Cours

qkzk

Diviser pour régner

Classe d'algorithme où l'on * découpe un problème en sous problèmes *indépendants* qui s'énoncent de la même manière * qu'on recompose à la fin pour former une solution

C'est une approche "du haut vers les bas", généralement, les algorithmes sont récursifs

Nous allons exposer quelques problèmes simples afin d'explorer la démarche. Dans un premier temps, nos nouveaux algorithmes seront moins efficaces que leur version classique.

Ensuite nous traiterons de vrais problèmes avec l'approche diviser pour régner.

Recherche du maximum dans une liste

On dispose d'un tableau de nombres, on en cherche le plus grand élément.

```
tableau = [5, 71, 23, 45, 28, 89, 63, 39]
```

Algorithme itératif naturel

On a déjà vu un algorithme en première :

1. On initialise `max = tableau[0]`, on parcourt élément par élément,
2. Pour chaque élément `elt` du tableau,

   Si `elt > max` alors `max = elt`
3. On retourne `max`

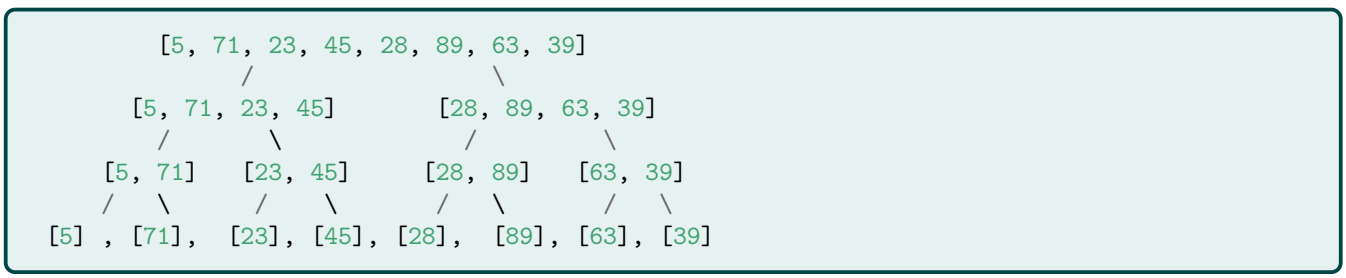
Version diviser pour régner

fonction `maximum`: `tableau ---> entier`

0. Le maximum d'un tableau de taille 1 est son unique élément.
1. On sépare le tableau en deux parties (sensiblement de même taille),
2. on retourne le plus grand des maxima des parties gauche et droite.

```
tableau = [5, 71, 23, 45, 28, 89, 63, 39]
```

1. séparer



2. Recombiner : on ne garde que le plus grand de chaque paire

```
[5, 71],   [23, 45],  [28, 89],  [63, 39]
  \          /          \        /
  [71]      [45]        [89]     [63]
      [71, 45]            [89, 63]
           \                 /
           [71]           [89]
                [71,  89]
                    |
                  [89]
```

Est-ce plus efficace ? Non... c'est même plus lent !

maximum

Recherche d'un élément dans une liste (pas forcément trié)

On dispose d'un tableau d'entiers. On cherche à savoir s'il contient un élément.

Version itérative (cf première)

fonction `chercher : (tableau, clé) ----> booléen`

1. on initialisé `trouvé = Faux`
2. on parcourt le tableau élément par élément:
   - Si `élément == clé`, alors `trouvé = Vrai`
3. on retourne `trouvé`

Version diviser pour régner

fonction `chercher : (tableau, clé) ----> booléen`

1. Pour un tableau de taille 1, il contient la clé si valeur est la clé
2. On sépare le tableau en deux parties sensiblement de même taille (gauche et droite)
3. Le tableau contient la clé si
   `chercher(gauche, clé)` ou `chercher(droite, clé)`
   est vrai.

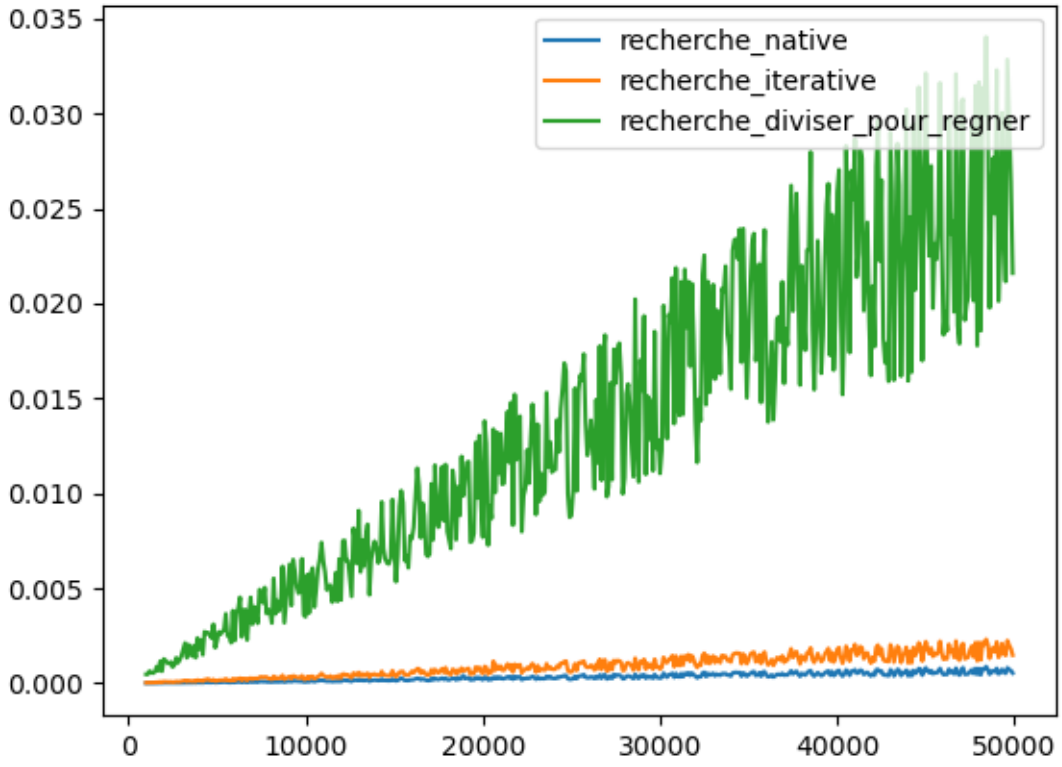
Exemple

```
tableau = [4, 10, 20, 5]
clé = 10
```

A-t-on `clé` dans `tableau` ?

```
clé = 10
                  [4, 10, 20, 5]
diviser      [4, 10]       [20, 5]
diviser     [4]   [10]     [20]   [5]
combiner   Faux ou Vrai | Faux ou Faux
combiner       Vrai    ou    Faux
combiner              Vrai
```

C'est mieux cette fois ??? Toujours pas.



Le coût est toujours linéaire, avec un coefficient assez mauvais.

Pourquoi est-ce inefficace dans ces cas ?

- Pour le maximum, on fait autant de comparaison que dans la méthode itérative.
- Pour la recherche on fait autant de comparaison ET on ajoute `n-1` "Vrai ou Faux".

Quand est-ce intéressant ?

- **Quand on a une structure particulière,**
- **Quand on peut éviter beaucoup d'étapes**
- **Quand on peut remplacer un calcul coûteux par un calcul moins coûteux,**

Dichotomie : c'est diviser pour régner

En première on a vu la recherche dichotomique, rappelons rapidement le principe

On cherche **dans un tableau trié** la présence d'un élément.

- On initialisé `trouvé = False`
- On regarde l'élément central du tableau,
- S'il est égal à la clé : `trouvé = Vrai`
- S'il est plus grand que la clé, on cherche entre le début et la *valeur centrale*,
- Sinon, on cherche entre *la valeur centrale* et la fin,

Dichotomie : récursif

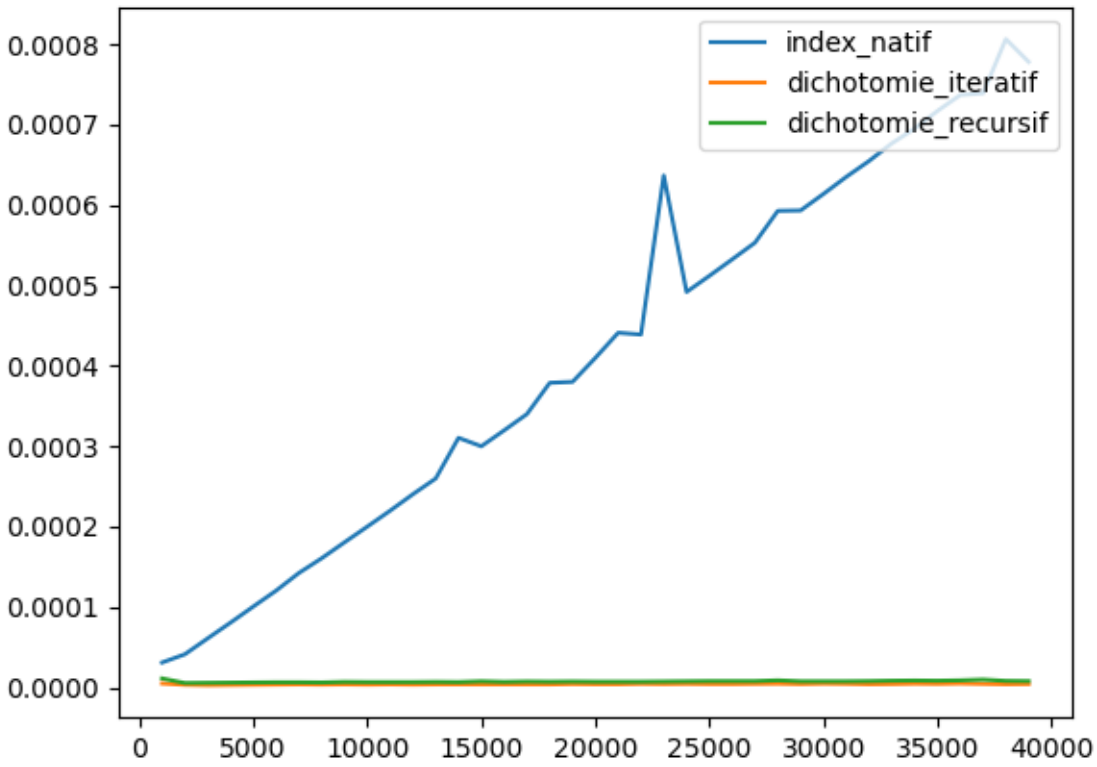
La version que nous avions étudiée était itérative.

On peut l'écrire en récursif.

En Python, ce n'est pas plus rapide :(

Python, n'est pas un langage *fonctionnel*, les récursions ne sont pas optimisées.

Mais



Calculer la puissance d'un nombre

Comment calculer 3^7 ?

$$3^7 = 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3$$

C'est déjà un algorithme !

Algorithme naïf pour y^n

```
Puissance(y, n)

1. initialiser p = 0 et i = 0
2. Tant que i < n, faire
    p = p * y
    i = i + 1
renvoyer p
```

Complexité ?

Clairement linéaire. Une seule boucle qui itère autant de fois que la puissance voulue.

Exponentiation rapide

Algorithme

ExpoRapide : $(y, n) \mapsto y^n$

```
Exporapide(y, n)

Si n = 0, renvoyer 1
Sinon, si n est pair,
    a = exporapide(y, n // 2)
    renvoyer a * a
Sinon,
    renvoyer y * exporapide(y, n - 1)
```

Un exemple

On veut calculer 3^7

Appels récursifs

```
exporapide(3, 7)
    7 > 0 et 7 impair
    on renvoie 3 * exporapide(3, 6)

exporapide(3, 6)
    6 > 0 et 6 pair
    a = exporapide(3, 3)
    on renvoie a * a

exporapide(3, 3)
    3 > 0 et 3 impair
    on renvoie 3 * exporapide(3, 2)

exporapide(3, 2)
    2 > 0 et 2 pair
    a = exporapide(3, 1)
    renvoyer a * a

exporapide(3, 1)
    1 > 0 et 1 impair
    renvoyer 3 * exporapide(3, 0)

exporapide(3, 0)
    0 est nul
    renvoyer 1
```

<—> Valeurs de retour

```
exporapide(3, 0) -> 1
exporapide(3, 1) -> 3 * 1   = 3
exporapide(3, 2) -> 3 * 3   = 9
exporapide(3, 3) -> 3 * 9   = 27
exporapide(3, 6) -> 27 * 27 = 729
exporapide(3, 7) -> 3 * 729 = 2187
```

On gagne assez peu d'étapes avec cet exemple !

Recommençons avec 3^9

Appels récursifs

```
exporapide(3, 9)
    9 > 0 et 8 impair
    on renvoie 3 * exporapide(3, 8)

exporapide(3, 8)
    8 > 0 et 8 pair
    a = exporapide(4, 4)
    on renvoie a * a

exporapide(3, 4)
    4 > 0 et 3 pair
    a = exporapide(3, 2)
    on renvoie a * a

exporapide(3, 2)
    2 > 0 et 2 pair
    a = exporapide(3, 1)
    renvoyer a * a

exporapide(3, 1)
    1 > 0 et 1 impair
    renvoyer 3 * exporapide(3, 0)

exporapide(3, 0)
    0 est nul
    renvoyer 1
```

<—> Valeurs de retour

```
exporapide(3, 0) -> 1
exporapide(3, 1) -> 3 * 1   = 3
exporapide(3, 2) -> 3 * 3   = 9
exporapide(3, 4) -> 9 * 9   = 81
exporapide(3, 8) -> 81 * 81 = 6561
exporapide(3, 7) -> 3 * 729 = 19683
```

Cette fois on passe de 9 étapes à seulement 5.

Pour les exposants légèrement plus grands qu'une puissance de deux, c'est un excellent algorithme.

Vitesses

Complément : rotation d'un quart de tour

rotation d'un quart de tour

Conclusion

La méthode ***diviser pour régner*** :

- découper le problème en sous-problèmes *indépendants* qui s'énoncent de la même manière
- résoudre les cas limites
- combiner les solutions

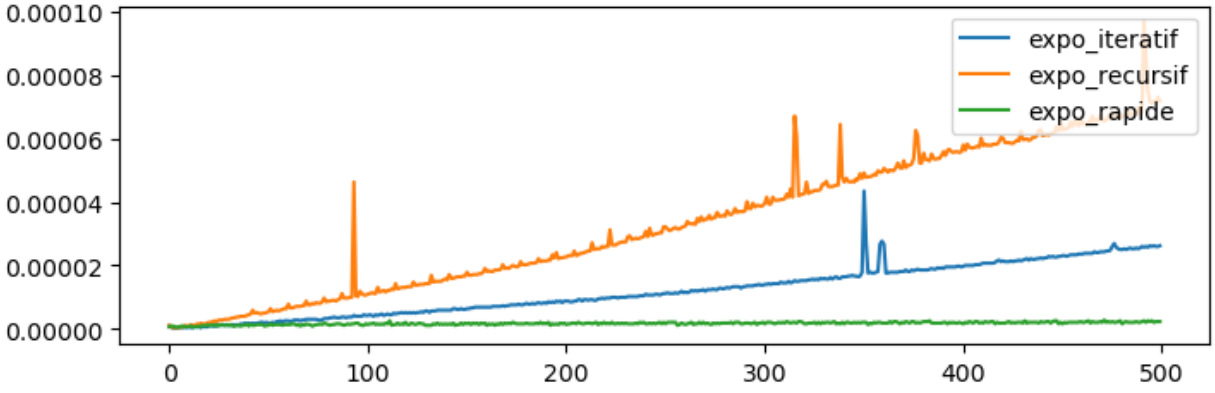


Figure 1: expo

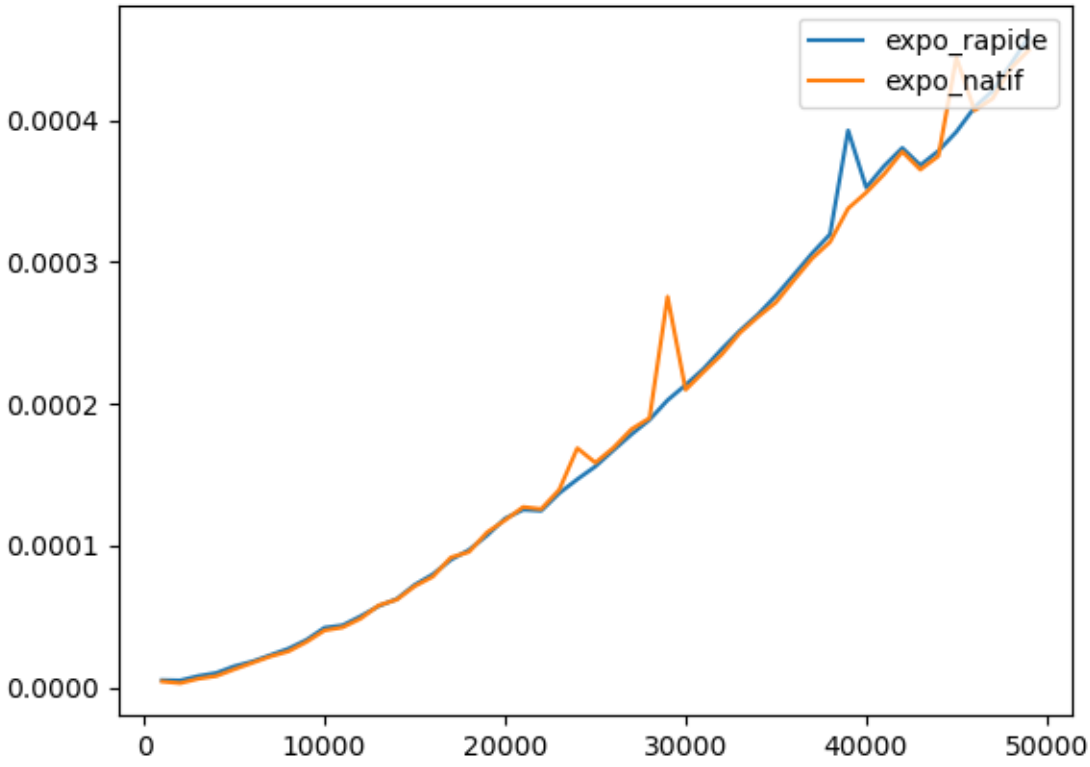


Figure 2: expo

Algorithmes

Les algorithmes présentés s'énoncent facilement de manière récursive.

Ce sont parfois les meilleurs possibles (expo rapide, recherche dichotomique, tri fusion)

Implémentation

Elle n'est pas toujours plus efficace. Cela dépend du langage employé.