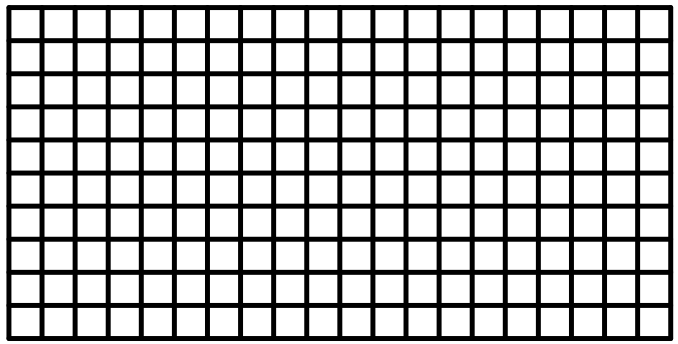
un autre exemple

On veut construire un labyrinthe parfait, i.e., il existe un chemin et un seul entre deux cases.

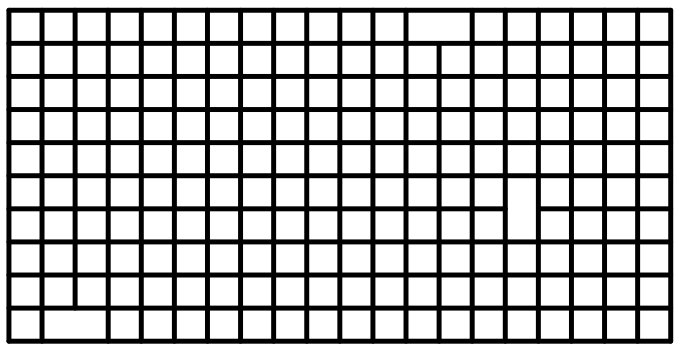


labyrinthe parfait : un algorithme

1. On part d'une grille où toutes les cases sont isolées.



2. Tant que toutes les cases ne sont pas reliées entre elles, on choisit aléatoirement une paire de cases adjacentes et on supprime le mur entre les deux si elles ne sont pas déjà reliées par un chemin.



labyrinthe parfait : un algorithme

On peut montrer que cet algorithme termine avec probabilité 1.
Mais cela va prendre beaucoup de temps en pratique.

Fort heureusement, on peut le réécrire ainsi :

1. Construire un tableau de toutes les paires de cases adjacentes.
2. Le mélanger.
3. Le parcourir et procéder comme précédemment pour chaque paire.

Maintenant le temps de calcul est déterministe.

labyrinthe parfait : un algorithme

Avec cet algorithme, tous les labyrinthes parfaits sont-ils équiprobables ?

- Non
  (mais cela reste un très bon algorithme).

Est-il possible de concevoir un autre algorithme donnant tous les labyrinthes parfaits avec la même probabilité ?

- Oui.

méthodes de Monte-Carlo

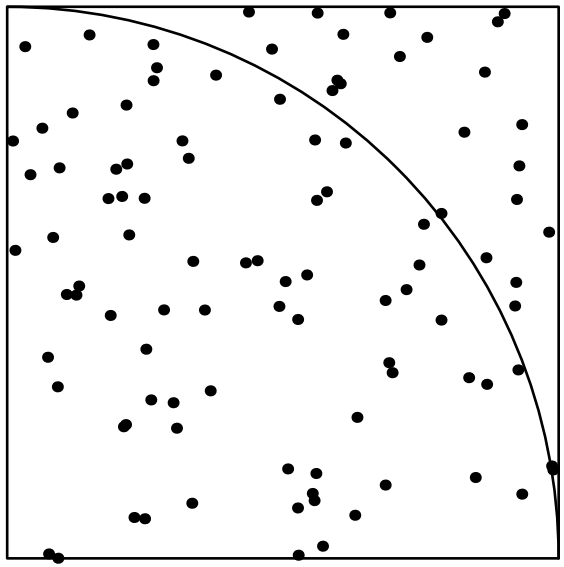
Rappel :

- Le résultat n'est pas toujours correct.
- Le temps de calcul est déterministe.
- C'est la probabilité de la correction qui nous intéresse.

un grand classique

Approximation de π par la méthode de Monte-Carlo.

- On tire des points au hasard dans un carré.
- On compte combien tombent dans le quart de cercle.
- La proportion doit être $\frac{\pi}{4}$.



approximation de π par la méthode de Monte-Carlo

```
def approx(n):
    dedans = 0
    for _ in range(n):
        x, y = random(), random()
        if x**2 + y**2 <= 1:
            dedans += 1
    return 4 * dedans / n
```

Avec 10^7 points, on obtient $3,141208$. Ce n'est pas terrible...

Néanmoins, il est possible d'affirmer que le résultat est dans

$$[\pi - \frac{1}{\sqrt{n}}, \pi + \frac{1}{\sqrt{n}}]$$

avec une probabilité de 95%.

un meilleur exemple

Pour tester si on nombre est premier, on peut utiliser un algorithme probabiliste, avec l'idée suivante :

- s'il répond que le nombre est composé, il l'est ;
- s'il répond que le nombre est premier, il l'est probablement.

test de primalité de Fermat

Repose sur le petit théorème de Fermat : p est premier si et seulement si

$$\text{pour tout } 1 \leq a < p, \text{ on a } a^{p-1} \equiv 1 \mod p$$

D'où l'idée suivante pour tester si n est premier :

- On tire un entier $1 \leq a < n$ au hasard.
- Si $a^{n-1} \not\equiv 1 \mod n$, on répond non (correct).
- Si $a^{n-1} \not\equiv 1 \mod n$, on répond oui (peut-être faux).

code Python

```
def expo(x, k, m):
    """renvoie x^k mod m"""
    assert 0 <= x < m
    ....

def fermat(n):
    """test de primalité de Fermat"""
    a = randint(2, n-1)
    return expo(a, n-1, n) == 1
```

résultat

La probabilité que `fermat(n)` renvoie `True`,

- lorsque `n` est premier, vaut 1 ;
- lorsque `n` est composé, est $\leq \frac{1}{2}$.

En particulier, si on répète k fois l'expérience, on a maintenant une probabilité inférieure à

$$\frac{1}{2^k}$$

de s'être trompé.

Avec $k = 20$, on a une probabilité d'erreur inférieure à 10^{-6}.

méthodes de Las Vegas

Rappel :

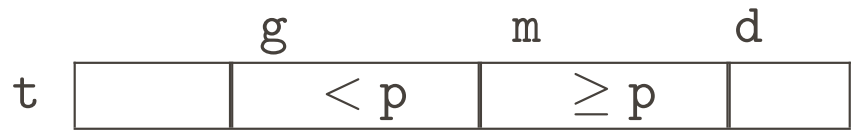
- Le résultat est toujours correct.
- Le hasard influence le temps de calcul : il est petit avec une forte probabilité.

un exemple fondamental : le tri rapide

Le tri rapide (*quicksort*) fonctionne ainsi :

- Il applique la méthode diviser pour régner.
- Pour trier le segment `t[g..d[`,

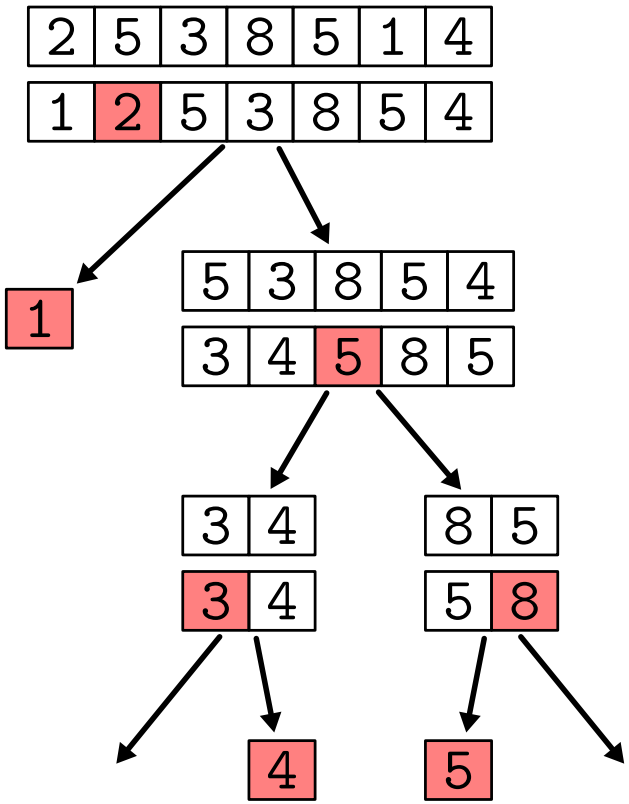
  1. on réarrange les éléments pour avoir



     pour une certaine valeur `p` appelée pivot

  2. on trie récursivement `t[g..m[` et `t[m..d[`

exemple



pire cas

Le pire cas correspond à un pivot systématiquement au bord.

On a lors une complexité quadratique.

Idée : prendre comme pivot un élément au hasard dans `t[g..d[`.

Encore plus simple : mélanger avant de trier !

code

```
def partition(t, g, d):
    ...
```

```
def quickrec(t, g, d):
    """trie t[d..d[ avec le tri rapide"""
    ...
```

```
def quicksort(t):
    melange(t)
    quickrec(t, 0, len(t))
```

résultat

On peut majorer la probabilité que le tri rapide... ne soit pas rapide.

Exemples :

- Sur un tableau d'un million d'éléments, la probabilité que le temps de calcul soit dix fois supérieur au temps de calcul moyen (qui est en $N \log N$) est inférieure à $0,00001$.

- Sur un gros tableau, la probabilité que le tri rapide fasse autant de comparaisons que le tri par insertion est plus faible que la probabilité que la foudre frappe votre ordinateur pendant le tri !

Exercices