COMPLEXITÉ

À la fin de ce chapitre, je sais :

- ☞ définir les concepts de complexité temporelle et complexité mémoire
- ☞ calculer la complexité d'algorithmes simples
- ☞ calculer la complexité d'algorithmes récursifs

A Complexités algorithmiques

Lorsque la taille des données d'entrée à traiter d'un algorithme augmente, le résultat est que l'algorithme met généralement plus de temps à s'exécuter. Pourquoi est-ce un problème ? C'est un problème car, dans les activités humaines, lorsqu'un système fonctionne, on a souvent tendance à lui en demander plus, très vite. Or, si le système développé est capable de gérer trois utilisateurs, peut-il en gérer un million en un temps raisonnable et sans s'effondrer ? C'est peu probable.

La question est donc de savoir :

- comment mesurer la sensibilité d'un algorithme au changement d'échelle des données d'entrée,
- comment mesurer cette sensibilité indépendamment des machines concrètes (processeurs), car on se doute bien que selon la puissance de la machine, le résultat ne sera pas le même.

■ **Définition 1 — Complexité temporelle.** La complexité temporelle est une mesure de l'évolution du temps nécessaire à un algorithme pour s'exécuter correctement en fonction de la taille des données d'entrée. La complexité temporelle est directement liée au nombre d'instructions à exécuter.

■ **Définition 2 — Complexité mémoire ou spatiale.** La complexité mémoire est une mesure de l'évolution de l'espace nécessaire à un algorithme pour s'exécuter correctement en fonction de la taille des données d'entrée. La complexité mémoire est associée à la taille de l'espace mémoire occupé par un algorithme au cours de son exécution.

D'un point de vue opérationnel, si la taille n des données d'entrées augmente, un bon algorithme doit pouvoir délivrer des résultats en un temps fini, même si le nombre d'instructions

lié aux boucles ou aux appels récursifs dépend de n. C'est pourquoi la complexité est un **calcul asymptotique** : on s'intéresse au comportement de l'algorithme lorsque n tend vers l'infini.

B Notation asymptotique

On utilise la notation de Landau O pour la qualifier le comportement asymptotique de la complexité [1]. Le tableau 1 recense les principales complexités et donne un exemple associé.

■ **Définition 3 — Notation de Landau** O**.** Soit $f : \mathbb{N} \to \mathbb{R}_+$ et $g : \mathbb{N} \to \mathbb{R}_+$. On dit que f ne croit pas plus vite que g et on note $f = O(g)$ si et seulement si :

$$\exists C \in \mathbb{N}, \exists n_i \in \mathbb{N}, \forall n \in \mathbb{N}, n \geqslant n_i \Rightarrow f(n) \leqslant Cg(n)$$

R Cette définition signifie simplement qu'au bout d'un certain rang, la fonction f ne croit jamais plus vite que la fonction g.

Théorème 1 — Propriétés de O**.** Soit f, f_1, f_2, g, g_1 et $g_2 : \mathbb{N} \to \mathbb{R}_+$.

1. $\forall k \in \mathbb{N}, O(k.f) = O(f)$
2. $f = O(g) \Rightarrow f + g \in O(g)$
3. $f_1 = O(g_1), f_2 = O(g_2) \Rightarrow f_1 + f_2 = O(max(g_1 + g_2))$
4. $f_1 = O(g_1), f_2 = O(g_2) \Rightarrow f_1 f_2 = O(g_1 g_2)$
5. $\forall k \in \mathbb{N}, f = O(g) \Rightarrow k.f = O(g)$

■ **Exemple 1 — Simplification de notations asymptotiques.** Supposons qu'on ait compté le nombre d'opérations d'un algorithme en fonction de n et qu'on ait trouvé : $2n^2 + 4n + 3$. Alors la complexité de l'algorithme est $O(n^2)$. En effet, on a bien $\forall n \in \mathbb{N}, 2n^2 + 4n + 3 < 10n^2$. Si vous avez un doute, étudiez le signe du trinôme $-8n^2 + 4n + 3$.

De même, $10\log(n) + 5(\log(n))^3 + 7n + 3n^2 + 6n^3 = O(n^3)$. Pour le montrer, il suffit d'utiliser le théorème sur les croissances comparées.

1. $O(n)$ se dit «grand o de n».

Complexité	**Nom**	**Description**
$O(1)$	Constante	Instructions exécutées un nombre constant de fois **Indépendante de la taille de l'entrée**
$O(\log(n))$	Logarithmique	Légèrement plus lent lorsque n augmente.
$O(n)$	Linéaire	L'algorithme effectue une tâche constante pour chaque élément de l'entrée.
$O(n\log(n))$	Linéarithmique	L'algorithme effectue une tâche logarithmique pour chaque élément de l'entrée.
$O(n^2)$	Quadratique	L'algorithme effectue une tâche linéaire pour chaque élément de l'entrée.
$O(n^k)$	Polynomiale	Typiquement k tâches linéaires imbriquées.
$O(k^n)$	Exponentielle	L'algorithme effectue une tâche constante sur tous les sous-ensembles de l'entrée.
$O(n!)$	Factorielle	L'algorithme effectue une tâche dont la complexité est multipliée par une quantité croissante proportionnelle à n.

TABLE 1 – Hiérarchie des complexités temporelles de la moins complexe à la plus complexe.

Taille de l'entrée	$O(\log n)$	$O(n)$	$O(n\log n)$	$O(n^2)$	$O(n^3)$	$O(2^n)$
10^2	2,3 ns	50 ns	230 ns	5 μs	500 μs	335 années
10^3	3,4 ns	500 ns	3,45μs	500 μs	500 ms	10^{282} années
10^4	4,6 ns	5 μs	46 μs	50 ms	500 s	...
10^5	5,7 ns	50 μs	575 μs	5 s	2h20 min	...
10^6	6,9 ns	500 μs	6,9 ms	500 s	96 jours	...
10^9	10 ns	500 ms	10,4 s	96 jours	...	...

TABLE 2 – Sur une machine cadencée à 2 Ghz, quelle est la durée prévisible d'exécution d'un algorithme en fonction de la taille des données d'entrée et de sa complexité ? On suppose qu'une seule période d'horloge est nécessaire au traitement d'une donnée.

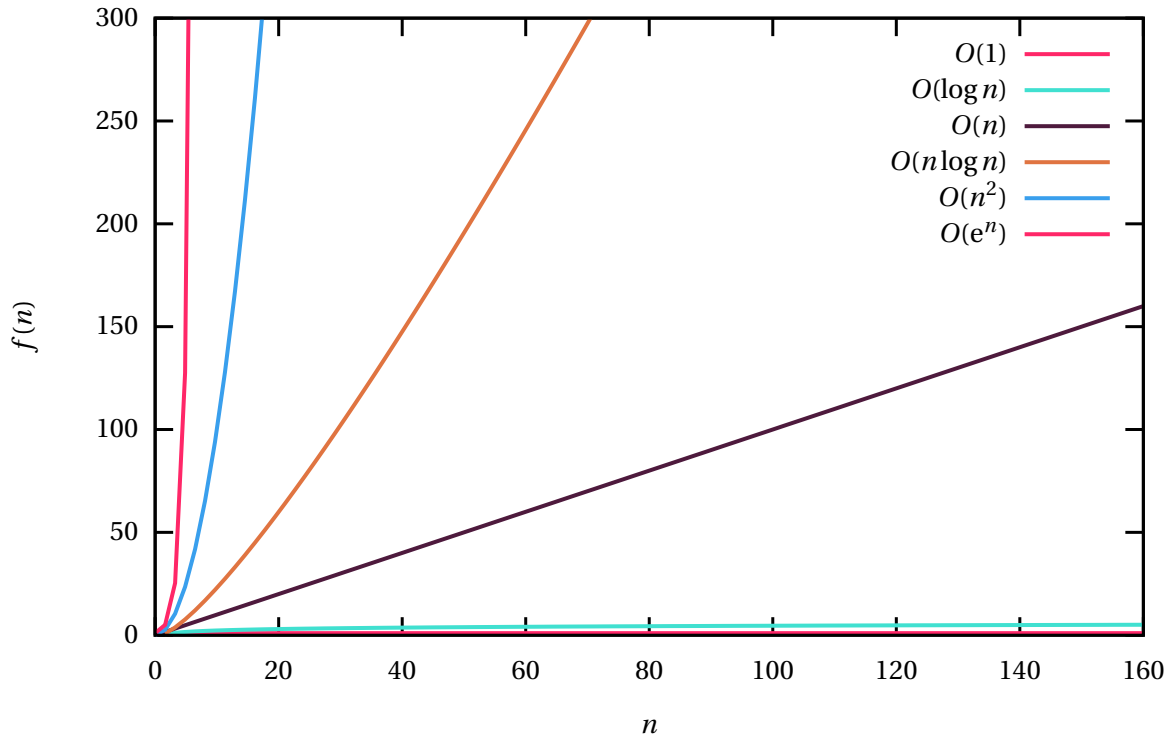

FIGURE 1 – Comparaison des croissances des complexités usuelles

C Typologie de la complexité

Selon l'algorithme étudié, on est amené à s'intéresser à différentes complexités :

- La complexité dans le pire des cas, c'est à dire l'estimation du nombre d'instructions nécessaires dans le cas le plus défavorable.
- La complexité dans le meilleur des cas, idem dans le cas le plus favorable.
- La complexité moyenne, c'est à dire une moyenne de la complexité de tous les cas possibles.

R Ces trois calculs de complexité sont parfois nécessaires pour faire un choix d'algorithme et la connaissance statistique de la nature des données d'entrée peut influer sur le ce choix.

D Calcul du coût d'une instruction

Il est difficile de savoir exactement en combien de temps une instruction d'un programme s'exécute pour plusieurs raisons :

1. les compilateurs disposent de fonctions d'optimisation en langage machine ou en code interprétable qui font que le code source n'est pas nécessairement représentatif du code exécuté. Il serait donc nécessaire d'examiner le code exécutable pour statuer.
2. selon les architectures électroniques et les machines virtuelles, le coût d'une même opération varie.

Cependant, dans le cadre d'un calcul de complexité d'un algorithme, on peut s'abstraire de ces considérations électroniciennes et considérer qu'une opération élémentaire i possède un coût constant qu'on notera c.

Dans ce qui suit, on suppose qu'on dispose d'une machine pour tester l'algorithme. On fait l'hypothèse réaliste que les opérations élémentaires suivantes sont réalisées en un temps constant c par cette machine :

- opération arithmétique $+, -, *, /, //, \%$,
- tests $==, !=, <, >$,
- affectation $\leftarrow$,
- accès à un élément indicé $t[i]$,
- structures de contrôles (structures conditionnelle et boucles), coût associé négligé,
- échange de deux éléments dans le tableau,
- accès à la longueur d'un tableau.

Finalement, on fait l'approximation supplémentaire qu'une combinaison simple de ces opérations est également réalisée en un temps constant c.

E Calculs classiques de complexité

■ **Exemple 2 — Calcul d'une complexité linéaire.** On souhaite calculer la complexité de l'algorithme 1. La taille du problème dépend de n, c'est à dire la puissance à laquelle on veut calculer le nombre a. En effet, pour différents a, plus petits ou plus grands, l'exécution ne sera pas plus chronophage. Le coût total $C(n)$ associé à cet algorithme peut donc s'écrire :

$$C(n) = c + n \times c = O(nc) = cO(n) = O(n) \tag{1}$$

Comme un coût c constant est $O(1)$, car constant en fonction de n, la complexité de l'algorithme 1 est donc linéaire.

Algorithme 1 Calcul de a^n

```
1 : Fonction PUISSANCE(a, n)                    ▷ a et n sont des entiers naturels
2 :     p ← 1                                   ▷ coût : c
3 :     pour i = 1, ..., n répéter              ▷ on répète n fois
4 :         p ← p × a                           ▷ coût : c
5 :     renvoyer p
```

■ **Exemple 3 — Calcul d'une complexité quadratique.** On souhaite calculer la complexité de l'algorithme 2. Le coût total associé à cet algorithme peut donc s'écrire :

$$C(n) = c + \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} c = c + c\sum_{i=0}^{n-1}\sum_{j=0}^{n-1} 1 = c + n^2 c = O(n^2) \tag{2}$$

C'est pourquoi, la complexité de l'algorithme 2 est en $O(n^2)$.

Algorithme 2 Produit de deux vecteurs $(n,1) \times (1,n) \longrightarrow (n,n)$

```
1 : Fonction PVEC(u, v)                         ▷ u est (n,1) et v est (1,n)
2 :     t ← nouveau tableau de taille (n,n)     ▷ coût : c
3 :     pour i de 0 à n − 1 répéter             ▷ on répète n fois
4 :         pour j de 0 à n − 1 répéter         ▷ on répète n fois
5 :             t[i, j] ← u[i] × v[j]           ▷ coût : c
6 :     renvoyer t                              ▷ le résultat
```

■ **Exemple 4 — Calcul d'une complexité quadratique plus subtile.** On souhaite calculer la complexité de l'algorithme 3 qui fait lui-même appel à un autre algorithme qui calcule une puissance en une complexité linéaire [a] $c_p n$.

Le coût total associé à cet algorithme 3 peut s'écrire :

$$C(n) = c + n \times (c + c_p n) = c + cn + c_p n^2 = O(n^2) \tag{3}$$

C'est pourquoi, la complexité de l'algorithme 3 est en $O(n^2)$. C'est pourquoi, il faut veiller à bien étudier tous les coûts, directs et indirects, afin de ne pas conclure hâtivement parce qu'il n'y a qu'une seule boucle que la complexité est linéaire…

a. Oui, on peut faire mieux!

Algorithme 3 Somme de puissances $1 + 2^n + \ldots + n^n$

```
1 : Fonction SOMME_PUISSANCE(n)                    ▷ n est un entier naturel
2 :     acc ← 0                                    ▷ coût : c
3 :     pour k de 1 à n répéter                    ▷ on répète n fois
4 :         acc ← acc + PUISSANCE(k, n)            ▷ coût : c + c_p n
5 :     renvoyer acc                               ▷ le résultat
```

■ **Exemple 5 — Complexité quadratique.** On souhaite calculer la complexité de l'algorithme 4. On peut calculer le coût total de l'algorithme 4 comme suit :

$$C(n) = c + \sum_{k=1}^{n} \sum_{i=1}^{k} c \tag{4}$$

$$= c + c \sum_{k=1}^{n} \sum_{i=1}^{k} 1 \tag{5}$$

$$= c + c \sum_{k=1}^{n} k \tag{6}$$

$$= c + c \frac{n(n+1)}{2} \tag{7}$$

$$= O(n^2) \tag{8}$$

Algorithme 4 Accumuler

```
1 : Fonction QACC(n)
2 :     a ← 0                                      ▷ coût : c
3 :     pour k de 1 à n répéter                    ▷ on répète n fois
4 :         pour i de 1 à k répéter                ▷ on répète k fois
5 :             a ← a + i                          ▷ coût : c
6 :     renvoyer a                                 ▷ le résultat
```

■ **Exemple 6 — Complexité polynomiale.** On souhaite calculer la complexité de l'algorithme 5. On suppose qu'on connaît la complexité de f et qu'elle est linéaire. On peut donc calculer

le coût total de l'algorithme 5 comme suit :

$$c = c + \sum_{k=1}^{n}\sum_{i=1}^{k} c + c_f i = c + c\sum_{k=1}^{n}\sum_{i=1}^{k} 1 + c_f \sum_{k=1}^{n}\sum_{i=1}^{k} i \tag{9}$$

$$= c + c\frac{n(n+1)}{2} + \sum_{k=1}^{n} \frac{k(k+1)}{2} = c + c\frac{n(n+1)}{2} + \sum_{k=1}^{n} \frac{k}{2} + \frac{k^2}{2} \tag{10}$$

$$= c + c\frac{n(n+1)}{2} + c_f \frac{n(n+1)}{4} + c_f \frac{n(n+1)(2n+1)}{12} \tag{11}$$

$$= O(n^3) \tag{12}$$

Algorithme 5 Appliquer une fonction et accumuler

```
1 : Fonction FACC(n)                          ▷ Applique f et accumule n fois
2 :     a ← 0                                 ▷ coût : c
3 :     pour k de 1 à n répéter               ▷ on répète n fois
4 :         pour i de 1 à k répéter           ▷ on répète k fois
5 :             a ← a + f(i)                  ▷ coût : c + c_f i
6 :     return c                              ▷ le résultat
```

D'autres exemples de calcul de complexité sont abordés dans ce cours, notamment au chapitre **??** et au chapitre **??**.

F Exemple de la recherche dichotomique

■ **Exemple 7 — Recherche dichotomique.** L'algorithme de recherche dichotomique 6 est un exemple d'algorithme de type diviser pour régner : la division du problème en sous-problèmes est opérée via la ligne 5. La résolution des sous-problèmes est effectuée par des appels récursifs. La combinaison des résultats n'est pas explicite mais s'effectue sur le tableau lui-même grâce aux indices g et d.

R La recherche dichotomique est donc bien un cas particulier d'algorithme diviser pour régner avec $r = 1$ et $d = 2$, c'est à dire un seul appel récursif par chemin d'exécution et une division de la taille du problème par deux.

Supposons que le tableau d'entrée de l'algorithme possède n éléments et que le nombre d'opérations nécessaires à l'algorithme est $T(n)$. Pour simplifier le calcul, on fait l'hypothèse que n est une puissance de deux. On peut expliciter formellement la relation de récurrence qui existe entre $T(n)$ et $T(n/2)$: on a $T(n) = T(n/2) + c$, car en dehors de l'appel récursif, le coût de l'exécution vaut c. Les différents appels récursifs sont illustrés sur la figure 2.

Algorithme 6 Recherche récursive d'un élément par dichotomie dans un tableau trié

```
1 : Fonction REC_DICH(t, g, d, elem)
2 :     si g > d alors                                   ▷ Condition d'arrêt
3 :         renvoyer l'élément n'a pas été trouvé
4 :     sinon
5 :         m ← (g+d)//2                                ▷ Diviser
6 :         si t[m] = elem alors
7 :             renvoyer m
8 :         sinon si elem < t[m] alors
9 :             renvoyer REC_DICH(t, g, m-1, elem)      ▷ résoudre
10 :        sinon
11 :            renvoyer REC_DICH(t, m+1, d, elem)      ▷ résoudre
```

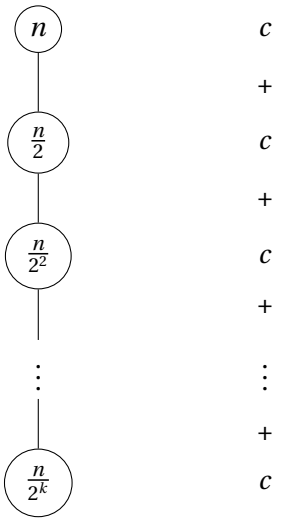

FIGURE 2 – Structure d'arbre et appels récursifs pour la récurrence de la recherche dichomotique : $T(n) = T(n/2) + c$ et $\frac{n}{2^k} = 1$. Hors appel récursif, la fonction opère un nombre constant d'opérations c.

On peut donc écrire :

$$T(n) = T(n/2) + c \tag{13}$$
$$= T(n/4) + c + c = T(n/4) + 2c \tag{14}$$
$$= T(n/8) + 3c \tag{15}$$
$$= \dots \tag{16}$$
$$= T(n/2^k) + kc \tag{17}$$
$$= T(1) + kc \tag{18}$$

D'après l'algorithme 6, la condition d'arrêt s'effectue en un nombre constant d'opérations : $T(1) = O(1)$. Donc on a $T(n) = O(k)$. Or, on a $\frac{n}{2^k} = 1$. Donc $k = \log_2 n$ et $T(n) = O(\log n)$.

On peut également le montrer plus mathématiquement en considérant $k = \log_2 n$ et la suite $(u_k)_{k\in\mathbb{N}^\star}$ telle que $u_k = u_{k-1} + c$ et $u_1 = c$. C'est une suite arithmétique, $u_k = kc$. D'où le résultat.

G Exemple de l'exponentiation rapide

L'algorithme naïf de l'exponentiation (cf. algorithme 7) qui permet d'obtenir a^n en multipliant a par lui-même n fois n'est pas très efficace : sa complexité étant en $O(n)$.

Algorithme 7 Exponentiation naïve a^n

```
1 : Fonction EXP_NAIVE(a,n)
2 :     api ← 1
3 :     pour i de 0 à n − 1 répéter
4 :         api ← api × a
5 :     renvoyer api
```

Or, l'exponentiation est une opération très récurrente qu'il est nécessaire de pouvoir exécuter le plus rapidement possible. L'exponentiation rapide (cf. algorithme 8) propose une version récursive de type diviser pour régner dont la complexité est en $O(\log n)$.

Algorithme 8 Exponentiation rapide a^n

```
1 : Fonction EXP_RAPIDE(a,n)
2 :     si n = 0 alors                          ▷ Condition d'arrêt
3 :         renvoyer 1
4 :     sinon si n est pair alors
5 :         p ← EXP_RAPIDE(a, n//2)             ▷ Appel récursif
6 :         renvoyer p × p
7 :     sinon
8 :         p ← EXP_RAPIDE(a, (n-1)//2)         ▷ Appel récursif
9 :         renvoyer p × p × a
```

L'analyse de l'algorithme 8 montre que :

- c'est un cas particulier d'algorithme diviser pour régner avec $r = 1$ et $d = 2$, c'est à dire un seul appel récursif par chemin d'exécution et une division de la taille du problème par deux[2],
- l'évolution du coût ne dépend pas de a mais de n, c'est à dire l'exposant.

On peut procéder de la même manière qu'avec l'algorithme 6 pour calculer la complexité et s'appuyer sur l'arbre de la figure 2. Pour simplifier le calcul, on peut considérer que la taille du problème est divisée par deux. Le coût hors appel récursif est constant car il s'agit de multiplications. On a donc $T(n) = O(\log n)$.

2. à un près si n est pair

H Exemple du tri fusion

Les tris génériques abordés jusqu'à présent, par sélection ou insertion, présentent des complexités polynomiales en $O(n^2)$ dans le pire des cas. L'algorithme de tri fusion a été inventé par John von Neumann en 1945. C'est un bel exemple d'algorithme de type diviser pour régner avec $r = 2$ et $d = 2$, c'est à dire deux appels récursifs par chemin d'exécution et une division de la taille du problème par deux (cf . figure 3). Il permet de dépasser cette limite et d'obtenir un tri générique de complexité logarithmique. Ce tri est comparatif, il peut s'effectuer en place et les implémentations peuvent être stables.

Son principe (cf. algorithmes 9, 11 et 10) est simple : transformer le tri d'un tableau à n éléments en sous-tableaux ne comportant qu'un seul élément [3] puis les recombiner en un seul tableau en conservant l'ordre. L'algorithme est divisé en deux fonctions :

- TRI_FUSION qui opère concrètement la division et la résolution des sous-problèmes,
- FUSION qui combine les solutions des sous-problèmes en fusionnant deux sous-tableaux triés.

Il n'y a pas de pire ou meilleur cas : l'algorithme effectue systématiquement la découpe et la fusion des sous-tableaux.

Pour le calcul de la complexité, on a la relation de récurrence $T(n) = 2T(n/2) + f(n)$ où $f(n)$ représente le nombre d'opérations élémentaires nécessaires pour fusionner deux sous-tableaux de taille $n/2$. La complexité de la fonction FUSION est linéaire, car on effectue n fois les instructions élémentaires de la boucle. Donc on peut simplifier la récurrence en $T(n) = 2T(n/2) + n$.

On fait l'hypothèse que n est une puissance de deux pour simplifier le calcul. Soient les suites auxiliaires $u_k = T(2^k)$ et $v_k = u_k/2^k$. La récurrence s'écrit alors :

$$T(2^k) = T(2^{k-1}) + 2^k = u_k = u_{k-1} + 2^k$$

On en déduit que la suite v_k vérifie : $v_k = v_{k-1} + 1$. v^k est une suite arithmétique de raison 1. Si on suppose que $u_0 = 0$, c'est-à-dire le coût de traitement d'un tableau vide est nul, alors $v_0 = 0$. On en déduit que : $v_k = v_0 + k \times 1 = k$ et donc :

$$u_k = k2^k = T(2^k)$$

La taille du tableau étant $n = 2^k$, la complexité de l'algorithme est $\mathcal{O}(n \log_2 n)$ dans **tous** les cas (le pire comme le meilleur).

3. et donc déjà triés !

Algorithme 9 Tri fusion

```
1 : Fonction TRI_FUSION(t)
2 :     n ← taille de t
3 :     si n < 2 alors
4 :         renvoyer t
5 :     sinon
6 :         t1, t2 ← DÉCOUPER_EN_DEUX(t)
7 :         renvoyer FUSION(TRI_FUSION(t1),TRI_FUSION(t2) )
```

Algorithme 10 Découper en deux

```
1 : Fonction DÉCOUPER_EN_DEUX(t)
2 :     n ← taille de t
3 :     t1, t2 ← deux listes vides
4 :     pour i = 0 à n//2 − 1 répéter
5 :         AJOUTER(t1, t[i])
6 :     pour j = n//2 à n − 1 répéter
7 :         AJOUTER(t2, t[j])
8 :     renvoyer t1, t2
```

Algorithme 11 Fusion de deux sous-tableaux triés

```
1 : Fonction FUSION(t1, t2)
2 :     n1 ← taille de t1
3 :     n2 ← taille de t2
4 :     n ← n1 + n2
5 :     t ← une liste vide
6 :     i1 ← 0
7 :     i2 ← 0
8 :     pour k de 0 à n - 1 répéter
9 :         si i1 ⩾ n1 alors
10 :            AJOUTER(t, t2[i2])
11 :            i2 ← i2 + 1
12 :        sinon si i2 ⩾ n2 alors
13 :            AJOUTER(t, t1[i1])
14 :            i1 ← i1 + 1
15 :        sinon si t1[i1] ⩽ t2[i2] alors
16 :            AJOUTER(t, t1[i1])
17 :            i1 ← i1 + 1
18 :        sinon
19 :            AJOUTER(t, t2[i2])
20 :            i2 ← i2 + 1
21 :    renvoyer t
```

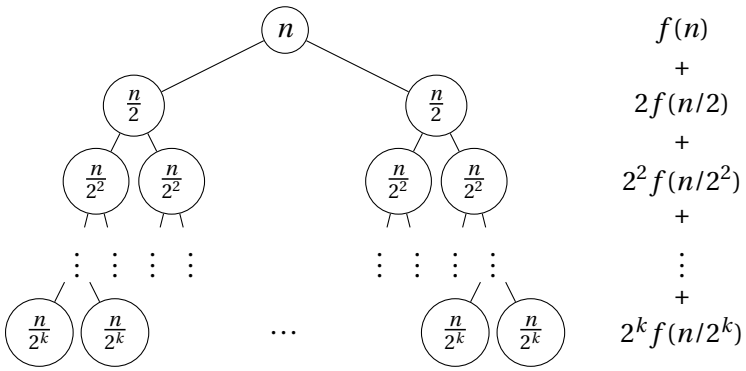

FIGURE 3 – Structure d'arbre et appels récursifs pour le tri fusion : $T(n) = 2T(n/2) + f(n)$ et $\frac{n}{2^k} = 1$. La fonction FUSION opère un nombre d'opérations $f(n)$.

I Exemple du tri rapide

La complexité du tri rapide est $\mathcal{O}(n \log_2 n)$ dans le meilleur cas et en moyenne. Cependant, dans le pire des cas, si on choisit systématiquement mal le pivot, c'est-à-dire si on prend le plus petit élément ou le plus grand, alors la complexité est en $\mathcal{O}(n^2)$, car cela revient à faire un tri par sélection.

Algorithme 12 Tri rapide

```
1 : Fonction TRI_RAPIDE(t)
2 :     n ← taille de t
3 :     si n < 2 alors
4 :         renvoyer t
5 :     sinon
6 :         t1, pivot, t2 ← PARTITION(t)
7 :         renvoyer CONCATÉNER(TRI_RAPIDE(t1), pivot, TRI_RAPIDE(t2))
```

Algorithme 13 Partition en deux sous-tableaux

```
1 : Fonction PARTITION(t)
2 :     n ← taille de t
3 :     pivot ← 0
4 :     i_pivot ← un nombre au hasard entre 0 et n − 1 inclus
5 :     t1, t2 ← deux listes vides
6 :     pour k = 0 à n répéter
7 :         si k = i_pivot alors
8 :             pivot ← t[k]
9 :         sinon si t[k] ⩽ t[i_pivot] alors
10 :            AJOUTER(t1, t[k]))
11 :        sinon
12 :            AJOUTER(t2, t[k]))
13 :    renvoyer t1, pivot, t2
```

J Synthèse

M **Méthode 1 — Complexité d'une fonction** Pour trouver la complexité d'une fonction :

1. Trouver le(s) paramètre(s) de la fonction étudiée qui influe(nt) sur la complexité.
2. Déterminer si, une fois ce(s) paramètre(s) fixé(s), il existe un pire ou un meilleur des cas.
3. Calculer la complexité en :
   - calculant éventuellement une somme d'entiers (fonction itérative),

- posant une formule récurrente sur la complexité (fonction récursive).

Le tableau 3 récapitule les complexités des algorithmes récursifs à connaître.

Récurrence	**Complexité**	**Algorithmes**
$T(n) = 1 + T(n-1)$	$\rightarrow O(n)$	factorielle
$T(n) = 1 + T(n/2)$	$\rightarrow O(\log n)$	dichotomie, exponentiation rapide
$T(n) = n + 2T(n/2)$	$\rightarrow O(n \log n)$	tri fusion, transformée de Fourier rapide

TABLE 3 – Récurrences et complexités associées utiles et à connaître

K Complexité de l'algorithme d'Euclide

La complexité de l'algorithme d'Euclide n'est pas triviale à mesurer. Pour y parvenir, nous allons nous appuyer sur la suite des restes et la comparer à la suite de Fibonacci. On note également que n, l'indice du dernier reste non nul donne directement une mesure de la complexité de la boucle.

Algorithme 14 Algorithme d'Euclide (optimisé)

```
1 : Fonction PGCD(a, b)                          ▷ On suppose que (a, b) ∈ ℤ, b ⩽ a.
2 :     a ← | a |
3 :     b ← | b |
4 :     r ← a mod b
5 :     tant que r > 0 répéter                   ▷ On connaît la réponse si r est nul.
6 :         a ← b
7 :         b ← r
8 :         r ← a mod b
9 :     renvoyer b                               ▷ Le pgcd est b
```

■ **Définition 4 — Suite des restes de la division euclidienne.** Soient a et b des entiers. On définit la suite des restes de la division euclidienne comme suit :

$$r_0 = |\, a \,| \tag{19}$$
$$r_1 = |\, b \,| \tag{20}$$
$$q_k = \lfloor r_{k-1} / r_k \rfloor, 1 \leqslant k \leqslant n \tag{21}$$

Alors on a :

$$r_{k-1} = q_k r_k + r_{k+1} \tag{22}$$
$$r_{k+1} = r_{k-1} \bmod r_k \tag{23}$$

Théorème 2 — Stricte décroissance de $(r_n)_{n\in\mathbb{N}}$. La suite des restes de la division euclidienne est positive, strictement décroissante et minorée par zéro.

Théorème 3 — Quotients de la suite des restes. Soit n l'indice de la suite des restes correspondant au dernier reste non nul. Alors on a :

$$\forall k \in [\![1, n-1]\!],\ q_k \geqslant 1 \tag{24}$$

$$k = n,\ q_n \geqslant 2 \tag{25}$$

Démonstration. D'après la proposition 2, $r_{k-1} > r_k > r_{k+1}$. De plus, q_k est un entier strictement positif d'après l'équation 21. Donc $\forall k \in [\![1, n-1]\!], q_k \geqslant 1$.

Par ailleurs, si q_n valait 1, alors on aurait $r_n = r_{n-1}$, ce qui n'est pas possible car la suite est strictement décroissante. C'est pourquoi, $q_n \geqslant 2$. ■

Théorème 4 — Des restes et des éléments de la suite de Fibonacci. Soit n l'indice du dernier reste non nul de la suite des restes de la division euclidienne.

Soit f_i le i$^{\text{e}}$ terme de la suite de Fibonacci définie par $f_{i+1} = f_i + f_{i-1}$, $f_0 = 1$ et $f_1 = 1$. Alors on a :

$$\forall k \in [\![0, n]\!], r_k \geqslant f_{n-k}. \tag{26}$$

Démonstration. D'après le théorème 3, on a $r_{n-1} = q_n r_n \geqslant 2 = f_2$, car r_n est non nul. Comme la suite des restes est décroissante, $r_{n-1} \geqslant 2r_n$ et donc $r_n \geqslant f_2/2 = f_1$. En réitérant n fois ce raisonnement en faisant décroitre n, c'est-à-dire en remontant la suite des restes, on trouve que $r_0 \geqslant f_n$ ainsi que tous les résultats. ■

Supposons qu'il y a n étapes lors de l'algorithme d'Euclide. Alors, on a

$$b = r_1 \geqslant f_{n-1} \tag{27}$$

Or, la suite de Fibonacci est une suite récurrente linéaire d'ordre deux. On connaît donc sa forme explicite.

$$f_n = \frac{1}{\sqrt{5}}\left(\phi^n - \left(-\frac{1}{\phi}\right)^n\right) \tag{28}$$

avec le nombre d'or $\phi = \frac{1+\sqrt{5}}{2}$. De plus, on peut montrer que

$$f_n \simeq \phi^n \tag{29}$$

et donc, au rang $n-1$, on a :

$$\log(f_{n-1}) \simeq (n-1)\log(\phi) \tag{30}$$

En utilisant l'équation 27, on en conclut que :

$$n \geqslant 1 + \frac{\log(b)}{\log(\phi)} \tag{31}$$

La complexité de l'algorithme d'Euclide est donc $O(1+1,44\log_2(b))$, où $\log_2(b)$ est le nombre bits nécessaires pour coder b. Elle est logarithmique en fonction de la taille du codage l'entier et donc très efficace, ce qui est très important pour les opérations de chiffrement ou de codage. Cette conclusion est également le théorème de Lamé.

Théorème 5 — Théorème de Lamé. Le nombre de divisions euclidiennes nécessaires pour calculer PGCD(a,b) par l'algorithme d'Euclide est inférieur ou égal à 5 fois le nombre de chiffres de b en base 10.