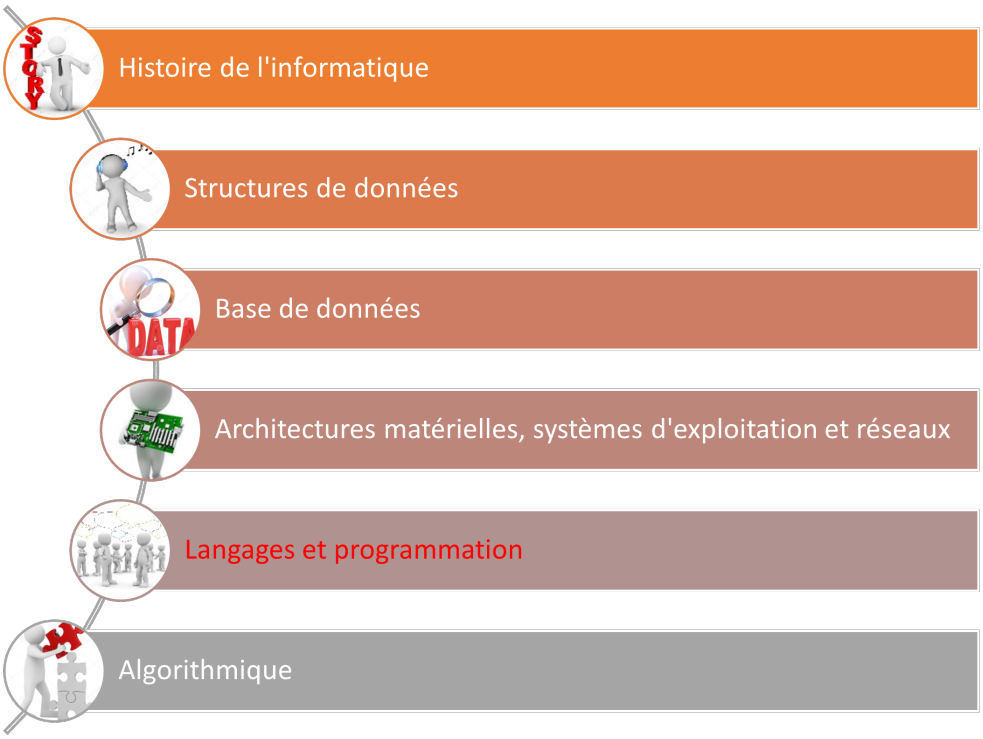


Table des matières

1 Le principe général diviser pour régner

1.1 Introduction

Définition :
Diviser pour Régner est un concept algorithmique pour résoudre plus rapidement des problèmes. Il se décompose en 3 grandes étapes :

- Diviser le problème initial en sous-problèmes (plus petits ou plus faciles a résoudre),
- Résoudre chaque sous-problème (récursivement ou directement),
- Combiner les solutions des sous-problèmes pour obtenir la solution du problème de départ.

1.2 Algorithme général diviser pour régner

```
FONCTION DiviserRegner(P : Probleme) :
  Si (P est un probleme simple) :
    Resoudre P et Retourner Solution
  Sinon
    Pour i = 1 à k :
      Solutioni = DiviserRegner(Pi )
  Solution = Combiner(solution1,..., solutionk)
  Retourner Solution
```

2 Un premier exemple : la recherche dichotomique

2.1 Rappel de l'algorithme vu en première

On rappelle qu'il s'agit de déterminer si un entier `val` apparait dans un tableau `tab`, ce dernier étant trié dans l'ordre croissant. L'algorithme renvoie l'indice d'une cellule (n'importe laquelle) de `tab` dans laquelle se trouve la valeur `val` ou `None` si la valeur n'apparait pas dans le tableau.

L'idée principale est de limiter la portion de recherche dans le tableau en manipulant les indices qui la bornent `g` et `d`. On découpe alors l'intervalle en deux (ce qui correspond bien à l'idée de "diviser pour régner") en testant la valeur au centre de l'intervalle `g ... d`.

L'algorithme vu en première est rappelé ci-dessous :

```
def recherche_dicho(val, tab):
    g = 0
    d = len(tab)
    while g < d - 1:
        m = (g + d)//2
        if val == tab[m]:
            return m
        elif val > tab[m]:
            g = m
        else:
            d = m
    return None
```

Justifions que le programme se termine toujours : le variant de boucle $d - c$ diminue à chaque puisque l'on coupe en deux à chaque étape la portion de recherche, or il s'agit d'un entier naturel, donc la boucle se termine.

Pour un tableau trié contenant un milliard de valeurs quel serait le nombre de récursions nécessaires ?
$2^{29} < 10^9 < 2^{30}$.
Ainsi après 30 itérations l'algorithme aura terminé son travail.

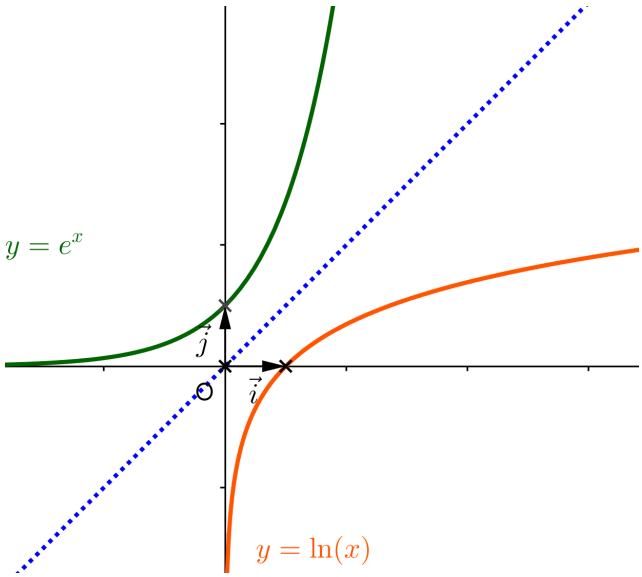
En première, nous avons utilisé une boucle `while` pour conclure, maintenant nous allons l'écrire sous une forme récursive : voir exercice n° 1

2.2 Les fonctions logarithmes

Définition :
Les fonctions logarithmes sont des fonctions définies sur $]0\ ;\ +\infty[$ qui vérifient :

$$\forall a > 0,\ \forall b > 0\ :\qquad f(ab) = f(a) + f(b)$$

Définition :
Parmi ces fonctions on étudie particulièrement en mathématiques la fonction logarithme népérien, notée ln, qui est la fonction réciproque de la fonction exponentielle.



Définition :
En sciences physiques et en chimie on utilise le logarithme décimal, noté log, défini par :

$$\forall x > 0 \text{ par : } \quad log(x) = \frac{ln(x)}{ln(10)}$$

Propriété :
La fonction log vérifie :

$$\forall k \in \mathbb{Z} \quad log\left(10^k\right) = k$$

Définition :
Nous utiliserons nous ici le logarithme de base 2, noté log_2, défini par :

$$\forall x > 0 \text{ par : } \quad log_2(x) = \frac{ln(x)}{ln(2)}$$

Propriété :
La fonction log_2 vérifie :

$$\forall k \in \mathbb{Z} \quad log\left(2^k\right) = k$$

Ainsi :
$log_2(8) = log_2(2^3) = 3$
$log_2(16) = log_2(2^4) = 4$
$log_2(1\,024) = log_2(2^{10}) = 10$

2.3 Complexité de la recherche dichotomique

On considère le script suivant qui va nous permettre de tester la complexité de l'algorithme de recherche dichotomique dans un tableau supposé trié :

Complexite_algorithme_recherche_dichotomique.py

Ainsi si notre tableau trié compte n valeurs, on découpe successivement à chaque étape la portion de recherche en deux. Le nombre d'étapes est donc le plus petit entier k tel que $2^k > n$

Ainsi : $k \approx log_2(n)$

Propriété :
La complexité de l'algorithme de recherche dichotomique dans un tableau supposé trié est donc en : $log_2(n)$.

3 Un deuxième exemple : le tri fusion

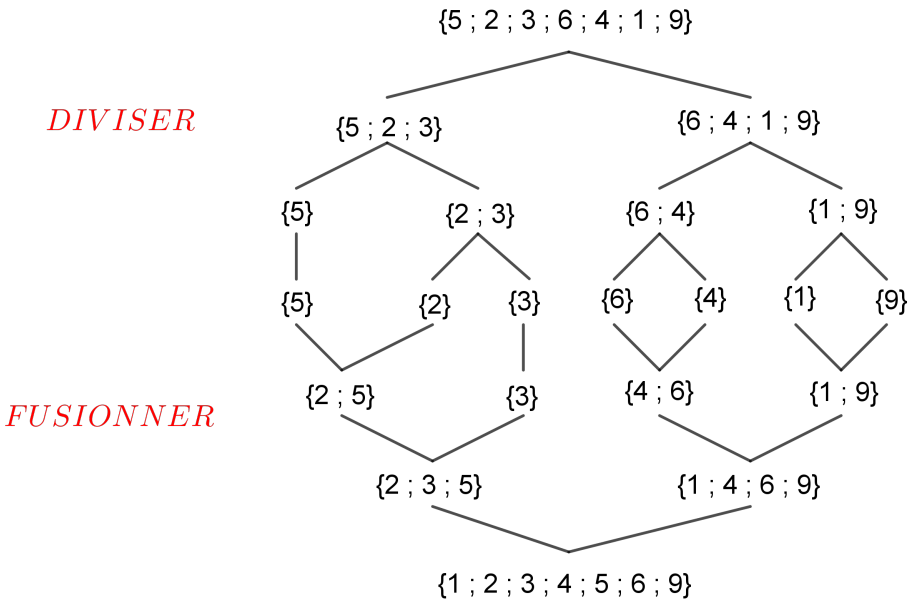
3.1 Principe général de l'algorithme de tri fusion

Soit n un nombre entier naturel non nul, on note $\mathcal{S}$ un ensemble de n nombres : $\mathcal{S} = \{a_1; a_2; \ldots; a_n\}$. L'objectif : trier les éléments de $\mathcal{S}$ dans l'ordre croissant.

Il existe une multitude d'algorithmes de tri, l'algorithme naïf a une complexité en n^2 : on parcourt le tableau dans son intégralité et l'on compare chaque valeur à toutes les autres valeurs qui suivent (on code une boucle `for i in range(n)` et une deuxième boucle imbriquée `for j in range(i, n)`).

Appliquons le principe Diviser pour Régner sur l'exemple suivant : $n = 7$ et $\mathcal{S} = \{5\ ;\ 2\ ;\ 3\ ;\ 6\ ;\ 4\ ;\ 1\ ;\ 9\}$

- **Étape 1** : Diviser le tableau en sous tableaux de taille 1 (et donc ordonnés !)
- **Étape 2** : Fusionner récursivement, les listes 2 par 2.



La fonction récursive correspondant au tri par fusion est :
`TriFusion(`a_1 ; a_2 ; ... ; a_n`) = Fusion(TriFusion (`a_1 ; a_2 ; ... ; $a_{n//2}$`) ; TriFusion(`$a_{n//2+1}$; ... ; a_n`) )`

3.2 Complexité du tri fusion

On divise récursivement les problèmes en deux sous-problèmes de taille identique
Comptons le nombre de sous- problèmes de taille identique générés,

- 2 sous-problèmes de taille $\frac{n}{2}$: $2 \times \frac{n}{2} = n$ opérations
- 4 sous-problèmes de taille $\frac{n}{4}$: $4 \times \frac{n}{4} = n$ opérations
- 8 sous-problèmes de taille $\frac{n}{8}$: $8 \times \frac{n}{8} = n$ opérations
- ...
- n sous-problèmes de taille 1 : $n \times 1 = n$ opérations

Il reste à connaitre le nombre d'étapes : notre tableau compte n valeurs, on découpe successivement à chaque étape le tableau en deux sous tableaux de même taille. Le nombre d'étapes est donc le plus petit entier k tel que $2^k > n$

Ainsi : $k \approx log_2(n)$

Propriété :
La complexité du tri fusion est donc en : $n \times log_2(n)$.

3.3 Implémentation en Python du tri fusion

```python
import sys
sys.setrecursionlimit(5000)

def Fusion(T1,T2):
    if T1 == [] : return T2
    if T2 == [] : return T1
    if T1[0] < T2[0]:
        return [T1[0]] + Fusion(T1[1:],T2)
    else:
        return [T2[0]] + Fusion(T1,T2[1:])

def TriFusion(T):
    n = len(T)
    if n <=1 : return T
    T1 = T[0:(n//2)]
    T2 = T[(n//2):]
    return Fusion(TriFusion(T1),TriFusion(T2))
```

4 Les exercices

Exercice n° 1.

On considère un problème simple : soit `tab =` $\{a_1; a_2; \ldots; a_n\}$ une liste de valeurs entières croissantes :

$$\forall i \in \mathbb{N},\ \text{tel que} : i \leqslant n :\ a_i \in \mathbb{N},\ \text{et} : a_1 \leqslant a_2 \leqslant \ldots \leqslant a_n$$

On se donne `val` un entier naturel, l'objectif est de tester si le nombre `val` appartient à la liste `tab`.
L'algorithme naturel a une complexité en `n` (on compare successivement chaque élément de la liste à `val`).

1. On considère la fonction `recherche_dichotomique_recursive(val, tab)`, elle appelle la fonction récursive `recherche(val, tab, g, d)` qui renvoie une position de `val` dans `tab[g..d]`, supposé trié, et `None` si elle ne s'y trouve pas.

```python
def recherche_dichotomique_recursive(val, tab):
    """renvoie une position de element dans tableau,
      supposé trié, et None si elle ne s'y trouve pas"""
    return recherche(val, tab, 0, len(tableau)-1)

```

 Exercice_1.py
Coder la fonction `recherche(val, tab, g, d)`.

2. La méthode `tableau_trie(n)` génère et retourne un tableau de n entiers triés dans l'ordre croissant, utiliser la pour tester votre fonction.

Exercice n° 2.

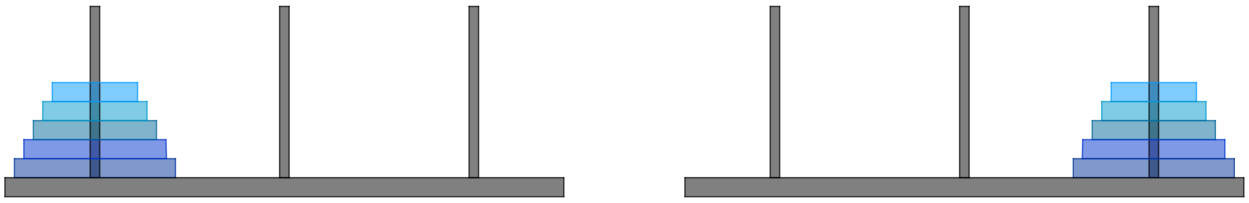
Soit f une fonction continue sur un intervalle $[a\ ;\ b]$ tel que : $f(a) \times f(b) < 0$.
Le théorème des valeurs intermédiaires assure que la fonction f s'annule sur l'intervalle $[a\ ;\ b]$.

Objectif : Trouver une valeur de $x \in [a\ ;\ b]$ telle que $f(x) = 0$.
On ne donnera évidemment pas la valeur exacte de la racine de f, mais une valeur approchée de x. Ainsi, si l'on se donne une précision ϵ souhaitée, on donnera comme valeur approchée de la racine $\dfrac{c+d}{2}$, si l'on sait que f s'annule sur l'intervalle $[c\ ;\ d]$ dont l'amplitude $d - c$ sera inférieure à ϵ.

1. Écrire un algorithme Diviser pour Régner et établir sa complexité temporelle (attention aux problèmes de précision numérique...).
2. Implémenter en Python l'algorithme et tester avec $f(x) = x^2 - 14x + 46$ dont une racine devrait se trouver dans l'intervalle $[7\ ;\ 10]$.

Exercice n° 3.

Le problème des tours de Hanoï est un jeu célèbre constitué de trois tiges verticales sur lesquelles sont empilés des disques de diamètres décroissants.



Il s'agit de déplacer tous les disques de la première tige (dessin de gauche) vers la dernière tige (dessin de droite) en respectant deux contraintes :

- on ne peut déplacer qu'un disque à la fois ;
- On ne peut poser un disque que sur un disque de diamètre plus grand.

Par exemple avec cinq disques, comme sur notre figure, il faudra au minimum 31 déplacements pour y parvenir.

1. Écrire une fonction `hanoi(n)` qui affiche la solution du problème des tours de Hanoï pour `n` disques, sous la forme de déplacements élémentaires désignant les trois tiges 1, 2 et 3, de la manière suivante :

   deplace 1 vers 3

   deplace 1 vers 2

   On commencera par coder une fonction récursive `deplace(a, b, c, k)` qui résout le problème plus général du déplacement de k disques de la tige a à la tige b en se servant de la tige c comme stockage intermédiaire.

2. On peut démontrer par récurrence que le nombre minimum de déplacements pour régler le problème des tours de Hanoï avec n disques est : $2^n - 1$.

   Modifier vos fonctions `deplace(a, b, c, n)` et `hanoi(n)` pour qu'elles prennent en argument une variable `cpt` (qu'on initialisera par défaut à 0 dans la fonction `hanoi`) et que la fonction `hanoi()` renvoie le nombre minimum de déplacements nécessaire pour solutionner le problème des tours de Hanoï avec `n` disques.

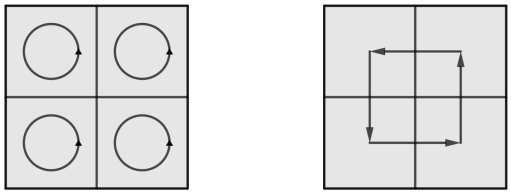
3. Reprendre le problème dans le cas où a, b et c sont des piles.

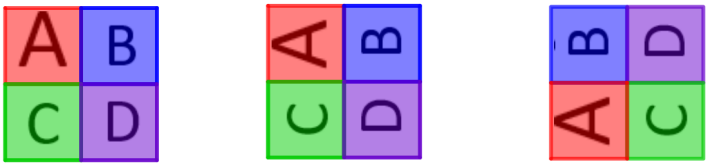
Exercice n° 4.

Dans cet exercice, on cherche à écrire une fonction qui effectue la rotation d'une image de 90 degrés en utilisant le principe "diviser pour régner".
Pour manipuler une image en Python, on peut utiliser par exemple la libraire `PIL` et plus précisément son module `Image`.

```
from PIL import Image
im = Image.open("image.png")
largeur, hauteur = im.size
px = im.load()
```

On charge l'image contenue dans le fichier `image.png`, on obtient ses dimensions `largeur` et `hauteur`, et la variable `px` est la matrice des pixels constituant l'image.
Pour `0` $\leqslant$ `x` $\leqslant$ `largeur` et `0` $\leqslant$ `y` $\leqslant$ `hauteur`, la couleur du pixel `(x, y)` est donnée par `px[x, y]`.
Une couleur est un triplet donnant les composantes rouge, vert et bleu sous la forme d'entiers entre 0 et 255.
On peut modifier la couleur d'un pixel avec une affectation de la forme `px[x, y] = c` où `c` est une couleur.
Dans cet exercice, on suppose que l'image est carrée et que sa dimension est une puissance de 2, par exemple : 256 x 256.
Notre idée consiste à découper l'image en quatre, à effectuer la rotation de 90 degrés de chacun des 4 morceaux, puis à les déplacer vers leur position finale. On peut illustrer ainsi :





Afin de pouvoir procéder récursivement, on va définir une fonction `rotation_aux(px, x, y, t)` qui effectue la rotation de la portion carrée de l'image comprise entre les pixels `(x, y)` et `(x + t, y + t)`. Cette fonction ne renvoie rien. Elle modifie le tableau `px` pour effectuer la rotation de cette portion de l'image, au même endroit. On suppose que `t` est une puissance de 2.

1. Écrire le code de la fonction `rotation_aux()`.
2. En déduire une fonction `rotation(px, taille)` qui effectue une rotation de l'image toute entière, sa dimension étant donnée par le paramètre `taille`. Une fois la rotation effectuée, on pourra sauvegarder le résultat dans un autre fichier avec la commande `im.save("rotation.png)`.

Exercice n° 5.

Dans ce chapitre on se propose d'appliquer le principe "diviser pour régner" pour multiplier 2 entiers, avec la méthode de Karatsuba. Le principe est le suivant. Considérons deux entiers x et y ayant chacun $2n$ chiffres en base 2. On peut les écrire sous la forme :

$$x = a2^n + b$$

$$y = c2^n + d$$

avec a, b, c et d quatre entiers naturels strictement inférieurs à 2^n, autrement dit a, b, c et d sont des entiers qui s'écrivent sur n chiffres en base 2. Dès lors, on peut calculer le produit $x \times y$ de la manière suivante :

$$\begin{aligned} xy &= (a2^n + b)(c2^n + d) \\ xy &= ac2^{2n} + (ad + bc)2^n + bd \\ xy &= ac2^{2n} + (ac + bd - (a-b)(c-d))2^n + bd \end{aligned}$$

Cette dernière ligne qui semble inutilement compliquée a le mérite de ne faire apparaitre que trois produits ac, bd et $(a-b)(c-d)$ (et pas quatre).
Ainsi, on a ramené la multiplication de deux entiers de $2n$ chiffres à trois multiplications d'entiers de n chiffres.
Pour faire chacune des ces trois multiplications, on peut appliquer le même principe jusqu'à obtenir de petits entiers dont la multiplication est immédiate. au final cela permet d'effectuer la multiplication en un temps proportionnel à $n^{1,58}$ environ au lieu de n^2. Ce qui est important pour de grandes valeurs de n.

1. Écrire une fonction `taille(x)` qui renvoie le nombre de chiffres de l'entier x en base 2.
2. Écrire une fonction `Karatsuba(x, y, n)` qui calcule le produit de x et y par la méthode de Karatsuba, en supposant que x et y s'écrivent avec n chiffres en base 2.
   Indications : on peut calculer 2^n en python avec l'expression `1<<n`, et décomposer x sous la forme $a2^n + b$ avec l'instruction : `a, b = x>>n, x % (1<<n)`.
3. En déduire une fonction `mult(x, y)` qui calcule le produit de x et y.
4. Tester votre fonction avec les tests mis en place dans l'exercice 4 du chapitre précédent.

Remarque : en pratique, on n'observera pas de progrès par rapport à la multiplication de Python, car celle ci repose déjà sur une bibliothèque de grands entiers qui implémente des algorithmes efficaces tels que Karatsuba (et même de plus efficace tels que Toom-Cook). Pour une comparaison honnête, il faut écrire une fonction comparable à la notre mais de complexité n^2, par exemple reposant sur l'identité :

$$xy = ac2^{2n} + (ad + bc)2^n + bd$$