Méthode : "Diviser pour régner"

1 Recherche dichotomique dans un tableau trié

Nous avons étudié en Première la recherche dichotomique dans un tableau trié.

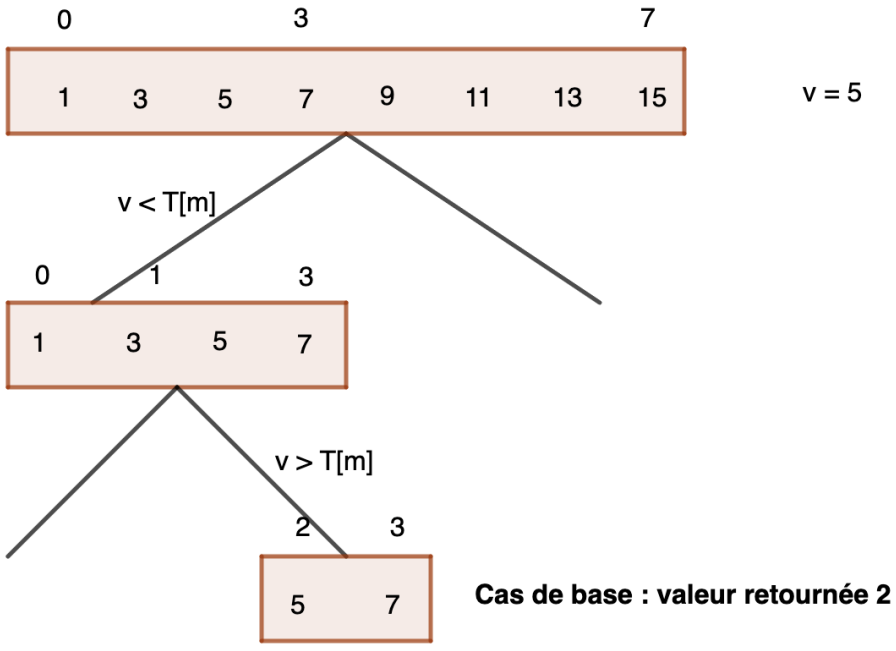
Cette année nous allons reprendre cette technique mais cette fois ci de manière récursive

Nous allons aussi mettre en évidence la méthode sous-jacente appelée **"diviser pour régner"**

La **relation de récurrence** que nous allons traduire en **fonction récursive** est

"Si $x \in [a, b[$ alors ou bien $x \in [a, m[$ ou bien $x \in [m, b[$ avec $m = \dfrac{a+b}{2}$

Regardons cela sur un exemple



d'où la fonction récursive

```
def recherche(t,v,g,d):
    if d == g + 1:
        if v == t[g]:
            return g
        else:
            return None
    m = (g + d)//2
    if v < t[m]:
        return recherche(t,v,g,m)
    else:
        return recherche(t,v,m,d)

def recherche_dicho_rec(t,v):
    return recherche(t,v,0,len(t) - 1)
```

Coût en temps

Si la longueur de la liste triée est 2^n avec $n \geqslant 2$ par dichotomie on atteint le cas de base en $n-1$ étapes car $2^n = 2^{n-1} \times 2^1$ (la longueur du tableau du cas de base est 2^1)

On peut donc écrire que la complexité est de l'ordre de $\ln_2(n)$ ce qui est meilleure que la recherche séquentielle dans une liste **non triée** dont la complexité est de l'ordre de n

2 Tri fusion

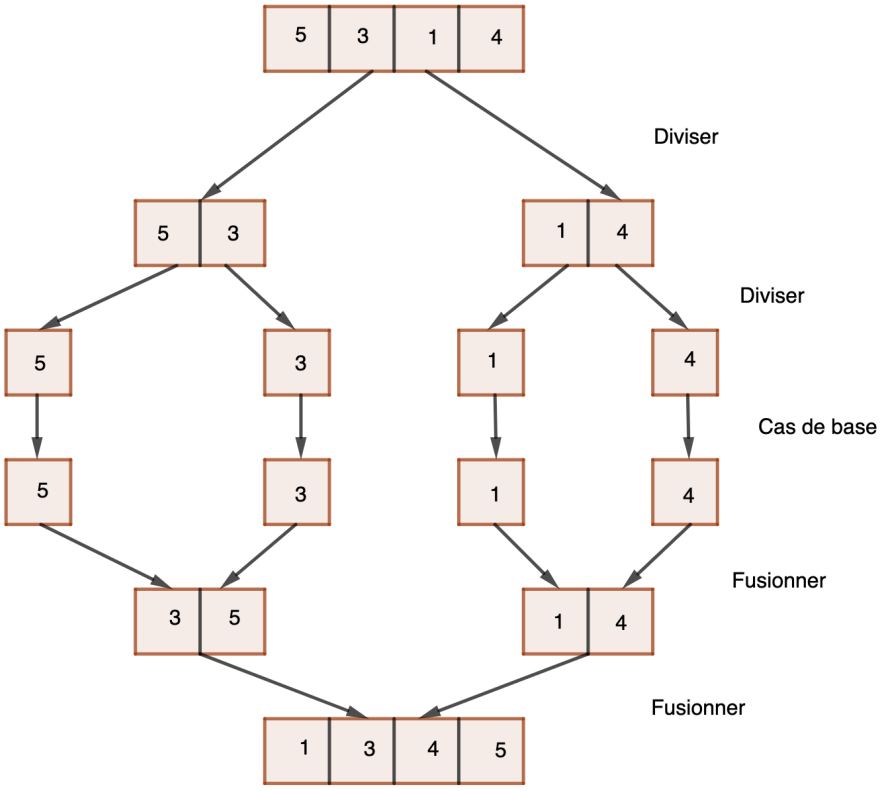
On reprend l'idée de dichotomie précédente

La relation de récurrence devient :

"Trier la liste L revient à trier les deux moitiés de liste de L puis à fusionner les deux moitiés de liste triées en une liste triée

Le cas de base est une liste de longueur 1 dans ce cas la valeur retournée est le seul élément de la liste"

Regardons cela sur un exemple



D'où la traduction en fonction récursive, en supposant que l'on dispose d'une fonction `fusion(L1,L2)` qui fusionne deux listes L1 et L2 triées dans l'ordre croissant en une nouvelle liste triée dans l'ordre croissant

```
def tri_fusion(L):
    n = len(L)
    if len(L) == 1:
        return L
    return fusion(tri_fusion(L[0:n//2]),\
        tri_fusion(L[n//2:]))
```

Nous avons utilisé les outils disponibles avec le type list de Python

Coût en temps

(Pour les élèves qui font la spécialité Maths en Terminale ou l'option maths complémentaires il est intéressant de voir un exemple d'étude de suites récurrentes pour la détermination de

l'ordre de grandeur de la complexité du tri fusion)

Supposons que le coût en temps est une suite c_n qui dépend de la taille n de la liste à trier

Supposons que le coût de la fonction fusion est linéaire, et on supposera pour simplifier que n est une puissance de 2

Dans ce cas on peut écrire la relation de récurrence

$c_{2^k} = 2^k + 2c_{2^{k-1}}$ avec $c_1 = 1$

On décline plusieurs fois cette relation pour **éliminer les termes intermédiaires**

$c_{2^k} = 2^k + 2c_{2^{k-1}}$

$2c_{2^{k-1}} = 2^k + 2^2 c_{2^{k-2}}$

$2^2 c_{2^{k-2}} = 2^k + 2^3 c_{2^{k-3}}$

....

$2^{k-2} c_2 = 2^k + 2^{k-1} c_1$

Ajoutons ces $k-1$ lignes il reste

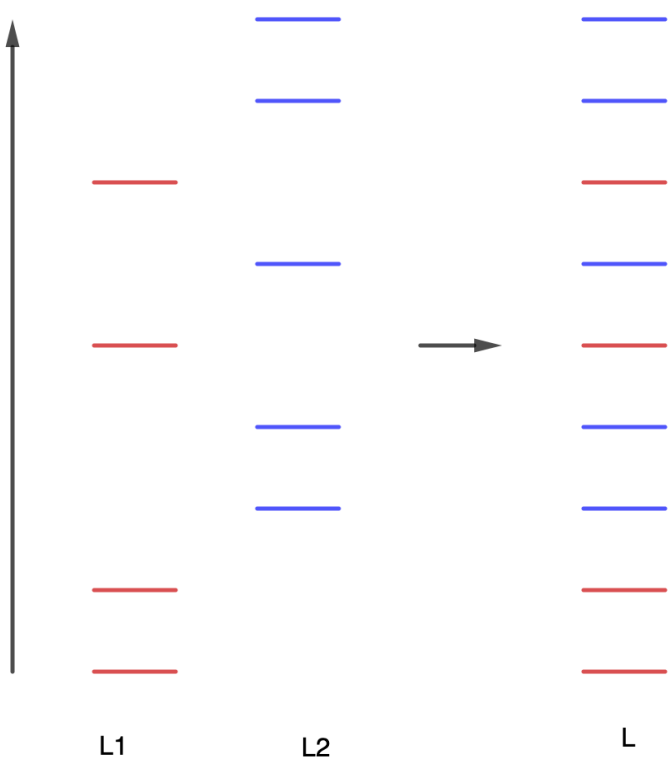
$c_{2^k} = (k-1)2^k + 2^{k-1} = (k - \frac{1}{2})2^k = n(\ln_2(n) - \frac{1}{2}) = n\ln_2(n) - \frac{n}{2}$

On ne conserve que la partie "dominante" quand "n devient grand" autrement dit la complexité du tri fusion a pour ordre de grandeur

$n\ln_2(n)$, meilleur que celui des tris par insertion et sélection dont l'ordre de grandeur est n^2

Pour avoir cette complexité il nous faut un algorithme de fusion **linéaire**

2.1 Fusion de deux listes triées en une liste triée



Représentons les deux listes comme des traits relativement à un repère et on comprend qu'il suffit de parcourir les deux listes en une fois et faire autant de comparaisons entre les éléments des deux listes d'où la linéarité de la complexité en fonction de la somme des deux tailles des deux listes

```
def fusion(L1,L2):

    i1 = 0
    i2 = 0
    i = 0
    L = [0]*(len(L1)+len(L2))

    while i1 < len(L1) and i2 < len(L2):
        if L1[i1] < L2[i2]:
            L[i] = L1[i1]
            i += 1
```

```
            i1 += 1
        else:
            L[i] = L2[i2]
            i += 1
            i2 += 1
    for j in range(i1,len(L1)):
        L[i] = L1[j]
        i += 1
    for j in range(i2,len(L2)):
        L[i] = L2[j]
        i += 1
    return L
```

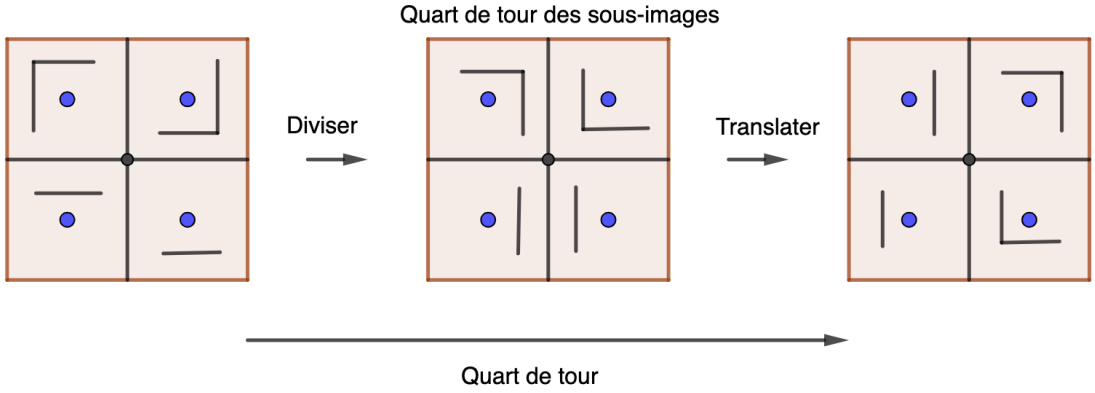
3 Rotation d'une image d'un quart de tour

On veut tourner d'un quart de tour dans le sens horaire une image carrée dont le côté est une puissance de 2

Par exemple , voici une image carrée de côté 512 pixels

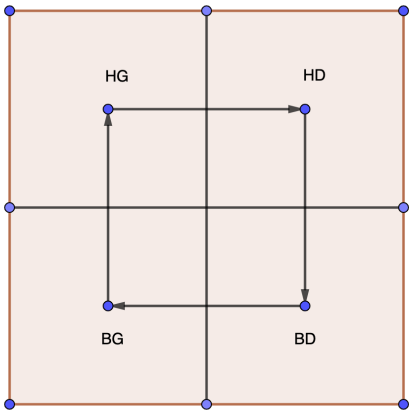
après une rotation de 90 degrés dans le sens horaire on obtient

La récurrence est basée sur l'idée suivante :

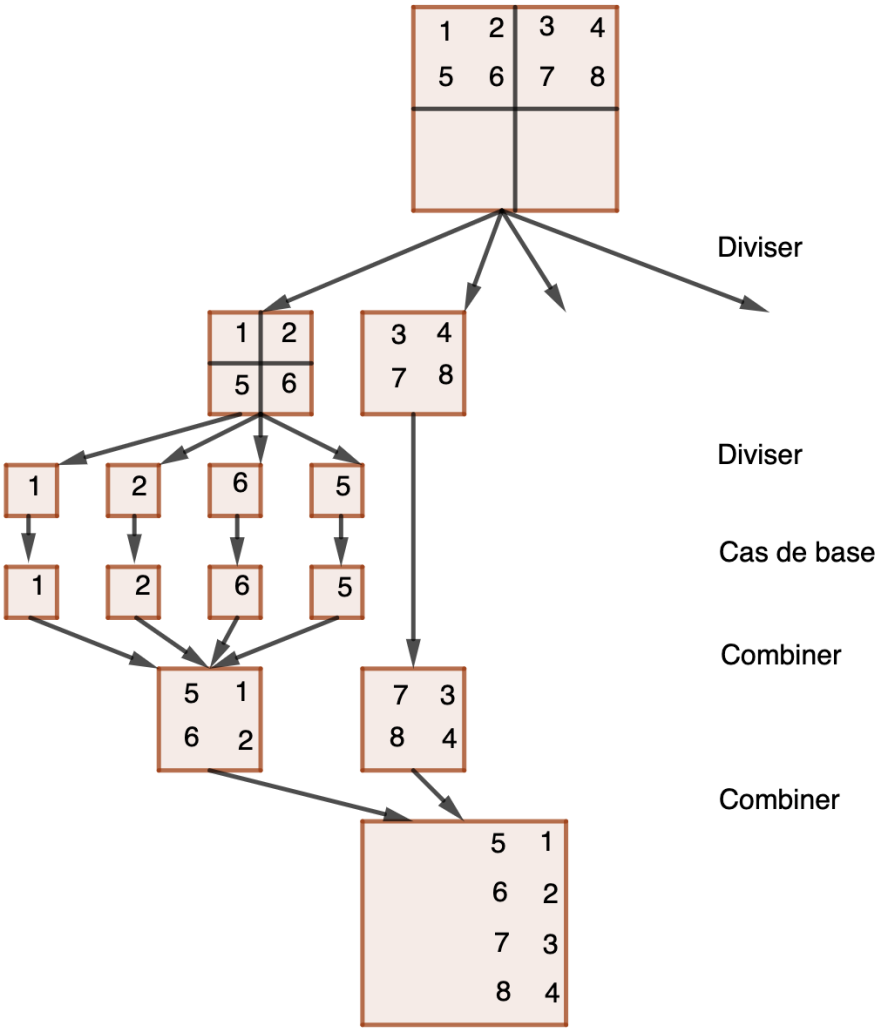


"Faire tourner une image carré de 90 degrés **autour de son centre, dans le sens horaire** revient à :

1. Diviser l'image en quatre images carrés HG (haut gauche), HD, BG et BD
2. Faire tourner les quatre sous-images **autour de leur centre respectif**
3. Translater HG vers HD puis HD vers BD puis BD vers BG et enfin BG vers HG
4. Pour le cas de base on ne fait rien



Regardons cela sur un exemple pour un tableau de 16 pixels où les nombres représentent une intensité pour un pixel



D'où les fonctions suivantes pour un tableau T à deux dimensions que l'on adaptera pour une image en TP (bibliothèque PIL)

On définit une fonction `quart_tour(T,x,y,t)` qui prend en paramètres un tableau T à deux dimensions dont le nombres de lignes et de colonnes est une puissance de 2, (x,y) sont les indices de ligne et de colonne permettant de repérer le **coin supérieur gauche** du bloc de T de côté t sur lequel on fait le quart de tour.

t est le côté du bloc.

Par exemple sur un tableau 16 x 16 :

Le premier appel est `quart_tour(T,0,0,16)`

Si (x,y) repère le coin supérieur gauche du bloc HG alors

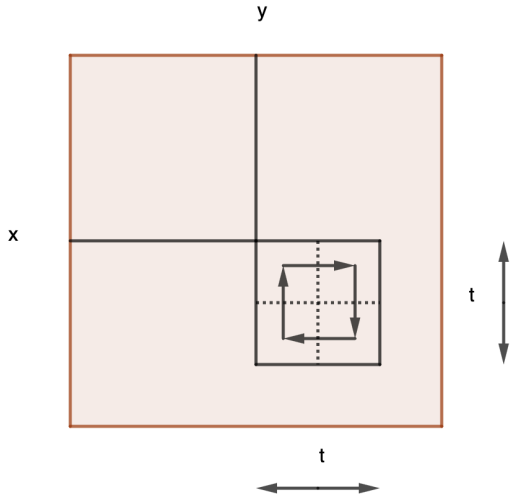
1. (x,y + t//2) repère le coin supérieur gauche du bloc HD.
2. (x + t//2,y) repère le coin supérieur gauche du bloc BG.
3. (x + t//2,y + t//2) repère le coin supérieur gauche du bloc BD.

D' où la fonction `quart_tour(T,x,y,t)`

```
def quart_tour(T,x,y,t):
    if t > 1:
        # diviser
        quart_tour(T,x,y,t//2)
        quart_tour(T,x,y+t//2,t//2)
        quart_tour(T,x+t//2,y+t//2,t//2)
        quart_tour(T,x+t//2,y,t//2)

        # combiner
        translation(T,x,y,t)
```

Il reste à coder une fonction qui déplace les blocs
HG -> HD, HD -> BD, BD -> BG et BG -> HG.
dans un sous-bloc repéré par (x,y) et de taille t dans le tableau T



```
def translation(T,x,y,t):
    for i in range(t//2):
        for j in range(t//2):
            temp = T[x+i][y+j]
            T[x+i][y+j]   = T[x+i+t//2][y+j]
            T[x+i+t//2][y+j]   = T[x+i+t//2][y+j+t//2]
            T[x+i+t//2][y+j+t//2]   = T[x+i][y+j+t//2]
            T[x+i][y+j+t//2]   = temp
```

Attention !

Contrairement au tri fusion qui apporte une réelle amélioration dans l'efficacité des tris par rapport aux tri sélection et insertion, **le quart de tour avec la méthode "diviser pour régner" ici n'est pas optimal en terme de coût en temps** et ne sert ici qu'à mettre en oeuvre la méthode "diviser pour régner".

Voici par exemple une fonction qui est "un peu plus efficace"

```
def quart_tour2(T):
    temp = deepcopy(T)
    for lig in range(len(T)):
        for col in range(len(T)):
            T[col][len(T) - 1 - lig] = temp[lig][col]
```

4 Diviser pour régner

En conclusion la méthode **diviser pour régner** consiste à :

1. **diviser** un problème de taille N en problèmes de taille beaucoup plus petite que N par exemple N//2
2. **résoudre** les problèmes de taille inférieure

3. **combiner** les solutions des sous-problèmes pour obtenir la solution du problème de taille supérieure

5 Exercices

Ex 1

Est il possible que la fonction récursive `recherche(t,v,g,d)` soit empilée plus de 1000 fois sur la pile ?

Ex 2

Proposer une solution pour fusionner les deux listes L1 et L2 triées dans l'ordre croissant en une nouvelle liste triée dans l'ordre croissant

Ex 3

Est il possible que la fonction récursive `tri_fusion(L)` soit empilée plus de 1000 fois sur la pile ?

Si oui proposer une solution itérative de cette fonction

Ex 4

1. Définir la récurrence pour le quart de tour **dans le sens antihoraire** par une image
2. Redéfinir la fonction `translation(T,x,y,t)`

Ex 5

1. Définir une fonction "naïve" `puissance(x,n)` qui calcule x^n avec x entier naturel et n entier naturel. Evaluer la complexité
2. Proposer une solution utilisant la méthode diviser pour régner

Défi

Proposer **votre** solution utilisant la méthode diviser pour régner pour calculer le nombre d'inversions dans une permutation