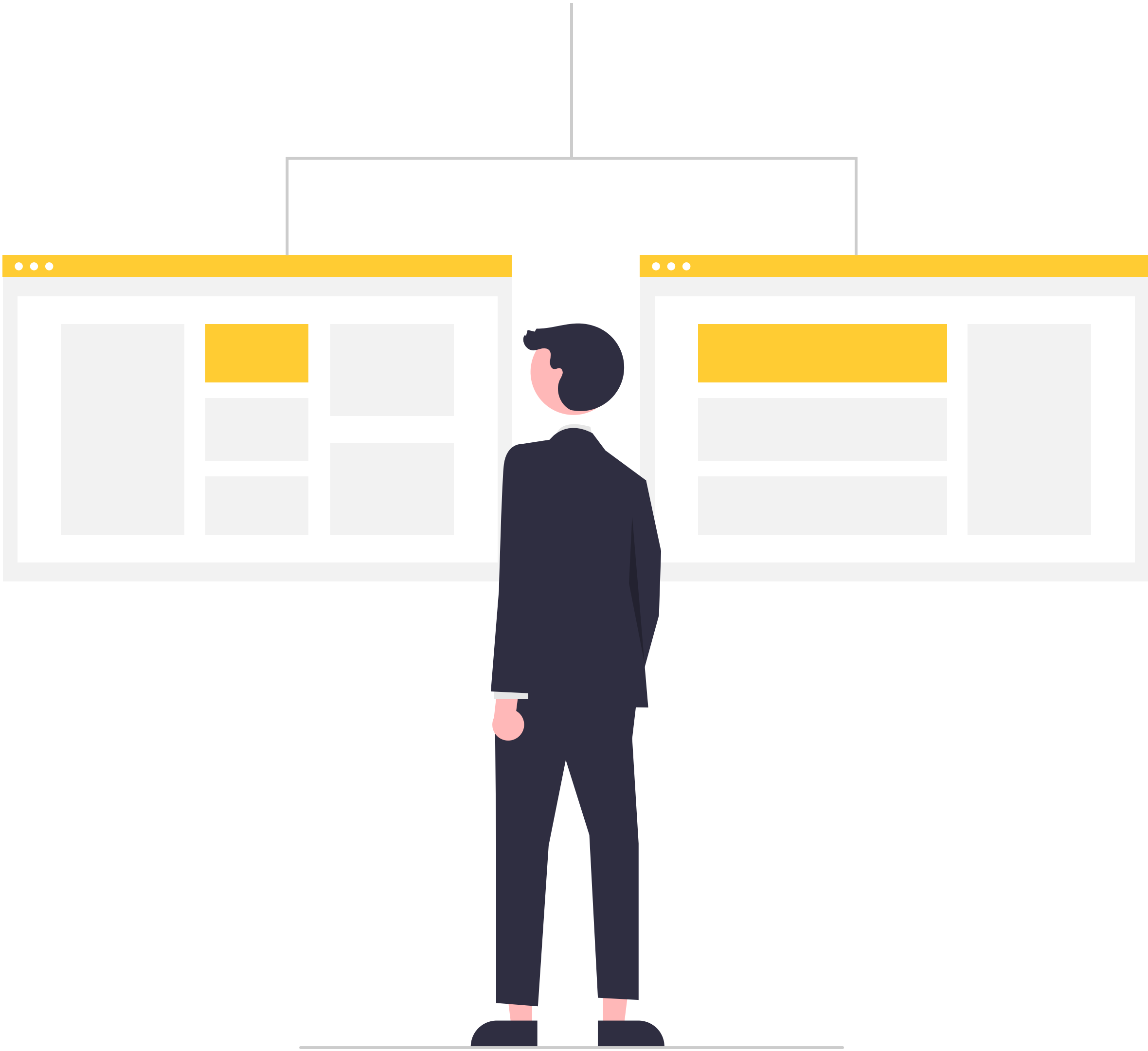
TD20 Diviser pour Regner



Le principe

La stratégie « diviser pour régner » consiste à

1. Décomposer un problème en un ou plusieurs *sous-problèmes* de même nature mais plus petits.
2. Résoudre les sous-problèmes, généralement de manière récursive, jusqu'à ce qu'on arrive aux *cas d'arrêt* : des sous-problèmes que l'on sait résoudre immédiatement.
3. Construire la solution au problème initial à partir des solutions des sous-problèmes.

Le formidable exemple des tours de Hanoï

On dispose de n palets empilés du plus grand au plus petit sur une tige numéro 1, on veut déplacer cette tour sur la tige numéro 3 en se servant de la tige numéro 2 comme intermédiaire, sachant que :

- on ne peut déplacer qu'un palet à la fois;
- on ne peut mettre un palet que sur une tige vide ou sur un palet plus gros.

Appelons `hanoi(n,T1,T2,T3)` la succession des mouvements nécessaires pour résoudre ce problème.

- `hanoi(0,T1,T2,T3)` est évident : il suffit de ne rien faire.
- `hanoi(1,T1,T2,T3)` est simple : déplacer le palet de la tige numéro 1 à la tige numéro 3.

- Si l'on connait `hanoi(n-1)` alors pour effectuer `hanoi(n)` on peut :
    - déplacer la tour de taille $n - 1$ de la tige numéro 1 à la tige numéro 2 en utilisant la tige numéro 3 comme intermédiaire, autrement dit appliquer `hanoi(n-1,T1,T3,T2)` ;
    - déplacer le gros palet de la tige numéro 1 à la tige numéro 3;
    - déplacer la tour de taille n - 1 de la tige numéro 2 à la tige numéro 3 en utilisant la tige numéro 1 comme intermédiaire, autrement dit appliquer `hanoi(n-1,T2,T1,T3)` . La méthode est illustrée ci-dessous

Ce qui se traduit par l'algorithme suivant :

```
fonction hanoi(n : entier naturel, T1 : pile, T2 : pile, T3 : pile)

    # hanoi(nb_palets, départ, intermédiaire, destination)

    si n = 0 alors
        ne rien déplacer
    sinon
        hanoi(n - 1, T1, T3, T2)
        déplace palet de T1 vers T3
        hanoi(n - 1, T2, T1, T3)
    finSi
```

Formidable, non ?

EX : Codage

Coder la fonction `hanoi` en python :

```
def hanoi(n : int, depart : str, inter : str, dest : str) -> None:
"""
Affiche la liste des mouvements pour deplacer une tour de
hauteur n de la tige depart vers la tige dest en utilisant
la tige inter.
Cette fonction n'utilise que des print, aucune liste ou
quoi que ce soit d'autre.
"""
```

Comme expliqué dans la `docstring`, la fonction ne fait rien de concret, elle se contente d'afficher les déplacements de palets d'une tige à l'autre.

Tester cette fonction : `hanoi(3, 'A', 'B, 'C')` doit afficher :

```
Déplacer un palet de A vers C
Déplacer un palet de A vers B
Déplacer un palet de C vers B
Déplacer un palet de A vers C
Déplacer un palet de B vers A
Déplacer un palet de B vers C
Déplacer un palet de A vers C
```

Déroulement de l'algorithme avec une tour de 4 palets

Le tri fusion

Principe

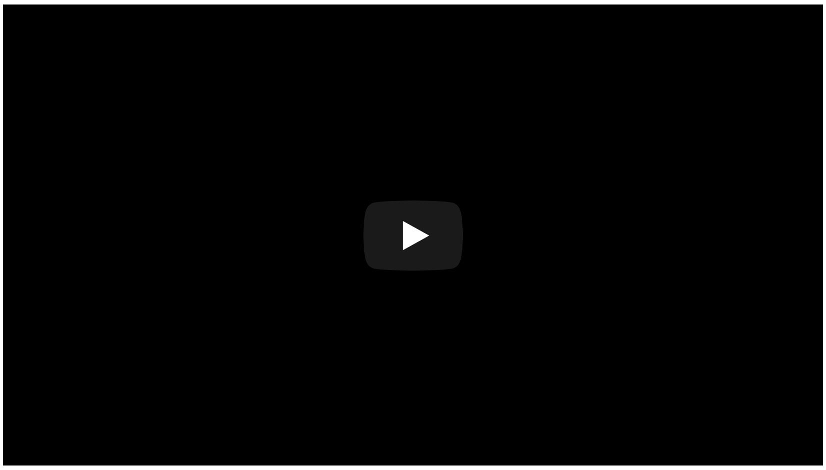
Voici un deuxième exemple d'application de cette stratégie : le *tri fusion*. On doit cet algorithme à John Von Neumann.

On dispose d'une liste d'entiers que l'on veut trier dans l'ordre croissant.

1. On scinde cette liste en deux listes de longueurs « à peu près égales ».

2. On trie ces listes en utilisant... le tri fusion.

3. On *fusionne* les deux listes triées par ordre croissant pour ne plus en obtenir qu'une.



Une chorégraphie du tri fusion

Voici comment coder le tri fusion :

On a tout d'abord besoin d'une fonction `scinde` qui renvoie la première moitié et la deuxième moitié de la liste qu'on lui passe en argument.

```
def scinde(lst: list) -> tuple:
    return lst[:len(lst) // 2], lst[len(lst) // 2:]
```

Ensuite, on a besoin d'une fonction `fusion` qui, étant donnée deux listes triées, les fusionne.

```
def fusion(lst1: list, lst2: list) -> list:
    if not lst1 or not lst2: # si l'une des listes est vide
        return lst1 or lst2 # alors on renvoie l'autre
    else:
        a, b = lst1[0], lst2[0]
        if a < b : # sinon on compare leurs premiers éléments
            return [a] + fusion(lst1[1:], lst2) # on place le plus petit en têt
        elif b > a:
            return [b] + fusion(lst1, lst2[1:])
        else : # dans le cas où les 2 éléments sont égaux on peut les placer to
            return [a, b]+ fusion(lst1[1:], lst2[1:])
```

Enfin, la fonction `tri_fusion`.

```
def tri_fusion(lst: list) -> list:
    if len(lst) < 2: # cas d'arrêt
        return lst
    lst1, lst2 = scinde(lst) # sinon on scinde
    return fusion(tri_fusion(lst1), tri_fusion(lst2)) # et on fusionne les sous
```

Complexité du tri fusion

Notons n la taille de la liste à trier et considérons comme seule *opération élémentaire* le fait d'accéder à un élément d'une liste.

En classe de Première, nous avons étudié des algorithmes de tri dits « lents », car de complexité *quadratique* : pour une liste de taille n, le nombre d'opérations élémentaires pour trier ce tableau est « de l'ordre de n^2 ».

Ainsi, pour trier une liste de 10^6 entiers avec le tri par sélection (par exemple), le nombre d'opérations élémentaires nécessaires est de l'ordre de 10^{12}.

Complexité du tri fusion

Le nombre d'opérations élémentaires nécessaires pour trier une liste de taille n par la méthode du tri fusion est de l'ordre de $n \times \ln n$.

$\ln n$ est le *logarithme néperien de* n mais peut être remplacé par le logarithme en base 2 ou en base 10 sans changer les ordres de grandeur.

Exemple

Pour trier une liste de 10^6 entiers, il faudra de l'ordre de 6 millions d'opérations élémentaires.

EX : Vérification expérimentale

On peut vérifier ce résultat expérimentalement ici.

EX : Recherche dichotomique

Enoncé

1. Expliquer pourquoi la recherche dichotomique d'un élément dans une liste d'entiers triés dans l'ordre croissant peut être vue comme un exemple de stratégie « diviser pour régner ».

2. Programmer la recherche dichotomique de manière récursive.

Solution

Pour savoir si un élément appartient à la liste, on regarde celui qui est «à peu près au milieu». Si c'est le bon c'est terminé, sinon on fait de même avec la sous-liste des éléments précédents et avec celle des éléments suivants.

```
def rech_dicho(lst, elt):
    n = len(lst)
    if n < 2:
        return elt in lst
    elif lst[n // 2] == elt:
        return True
    else:
        return rech_dicho(lst[:n // 2], elt) if lst[n // 2] > elt e
```

EX : Algorithme de Karatsuba

Il s'agit d'un algorithme qui applique la stratégie « diviser pour régner » pour effectuer des multiplications de manière efficace.

1. Chercher sur Internet ce qu'est cet algorithme.
2. Programmer cet algorithme en Python.

▼ Indices

Indice : principe

Si x et y s'écrivent **au plus** avec $2n$ bits, alors on peut les écrire $\begin{cases} x & = a \times 2^n + b \\ y & = c \times 2^n + d \end{cases}$, où a, b, c et d s'écrivent **au plus** n bits.

Mais alors $x \times y = ab2^{2n} + (ac + bd - (a-b)(c-d))2^n + bd$.

Dans cette écriture, il y a 3 multiplications à faire avec des nombres s'écrivant avec **au plus** n bits:

- ac
- bd
- $(a-b)(c-d)$

Le reste (additions et multiplication par 2^{2n} ou 2^n ne prend pas beaucoup de temps à faire (les additions sont plus rapides que les multiplications et multiplier un nombre par 2^n revient à lui ajouter n bits valant 0 à droite).

La stratégie « diviser pour régner » vient du fait qu'on calcule ces 3 produits en appliquant de nouveau l'algorithme de Karatsuba.

Indice de programmation 1

On peut déjà code une fonction `size` qui - en entrée prend un `int` x; - renvoie le nombre de bits de l'écriture binaire de x. Pour ce faire il suffit de diviser x par 2 (avec `//`) jusqu'à trouver 0.

Indice de programmation 2

- Pour multiplier `x` par 2^n on peut utiliser l'opérateur `<<` : `x << n`.
- Pour diviser, utiliser `>>`.
- Ainsi on pourra écrire

```
a = x >> (2 ** n)
b = x % (2 ** n)
```

Et caetera.

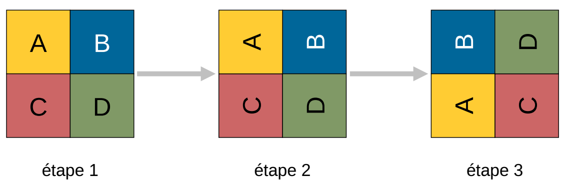
Rotation d'une image carrée d'un quart de tour

On suppose qu'on dispose d'une image carrée de côté n pixels, où n est une puissance de 2.

Pour l'exemple voic une photo (colorisée) carrée d'Alan Turing de côté 512 pixels.

On aimerait faire subir un quart de tour à cette photo (dans le sens antihoraire) en utilisant une stratégie de type «diviser pour régner». On va procéder ainsi :

1. On partage l'image en 4 carrés de côté deux fois moindre.
2. On fait tourner ces 4 carrés.
3. On fait subir une *permutation circulaire* aux 4 carrés



À l'étape 2, pour faire tourner les 4 carrés, on se retrouve avec le même problème mais avec des carrés de côté 2 fois plus petits. On répète donc le processus jusqu'à n'avoir plus que des carrés de côté 1 pixel (sur lesquels il n'y a pas besoin de faire quoi que soit).

Voici ce que cela donne

EX : Programmation de la rotation d'un quart de tour d'une image carrée

On peut la retrouver ici.