new idea

REDUCTIONS

Idée clé : la vue Boîte Noire d'un fonction

En tant qu'**utilisateur** d'une fonction, le fonctionnement interne ne nous importe pas. Seule l'**interface** nous intéresse

Une fois qu'une fonction est écrite, on doit pouvoir l'utiliser simplement en connaissant son **nom**, ses **paramètres** et sa **valeur de retour**.

Idée clé : la vue Boîte Noire d'un fonction

```python
def calculer_moyenne(x, y):
    """Calcule la moyenne de deux nombres."""
    total = x + y
    moyenne = total / 2
    return moyenne
```

L'utilisateur n'a pas besoin de connaître la variable `total`.

Il n'a pas besoin de savoir qu'on additionne puis qu'on divise.

Il fait confiance à la boîte pour faire son travail.

L'implémentation est sans importance pour l'utilisateur.

Bénéfice 1 : Modularité

Les fonctions permettent de **décomposer** un grand problème en sous-problèmes plus petits et gérables.

Bénéfice 2 : Réutilisabilité

Écrivez le code une fois, mettez-le dans une fonction, et appelez-le plusieurs fois.

Bénéfice 3 : Testabilité et Débogage

Les boîtes noires peuvent être testées en **isolation**.

Le Problème à Résoudre: Problème A

Nous disposons de plusieurs **boîtes noires** qui résolvent des problèmes spécifiques :

f_1 résout problème 1	f_2 résout problème 2	f_3 résout problème 3

Peut-on combiner ces boîtes noires pour résoudre le problème A ?

Utiliser les fonctions $f_1, f_2, f_3, \ldots$ comme des **briques élémentaires**

Réduction = Résoudre un problème en le transformant en un problème déjà résolu.

Problem: donnez deux réels a et b, calculez $\max(a, b)$ *sans comparaisons.*

Boîte noire disponible: la fonction $\min(a, b)$ pour calculer leur minimum.

Solution 1: $$\max(a, b) = \frac{a \cdot b}{\min(a, b)}, \text{ si } \min(a, b) \neq 0$$

Solution 2: $$\max(a, b) = a + b - \min(a, b)$$

```
a, b = 9, -11

mymax = a*b / min(a,b)
print("Max is: ", mymax)
```

↓

```
Max is:  9.0
```

```
a, b = 9, -11

mymax = a + b - min(a,b)
print("Max is: ", mymax)
```

↓

```
Max is:  9
```

Problem: trouvez le Plus Petit Commun Multiple (PPCM) de deux nombres.

Boîte noire disponible: la fonction $\gcd(a, b)$ pour calculer le pgcd

Idée clé: $$\text{ppcm}(a, b) = \frac{a \cdot b}{\text{pgcd}(a, b)}$$

Preuve:

Soient $a = \prod_i p_i^{a_i}$ et $b = \prod_i p_i^{b_i}$ les décompositions en facteurs premiers.

$$\gcd(a,b) = \prod_i p_i^{\min(a_i,b_i)}, \qquad \operatorname{lcm}(a,b) = \prod_i p_i^{\max(a_i,b_i)},$$

$$ab = \prod_i p_i^{a_i+b_i} = \prod_i p_i^{\min(a_i,b_i)+\max(a_i,b_i)} = \gcd(a,b) \cdot \operatorname{lcm}(a,b).$$

```
import math

a, b = 14, 5
print("lcm: ", a*b / math.gcd(a,b))

a, b = 14, 4
print("lcm: ", a*b / math.gcd(a,b))
```

→

```
lcm:  70.0
lcm:  28.0
```

Problem: déterminer si deux chaînes de caractères sont des anagrammes.

Boîte noire disponible: string ⟶ sorted ⟶ liste triée

list ⟶ join ⟶ string

Idée clé: Deux chaînes sont des anagrammes si leurs versions triées sont identiques.

```
s, t = "abfecgd", "fgdcba"
s_trie, t_trie = ''.join(sorted(s)), ''.join(sorted(t))

print(s, t, s_trie == t_trie)

s, t = "abfecgd", "efgdcba"
s_trie, t_trie = ''.join(sorted(s)), ''.join(sorted(t))

print(s, t, s_trie == t_trie)
```

↓

```
abfecgd fgdcba False
abfecgd efgdcba True
```

Problem: vérifier si une chaîne est une rotation d'une autre (`abcde` et `cdeab`), *sans boucles*

Boîte noire disponible: strings `s`, `t` ⟶ in ⟶ `True` ssi `s` est une sous-chaîne de `t`

Idée clé: `s` est une rotation de `t` ssi `s` est une sous-chaîne de `t+t`

Si `u = t + t`, alors `u` contient toutes les rotations de `t` comme sous-chaînes contiguës.
En effet, une rotation de `t` par i positions correspond à la sous-chaîne `u[i : i + len(t)]`.

```
t = "abcde"
u = t + t

for i in range(0, len(t)):
    print(u[i:i+len(t)])
```

↓

```
abcde
bcdea
cdeab
deabc
eabcd
```

```
t, s = "abcde", "cedab"
print(s, t, s in (t+t))

t, s = "abcde", "cdeab"
print(s, t, s in (t+t))
```

↓

```
cedab abcde False
cdeab abcde True
```

Problem: Trouver la taille l de string la plus courte qui contient deux strings `s` et `t` comme sous-séquences.

Boîte noire disponible: strings `s`, `t` ⟶ PLSC ⟶ taille de la plus longue sous-séquence commune

Idée clé: `l = len(s) + len(t) - len(PLSC(s,t))`

Une esquisse de preuve :

Construire une string `u` qui partage tous les caractères de `s` et `t` présents dans leur chaîne PLSC, et insérer séparément tous les caractères restants de `s` et `t` dans `u`. Donc,

$$l \leq \texttt{len(u)} = \texttt{len(s)} + \texttt{len(t)} - \texttt{len(PLSC(s,t))}$$

Soit `A` la liste des caractères dans la solution (de taille `l`) auxquels sont associés à la fois un caractère de `s` et un caractère de `t`. Alors `A` est une sous-chaîne de `s` et de `t`. Donc

$$\texttt{l} = \texttt{len(A)} + (\texttt{len(s)} - \texttt{len(A)}) + (\texttt{len(t)} - \texttt{len(A)}) \geq \texttt{len(s)} + \texttt{len(t)} - \texttt{PLSC(s,t)}$$

```
s:   A B C D G H
t:   A E D F H R
PLSC: A - D - H
SCS: A B C D E G H F R
```

Problem: étant donné une liste `A` d'entiers non triés, trouvez la médiane dans `A`.

Boîte noire disponible: liste `A` ⟶ sorted ⟶ liste `A` triés

Direct algorithme:

```
def findMedian(A):
    n = len(A)
    for i in range(0, n):
        count = 0
        for j in range(0, n):
            if A[j] < A[i]:
                count += 1

        if count == n // 2:
            return A[i]
```

Reduction à tri :

```
def findMedianFASTER(A):
    B = sorted(A)
    return B[ len(A) // 2]
```

En fait, `sorted` est beaucoup plus rapid!

```
def findMedianFASTER(A):
    B = sorted(A)
    return B[ len(A) // 2]


def findMedian(A):
    n = len(A)
    for i in range(0, n):
        count = 0
        for j in range(0, n):
            if A[j] < A[i]:
                count += 1

        if count == n // 2:
            return A[i]

import random
import time

n = 1000001
A = [ random.randint(0, 100000) for _ in range(0, n) ]

start = time.time()
m2 = findMedianFASTER(A)
print(f"Median: {m2} in time for f(n): {time.time()-start}")

start = time.time()
m1 = findMedian(A)
print(f"Median: {m1} in time for n^2: {time.time()-start}")
```

Un mathématicien doit faire chauffer de l'eau. Il prend une bouilloire vide, la remplit, la branche, l'eau bout.

Le lendemain, on lui donne une bouilloire déjà pleine. Il la vide dans l'évier et dit : Je ramène le problème à un problème déjà résolu , puis il la remplit à nouveau ...

Résumé : La Mentalité Boîte Noire

Abstraction : Traitez chaque fonction comme une boîte noire. Concentrez-vous sur *ce qu'*elle fait, pas *comment*.

Modularité : Décomposez les grands problèmes en petites fonctions indépendantes.

Réduction : Résolvez de nouveaux problèmes en réutilisant des solutions à d'anciens problèmes.

Composition : Construisez des programmes complexes en connectant des fonctions entre elles, en passant des données de l'une à l'autre.