Cours 3 : Complexité des algorithmes

Objectifs :

— Définir la notion de complexité d'un algorithme.
— S'interroger sur l'efficacité temporelle d'un algorithme.
— Distinguer par leurs complexités deux algorithmes résolvant un même problème.
— Implanter des algorithmes de recherches et de tris.

1 Notion de complexité

La complexité d'un algorithme peut se définir de deux manière différentes : la complexité en temps et la complexité en mémoire.

La complexité en temps peut être évaluée par la durée d'exécution de l'algorithme qui dépend de la taille des données gérées et des instructions à exécuter. La complexité en mémoire peut elle être évaluée par le nombre d'octets de mémoire nécessaire pour exécuter l'algorithme qui dépend là encore de la taille des données gérées et des instructions.

On admet une sorte d'équivalence temps mémoire, où un gain de temps pour un algorithme donné ne peut se faire que par une consommation plus gourmande de la mémoire.

1.1 Coût d'un algorithme

Pour déterminer le coût d'un algorithme, on se fonde en général sur le *modèle de complexité* suivant.

— Une affectation, une comparaison ou l'évaluation d'une opération arithmétique ayant en général un faible temps d'exécution, celui-ci sera considéré comme l'unité de mesure du coût d'un algorithme.
— Le coût des instructions `p` et `q` en séquence est la somme des coûts de l'instruction `p` et de l'instruction `q`.
— Le coût d'un test `if b : p else : q` est inférieur ou égal au maximum des coûts des instructions `p` et `q`, plus le temps d'évaluation de l'expression `b`.
— Le coût d'une boucle `for i in iterable : p` est égal au nombre d'éléments de l'itérable multiplié par le coût de l'instruction `p` si ce dernier ne dépend pas de la valeur de `i`. Quand le coût du corps de la boucle dépend de la valeur de `i`, le coût total de la boucle est la somme des coûts du corps de la boucle pour chaque valeur de `i`.
— Le cas des boucles `while` est plus complexe à traiter puisque le nombre de répétitions n'est en général pas connu *a priori*. On peut majorer le nombre de répétitions de la boucle de la même façon qu'on démontre sa terminaison et ainsi majorer le coût de l'exécution de la boucle.
— La complexité est notée dans ce cours sous la forme d'une relation de récurrence $C(n)$ où n est la taille de la donnée caractéristique de l'algorithme.

Exemple 1 (somme d'entiers)**.**

```
n = int(input('''n ? '''))
somme = 0
for i in range(n + 1) : somme += i
print(somme)
```

Exemple 2 (recherche dichotomique dans un tableau trié)**.**

Deux versions de cet algorithme sont proposées.

```
def cherche(x,t) :
    '''renvoie Vrai ou Faux selon que x est present ou non dans la liste t
t est non vide et trie en ordre croissant'''
    i, j = 0, len(t)
    while j - i > 1 :
        m = (i + j) // 2
        if t[m] <= x : i = m
        else : j = m
    return (t[i] == x)
```

```
def cherche(x,t) :
    '''renvoie Vrai ou Faux selon que x est present ou non dans la liste t
t est non vide et trie en ordre croissant'''
    if len(t) == 1 : return t[0] == x
    else :
        m = len(t) // 2
        if t[m] <= x : return cherche(x, t[m:])
        else : return cherche(x, t[:m])
```

Exemple 3 (Calcul du coefficient binomial $\binom{n}{k}$)**.**

```
def fact(n) :
    '''algorithme recursif qui renvoie n ! pour n entier naturel.'''
    if n <= 1 : return ( 1 ) # cas de base
    else : return ( n * fact( n - 1 ) )

# Calcul naif du coefficient binomial
def binom_naif(n, k) :
    '''renvoie le coefficient binomial (n k) = n! / (k! x (n - k)!)'''
    return ( fact(n) / (fact(k) * fact(n - k)) )

# Calcul efficace du coefficient binomial
def binom(n, k) :
    '''Calcul optimise du coefficient binomial (n k) selon le schema recursif
    suivant : (n k) = n/k . (n-1 k-1)'''
    if k == 0 :
        # cas de base
        return(1)
    else :
        return( (n / k) * binom(n - 1, k -1) )
```

Ce programme est utilisé dans le sujet d'oral de Centrale-Supélec 2015 de Math2 (sujet 27 de la filière PC). Le sujet amenait à calculer $\binom{2000}{1000}$.

L'utilisation de la version naïve implique deux dépassements :

— un dépassement de capacité :
`OverflowError: integer division result too large for a float`,
c'est le cas si le résultat de la division est reconnu comme $\pm$ `inf` par *Python*, soit 10e $\pm$ 308 ;

— un dépassement de la limite des appels récursifs :
`RecursionError:maximum recursion depth exceeded in comparison`.
Pour ce dépassement, on peut ajouter au programme les deux lignes suivantes vues au TD2.

```
import sys
sys.setrecursionlimit(10000)
```

En pratique, si l'on veut comparer les performances temporelles de deux algorithmes, il est possible d'utiliser le module `time`. Voici la proposition de fonction donnée dans le TD2.

```
import time

def compar_temp(L_fct, args, N) :
    '''compare le temps mis pour faire N appels des fonctions entrÃ©es dans
    la liste L_fct avec deux arguments entres dans un tuple.'''
    for fct in L_fct :
        debut = time.time()
        for k in range(N) :
            fct(args[0], args[1])
        fin = time.time()
        print('''duree pour''', N, '''appels de la fonction''', \
        str(fct).split()[1], ''':''', fin - debut)
```

Voici alors ce que renvoie le programme pour $\binom{2000}{200}$.

```
>>> compar_temp([binom_naif, binom], [2000, 200], 1000)
duree pour  1000 appels de la fonction  binom_naif :  4.936192512512207
duree pour  1000 appels de la fonction  binom :  0.07202363014221191
```

Exercice 1 :
Écrire pour tous ces exemples la relation de récurrence qui définit la complexité de l'algorithme en précisant bien la donnée caractéristique n utilisée et sa taille.

L'exemple 3 met en évidence une différence de complexité en mémoire tout autant qu'en temps.

1.2 Complexité et notation de *Landau* O

L'objectif n'est pas de déterminer le coût temporel exact d'un algorithme, car les différentes opérations arithmétiques ou de comparaison n'ont pas exactement toutes le même temps de calcul, et que les machines utilisées pour tester ces algorithmes n'ont pas forcément les mêmes caractéristiques. Mais ces considérations ne changent pas vraiment l'évaluation de l'efficacité de ces algorithmes dont on veut connaître un ordre de grandeur du temps d'exécution.

On utilisera donc pour caractériser la complexité d'un algorithme la notation mathématique $O(n)$. On dira qu'un algorithme a une *complexité* en $O(n)$ si son coût est, à partir d'un certain rang, inférieur au produit de $f(n)$ par une constante.

	Nom courant	Temps pour $n = 10^6$
$O(1)$	temps constant	1 ns
$O(\log n)$	logarithmique	10 ns
$O(n)$	linéaire	1 ms
$O(n \cdot \log n)$	quasi-linéaire	1 ms
$O(n^2)$	quadratique	1/4 h
$O(n^k)$	polynomiale	30 ans, si $k = 3$
$O(2^n)$	exponentielle	plus de 10^{300000} milliards d'années

Remarque 1. *Commentaires sur les différents temps d'exécution.*

1. $O(1)$ *: Le temps d'exécution ne dépend pas des données traitées, ce qui est assez rare !*
2. $O(\log n)$ *: En pratique, cela correspond à une exécution quasi instantanée. Bien souvent, à cause du codage binaire de l'information, c'est en fait la fonction* $\log_2 n$ *qu'on voit apparaître ; mais comme la complexité est définie à un facteur près, la base du logarithme n'a pas d'importance.*
3. $O(n)$ *: Le temps d'exécution d'un tel algorithme ne devient supérieur à une minute que pour des données de taille comparable à celle des mémoires vives disponibles actuellement. Le problème de la gestion de la mémoire se posera donc avant celui de l'efficacité en temps.*
4. $O(n \cdot \log n)$ *: Comme son nom l'indique, le temps d'exécution est très proche de celui d'un algorithme de complexité linéaire.*
5. $O(n^2)$ *: Cette complexité reste acceptable pour des données de taille moyenne* ($n < 10^6$)*, mais pas au delà.*
6. $O(n^k)$ *: Ici,* n^k *est le terme de plus haut degré d'un polynôme en* n
7. $O(2^n)$ *: Un algorithme d'une telle complexité est impraticable sauf pour de très petites données* ($n < 50$).

1.3 Complexité d'une fonction récursive

On reprends l'exercice 6 du Cours 2.

$$\begin{cases} u_0 = 2 \\ u_n = \frac{1}{2} \cdot \left(u_{n-1} + \frac{3}{u_{n-1}} \right) \end{cases}$$

```
def u(n) :
    '''renvoie le nieme terme de la suite u_n pour n entier naturel'''
    if n == 0 : return (2)  # cas de base
    else : return ( 0.5 * ( u(n - 1) + 3 / u(n - 1) ) )
```

Si n désigne la valeur de l'argument de cette fonction, on note $C(n)$ le nombre d'opérations arithmétiques qu'elle effectue. Alors, les équations définissant $C(n)$ sont les suivantes :

$$C(0) = 0$$

$$C(n) = C(n-1) + C(n-1) + 3$$

Car le calcul de u_n se fait grâce à deux appels récursifs de u_{n-1} et à trois opérations arithmétiques (multiplication, addition, division).

Il s'agit donc d'une suite arithmético-géométrique $C(n) = 2 \cdot C(n-1) + 3$ dont le terme général est :

$$C(n) = 3 \cdot (2^n - 1)$$

La complexité de cet algorithme est donc un $O(2^n)$, ce qui est exécrable on l'a vu !

Une solution simple pour améliorer cette complexité est de s'arranger pour ne faire qu'un seul appel de la fonction récursive.

```
def u(n) :
    '''renvoie le nieme terme de la suite u_n pour n entier naturel'''
    if n == 0 : return (2)  # cas de base
    else :
        v = u(n - 1)
        return ( 0.5 * ( v + 3 / v ) )
```

Exercice 2 :
Évaluer la complexité de ce nouvel algorithme.

1.4 Quelques nuances sur la complexité

Lors de l'évaluation de la complexité d'un algorithme, on dénombre le plus souvent un seul type d'opération (en général la plus coûteuse). Il est bon alors de préciser *complexité en nombre de multiplications*, opération usuelle pour les algorithmes arithmétiques, ou *complexité en nombre de comparaisons*, opération majeure des algorithmes de recherche et de tris.

Enfin, lorsque l'analyse de la complexité de l'algorithme s'y prêtera, trois types de complexités peuvent être envisagés :

— la *complexité dans le pire des cas*, C_{max} ; c'est un majorant du temps d'exécution sur toutes les entrées possibles d'une même taille ; la complexité dans le pire cas apporte une notion de sécurité sur le temps d'exécution ;
— la *complexité en moyenne*, C_{moy} ; il s'agit d'une moyenne sur le temps d'exécution sur toutes les entrées possibles d'une même taille en considérant une distribution équiprobable ; elle représente le comportement moyen d'un algorithme ;
— la *complexité dans le meilleur des cas*, C_{min} ; elle est peu utile, facile à calculer, mais n'apporte qu'une borne inférieure sur les temps d'exécution.

2 Algorithmes de tri

Dans toute cette partie, on ne va s'intéresser qu'à des listes d'entiers à trier. Mais les algorithmes que l'on va mettre en évidence seront valables pour n'importe quel type d'éléments, pourvu qu'il soit muni d'un *ordre total*, c'est à dire tel que $\forall (x,y) \in E^2, x \mathscr{R} y$ ou $y \mathscr{R} x$. On suppose qu'on trie des tableaux, dans l'ordre croissant et on note N, le nombre d'éléments à trier.

2.1 Tri par insertion

Le tri par insertion est un algorithme naïf qui consiste à insérer successivement chaque élément dans l'ensemble des éléments déjà triés. C'est souvent ce que l'on fait quand on trie un jeu de cartes par exemple.

Le tri par insertion d'un tableau `t` s'effectue en place, c'est-à-dire qu'il ne demande pas d'autre tableau que celui que l'on trie. Son coût en mémoire est donc constant si on ne compte pas la place occupée par les données. Il consiste à insérer successivement chaque élément `t[i]` dans la portion du tableau `t[0:i]` déjà triée.

Exemple 4. *Trier le tableau suivant en détaillant chaque ligne, résultat de cet algorithme.*

$$[0,7,5,2,4,9,8,3,6,1]$$

Voici une écriture de cet algorithme en Python.

```
def tri_ins(t) :
    '''trie la liste t par la methode d'insertion'''
    tt = t[:]
    for k in range(1, len(tt)) :
        temp = tt[k]
        j = k
        while j > 0 and temp < tt[j - 1] :
            tt[j] = tt[j - 1]
            j -= 1
        tt[j] = temp
    return (tt)
```

Exercice 3 :
Étude du tri par insertion

1. Montrer que la boucle `while` se termine.
2. Déterminer un invariant de boucle pour l'algorithme.

3. Déterminer la complexité au pire, au mieux puis moyenne, préciser leurs noms courants.
4. Proposer un exemple de tableau sur lequel le tri par insertion a un coût linéaire (meilleur cas). Proposer également un exemple de tableau sur lequel il a un coût quadratique (pire cas).
5. Proposer une version récursive de cet algorithme.

2.2 Tri par fusion

Le tri fusion consiste à appliquer la méthode *diviser pour régner* ou *divide and conquer*. L'idée récursive est la suivante :

— s'il y a au plus une valeur, le tableau est trié ;
— s'il y a au moins deux valeurs, couper le tableau en deux, trier (récursivement) les deux sous-tableaux obtenus, puis fusionner les résultats ; la fusion consiste à rassembler dans un seul tableau trié les valeurs contenues dans les deux tableaux triés fournis en paramètre (en respectant bien sûr les répétitions éventuelles : la longueur du résultat de la fusion est la somme des longueurs des deux tableaux initiaux).

Voici un programme Python, en supposant programmée la fonction `fusion` :

```
def tri_fusion(t) :
    '''renvoie la liste t triee par la methode du tri fusion'''
    if len(t) < 2 :
        return (t)
    else :
        m = len(t) // 2
        return (fusion(tri_fusion(t[:m]), tri_fusion(t[m:])))
```

Exercice 4 :
Étude du tri fusion

1. Trier le tableau de l'exemple du tri par insertion en détaillant chaque ligne.
2. Programmer la fonction `fusion`.
3. Montrer que votre algorithme se termine et est correct.
4. Analyser la complexité de cet algorithme.

2.3 Tri rapide

Le tri rapide consiste là encore à *diviser pour régner*. On partitionne d'abord le tableau `t` à trier autour d'un *pivot* : on choisit l'une des valeurs du tableau (ledit pivot), par exemple `t[0]` et l'on construit deux tableaux avec les `t[i]` pour `i > 0` (où l'on peut retrouver le pivot si sa valeur figure pour plusieurs indices...). :

— le premier `t1` avec les valeurs correspondant aux indices `i` tels que `t[i] < pivot` ;
— le second `t2` avec les valeurs correspondant aux indices `i` tels que `t[i] >= pivot`.

Il n'y a plus qu'à trier (récursivement bien sûr) `t1` et `t2` et à renvoyer les valeurs triées de `t1`, suivies de la valeur du pivot et des valeurs triées de `t2`. On obtient ainsi les valeurs de `t` triées. La preuve immédiate ne peut être donnée que par récurrence forte.

Exercice 5 :
Étude du tri rapide

1. Implanter cet algorithme en Python.
2. Montrer que votre algorithme se termine et est correct.
3. Déterminer la complexité de votre algorithme.

3 Synthèse sur l'évaluation de la complexité d'un algorithme

L'objectif de cette synthèse est de donner des exemples de démonstration pour déterminer de manière rigoureuse la complexité d'algorithmes.

Notations

- n désigne la taille de l'argument en entrée de l'algorithme.
- $C(n)$ traduit le nombre d'opérations arithmétiques qu'elle effectue.
- $C(n)$ peut être un coût en nombre de multiplications pour les algorithmes de calcul ou en nombre de comparaisons pour les algorithmes de recherche et de tri.

On cherche à exprimer $C(n)$ en fonction de constantes de n et des valeurs antérieures $C(k)$ avec $k < n$. Les constantes sont déterminées par les instructions simples alors que n et $C(k)_{k<n}$ apparaissent dans les boucles et les appels récursifs.

Cas 1. *C'est le cas tiré de l'exemple 1 du cours.*

```
n = int(input('''n ? '''))
somme = 0
for i in range(n + 1) : somme += i
print(somme)
```

Dans la boucle `for`, on répète a instructions n fois (ici, $a = 1$).

$$C(n) = n \times a = a \times n$$

C'est une **complexité linéaire** : $C(n) = O(n)$.

Cas 2. *C'est le cas tiré des exemples 2 & 3 du cours.*

```
def cherche(x,t) :
    '''renvoie Vrai ou Faux selon que x est present ou non dans la liste t
t est non vide et trie en ordre croissant'''
    i, j = 0, len(t)
    while j - i > 1 :
        m = (i + j) // 2
        if t[m] <= x : i = m
        else : j = m
    return (t[i] == x)

def cherche(x,t) :
    '''renvoie Vrai ou Faux selon que x est present ou non dans la liste t
t est non vide et trie en ordre croissant'''
    if len(t) == 1 : return t[0] == x
    else :
        m = len(t) // 2
        if t[m] <= x : return cherche(x, t[m:])
        else : return cherche(x, t[:m])
```

Dans cet exemple, le nombre d'opérations au rang n compte un certain nombre d'instructions et dépend du nombre d'opérations au rang $n/2$.

$$C(n) = a + C(\lfloor n/2 \rfloor)$$

On suppose que $\exists p \in \mathbb{N}^*,\ 2^p < n \leqslant 2^{p-1}$.
De plus $C(n)$ est une fonction croissante de n. En effet, $C(1) = 3$ et $C(2) = 4+3 = 7 \geqslant C(1)$, et soient 3 entiers successifs $2\cdot k-1, 2\cdot k, 2\cdot k+1$, avec $k \in \mathbb{N}^*$. $C(2\cdot k) = a + C(k)$, et $C(2\cdot k+1) = a + C(k)$, donc $C(2\cdot k+1) \geqslant C(2\cdot k)$. $C(2\cdot k-1) = a+C(k-1)$, et $C(2\cdot k) = a+C(k)$, donc $C(2\cdot k)-C(2\cdot k-1) = C(k)-C(k-1)$, donc par récurrence forte, $C(2\cdot k) \geqslant C(2\cdot k-1)$.
La fonction $C(n)$ est donc croissante.
Nous cherchons alors $C(2^p) = a + C(2^{p-1})$.
On peut donc construire une suite (U_p) telle que $U(p) = C(2^p)$. Alors $U(p) = a + U(p-1)$, la suite (U_p) est une suite arithmétique et $U(p) = a \times p + U(0)$, avec $U(0) = C(1)$.
Donc $C(2^p) = a \times p + C(1)$, c'est à dire $C(n) = a \times \log_2(n) + b$.

C'est une **complexité logarithmique** : $C(n) = O(\log n)$.

Cas 3. *C'est le cas tiré de l'exercice 6 du Cours 2 en version naïve.*

```
def u(n) :
    '''renvoie le nieme terme de la suite u_n pour n entier naturel'''
    if n == 0 : return (2)  # cas de base
    else : return ( 0.5 * ( u(n - 1) + 3 / u(n - 1) ) )
```

La fonction renvoie une expression composée de 3 opérations arithmétiques (2 multiplications et 1 addition) et de deux appels récursif de la fonction au rang $n-1$. La complexité en nombre de multiplications est donc :

$$C(n) = 2 + 2 \cdot C(n-1) = a \cdot C(n-1) + b$$

On est en présence d'une suite arithmético-géométrique $C(n) = f(C(n-1))$ avec $f : x \mapsto a \cdot x + b$.
Alors on construit la nouvelle suite (V_n) telle que $V(n) = C(n) - r$ avec r tel que $f(r) = r = a \cdot r + b$.
$V(n+1) = C(n+1) - r = a \cdot C(n) + b - r = a \cdot \big(C(n) - r\big) + \underbrace{a \cdot r + b - r}_{r-r}$
Donc $V(n+1) = a \cdot V(n)$, (V_n) est une suite géométrique de raison a, d'où $V(n) = a^n \cdot V(0)$ avec $V(0) = C(0) - r$.
Ici $C(0) = 0$, on peut donc en déduire que $C(n) = V(n) + r = a^n \cdot (-r) + r = r \cdot (1 - a^n)$
Dans notre exemple, $a = 2$, $b = 2$, donc $r = -2$ et $C(n) = 2 \cdot (2^n - 1)$.

C'est une **complexité exponentielle** : $C(n) = O(2^n)$.

Cas 4. *C'est le cas tiré de l'exercice 6 du Cours 2 dans sa deuxième version.*

```
def u(n) :
    '''renvoie le nieme terme de la suite u_n pour n entier naturel'''
    if n == 0 : return (2)  # cas de base
    else :
        v = u(n - 1)
        return ( 0.5 * ( v + 3 / v ) )
```

Avec cette nouvelle écriture de l'algorithme, on a 1 appel récursif `v = u(n - 1)` de rang $n-1$ et 2 multiplications dans l'expression renvoyée par la fonction.

$$C(n) = 2 + C(n-1) = a + C(n-1)$$

(C_n) est une simple suite arithmétique de raison a, donc $C(n) = a \times n + C(0)$.

C'est une **complexité linéaire** : $C(n) = O(n)$.

Cas 5. *C'est le cas du tri par insertion du cours 3.*

```
def tri_ins(t) :
    '''trie la liste t par la methode d'insertion'''
    tt = t[:]
    for k in range(1, len(tt)) :
        temp = tt[k]
        j = k
        while j > 0 and temp < tt[j - 1] :
            tt[j] = tt[j - 1]
            j -= 1
        tt[j] = temp
    return (tt)
```

Dans la boucle `for`, il y a 2 affectations puis une boucle `while`, puis une dernière affectation.
Le test de comparaison de la boucle `while` et les deux affectations qui s'y trouvent, sont réalisées selon les cas. Dans le meilleur des cas, la comparaison n'a lieu qu'une fois, et dans le pire des cas, k fois. On exprime la complexité en nombre de comparaisons (donc on néglige le coût des affectations).

$$C(n) = \sum_{k=1}^{n-1} 1 \text{ ou } k$$

Au pire des cas $C(n) = \sum_{k=1}^{n-1} k$, c'est la somme des $n-1$ premiers termes d'une suite arithmétique de raison 1. Donc $C(n) = \frac{n \cdot (n-1)}{2}$.

C'est une **complexité quadratique** : $C(n) = O(n^2)$.

Au meilleur des cas $C(n) = \sum_{k=1}^{n-1} 1 = n - 1$.

C'est une **complexité linéaire** : $C(n) = O(n)$.

Cas 6. *C'est le cas du tri fusion du cours 3.*

```
def tri_fusion(t) :
    '''renvoie la liste t triee par la methode du tri fusion'''
    if len(t) < 2 :
        return (t)
    else :
        m = len(t) // 2
        return (fusion(tri_fusion(t[:m]), tri_fusion(t[m:])))
```

Dans cette fonction récursive, on trouve une comparaison, une opération arithmétique et une affectation puis l'appel d'une nouvelle fonction `fusion` avec comme arguments, deux appels récursifs de la fonction `tri_fusion` au rang $n/2$. On exprime la complexité en nombre de multiplications (donc on néglige le coût des comparaisons et des affectations). On note $T(n)$ la complexité de la fonction `tri_fusion` et $F(n)$, celle de la fonction `fusion`.

$$T(n) = 1 + 2 \cdot T(\lfloor n/2 \rfloor) + F(n)$$

La fonction `fusion` va opérer quand les appels récursifs de `tri_fusion` vont arriver à leurs cas de base, donc pour des tableaux de taille 1. On aura $\frac{n}{2}$ fusions au premier rang pour obtenir des tableaux de taille 2, puis $\frac{\frac{n}{2}}{2}$ fusions au deuxième rang pour obtenir des tableaux de taille 4, etc.

$$F(n) = \frac{n}{2} + \frac{n}{4} + \cdots + \frac{n}{2^p} = n \cdot \sum_{k=1}^{p} \frac{1}{2^k}$$

Ici aussi, on peut supposer que $\exists p \in \mathbb{N},\ n = 2^p$.
La série $\sum_{k=1}^{p} \frac{1}{2^k}$ converge vers 1, donc $F(n) = O(n)$.

Il reste à déterminer $T(n)$ à partir de la relation de récurrence $T(n) = n+1+2 \cdot T(\lfloor n/2 \rfloor)$ ou $T(2^p) = 2^p+1+2 \cdot T(2^{p-1})$.
Cette expression peut se simplifier en considérant que $2^p >> 1$, donc $T(2^p) = 2^p + 2 \cdot T(2^{p-1})$.
On construit ici encore une nouvelle suite (V_p) telle que $V(p) = 2^{-p} \cdot T(2^p)$.
Alors $V(p+1) = 2^{-(p+1)} \cdot T(2^{p+1}) = 2^{-(p+1)} \cdot \left(2^{p+1} + 2 \cdot T(2^p)\right) = 1 + 2^{-p} \cdot T(2^p) = 1 + V(p)$.
La suite (V_p) est donc une simple suite arithmétique de raison 1, donc $V(p) = p + V(0) = p + T(1)$ avec $T(1) = 0$ (en nombre de multiplications).
On peut en déduire que $T(2^p) = p \cdot 2^p$ ou $V(n) = n \cdot \log_2(n)$.

C'est une **complexité quasi-linéaire** : $C(n) = O(n \cdot \log n)$.

C'est aussi le cas du tri rapide du cours 3. On peut montrer en suivant le même raisonnement que pour le tri fusion, que la complexité du tri rapide est quasi-linéaire : $C(n) = O(n \cdot \log n)$.

Tableau récapitulatif

$O(\log n)$	algorithmes basés sur la dichotomie, *Exemples 2 & 3 du cours 3*
$O(n)$	boucle `for` comportant un nombre fini d'instructions répétée n fois, *Exemple 1 du cours 3* fonction récursive comportant un nombre fini d'instructions et un appel récursif, *2e version de l'exercice 6 du Cours 2* tri par insertion au meilleur des cas
$O(n \cdot \log n)$	tri fusion et tri rapide
$O(n^2)$	tri par insertion au pire des cas, la complexité est du même ordre pour les tris par sélection et les tris à bulles.
$O(2^n)$	fonction récursive comportant deux appels récursifs, *1re version de l'exercice 6 du Cours 2*