Complexité, coût et logarithme base 2

1. Définitions

1.1. Les 4 niveaux d'un problème informatique

Résoudre un problème informatique implique souvent de distinguer **quatre niveaux** :

- le **problème** à résoudre ;
- l'**algorithme** choisi (ou conçu) pour y parvenir ;
- les **structures de données** employées pour opérer ;
- et enfin le **programme** qui *implémente* cet algorithme dans un *langage* de programmation.

Notion	Question posée	Exemples
Problème	Que veut-on faire ?	Trier une liste, chercher une valeur, rendre la monnaie.
Algorithme	Quelle méthode choisit-on ?	Tri fusion, tri par insertion, recherche dichotomique…
Structure de données	Comment organise-t-on les données à traiter ?	Tableau, dictionnaire, liste chaînée…
Implémentation	Comment écrit-on *concrètement* le programme ?	Python (avec indices ? compréhensions ? *slicing* ?) C avec tableau auxiliaire ? OCaml ?

💭 **Remarque : étanchéité des niveaux ?** ▼

Ces niveaux se chevauchent partiellement : le choix d'une structure de données fait souvent partie de la conception algorithmique, car il influe sur les opérations réalisables et leur « coût d'utilisation ». Mais ce choix peut également être contraint par le langage choisi pour implémenter l'algorithme : ainsi, le C ne propose pas nativement d'équivalent aux dictionnaires Python.

1.2. Complexité vs coût

L'analyse de la complexité s'applique d'abord à un algorithme : elle permet de dire « combien *cette façon* de résoudre un problème est efficace ». Le coût, lui, dépend de l'algorithme, mais aussi de son *implémentation* en code (ainsi que de la machine employée, en pratique).

On peut dire, en première approximation, que :

- le **coût** désigne la quantité de ressources *effectivement consommées* : temps, mémoire, nombre d'opérations, etc.

On peut donc parler de coût en temps, de coût en mémoire, ou encore de coût en nombre de comparaisons (selon ce que l'on choisit de compter) ;

- la **complexité** décrit *comment* ***augmente*** *le coût* quand la *taille des données d'entrée* ***croît*** (on parle *d'évolution asymptotique*, pour décrire une « tendance à long terme »). Elle donne donc *l'ordre de grandeur* de l'évolution du coût en fonction de celle de la taille des données d'entrée.

> 💭 **Remarque : notation de Landau ▼**
>
> On emploie souvent la *notation de Landau* : $O(n)$, $O(n \log n)$ ou $O(n^2)$ ou... Elle **ne** donne **pas** une mesure *exacte* (temps en secondes ou mémoire en octets).
>
> Les expressions après le O indiquent ***comment*** *le coût croît* quand les données d'entrée grossissent (dans ce contexte, n représente leur taille).
>
> **Si la notation elle-même n'est pas explicitement au programme, les « formules en n » le sont, elles.**

> 💭 **Remarque : deux « types » de coûts ! ▼**
>
> Précisions sur les deux types de coûts évoqués précédemment.
>
> - Le **coût en temps** indique comment évolue le temps d'exécution d'un algorithme avec l'augmentation de la taille des données d'entrée.
> - Le **coût en espace** fait de même pour la quantité de mémoire vive (RAM) nécessaire à l'exécution d'un algorithme, avec l'augmentation de la taille des données d'entrée.
>
> *Souvent, on doit arbitrer entre l'un et l'autre coût.* On est régulièrement amené à sacrifier l'occupation en mémoire vive pour gagner en temps d'exécution (voir la notion de *mémoïsation* dans la fiche sur la récursivité).

1.3. Notation « O » et signification

La définition exacte de la notion $O(n)$ est *hors-programme* : retenez que cela signifie à peu près « de l'ordre de n ». Dire d'un algorithme que « son temps d'exécution est en $O(n)$ » signifie donc *en gros* que ce temps d'exécution augmente *proportionnellement* à n.

Concrètement, n désigne la taille des données à traiter : on dit aussi *« taille de l'entrée »*. Un temps d'exécution en $O(n)$ implique que si l'on multiplie par 2 la taille de ces données d'entrée, alors le temps d'exécution est lui aussi multiplié par *environ* 2. Il en va de même si on multiplie la taille de l'entrée par 10 : le temps d'exécution sera alors multiplié par un facteur *de l'ordre* de 10.

2. Logarithme en base 2

Le **logarithme en base 2** intervient souvent en informatique, notamment quand on divise un problème par deux à chaque étape.

> ⚠️ **Quel logarithme ?**
>
> Quand on écrit $\log$ *en informatique*, c'est au $\log_2$ qu'on fait référence. Il existe en réalité une *infinité* de logarithmes différents ! Parmi les plus importants figurent :
>
> - le $\log_{10}$ des chimistes, qui intervient dans le calcul du *potentiel hydrogène* : $pH = -\log\left([H_3O^+]\right)$. *Dans ce*

contexte, log désigne le logarithme *décimal* $\log_{10}$) ;

- le $\log_e$ des mathématiciens. On le note plus communément $\ln$: c'est le « logarithme *népérien* ».

Tous ces logarithmes sont *proportionnels* les uns aux autres.

2.1. L'idée de base

Si on sépare une liste Python comptant n éléments en deux sous-listes de tailles (à peu près) égales, et qu'on répète ce processus sur chaque sous-liste plusieurs fois d'affilée (*récursivement*, donc), on finira par atteindre un ensemble de listes ne comportant qu'*un seul* élément. Le nombre total d'étapes nécessaire à ce processus vaut $\log_2(n)$ (parfois exactement, mais *approximativement* le plus souvent).

2.2. Définition mathématique

✅ **Logarithme en base 2**

Pour un nombre $n > 0$, $k = \log_2(n)$ est défini par $2^k = n$.

$\log_2(n)$ est donc la *puissance* à laquelle il faut élever 2 pour obtenir n.

💭 **Remarque : définition implicite** ▼

C'est le même genre de définition dite *implicite* que celle qu'on emploie pour définir la racine carrée. Pour mémoire, $\sqrt{x}$ est défini comme « le nombre positif y tel que $y^2 = x$ ».

Vous aurez avec $\log_2 x$ les mêmes difficultés qu'avec $\sqrt{x}$ en seconde. Avec la pratique vient l'habitude : on s'y fait et on « apprend à gérer ».

⚠️ **Ne pas confondre logarithme et puissance !**

$\log_2(n)$ dit « combien de fois » il faut multiplier par 2 pour atteindre n en partant de 1.

Vu « à l'envers », $\log_2(n)$ indique « combien de fois » diviser n par 2 pour atteindre 1.

Ce qui est compliqué (au début), c'est que ce « nombre de fois » **n'**est le plus souvent **pas** *entier*...

2.3. Exemple « à la main »

On a $2^3 = 8$, donc par définition $\log_2(8) = 3$. De même, $\log_2(16) = 4$. La difficulté, c'est de trouver $\log_2(x)$ pour $x \in]8\,;16[$... Mais cela a peu d'importance en NSI : c'est le *comportement* de la *fonction logarithme* lorsque n devient grand qui nous intéresse, **pas** le calcul de valeurs *numériques* (se reporter au graphique ci-dessous, permettant de comparer différentes évolutions).

Comprendre ce qui suit est important, en revanche.

Si vous voulez découper la liste Python `[1, 2, 3, 4, 5, 6, 7, 8, 9]` en un lot de 9 listes contenant chacune un unique élément, vous aurez besoin de **4 étapes** :

- étape 1 : `[1, 2, 3, 4]` et `[5, 6, 7, 8, 9]` ;

- étape 2 : `[1, 2]` ; `[3, 4]` ; `[5, 6]` et `[7, 8, 9]` ;
- étape 3 : `[1]` ; `[2]` ; `[3]` ; `[4]` ; `[5]` ; `[6]` ; `[7]` et `[8, 9]` ;
- étape 4 : `[1]` ; `[2]` ; `[3]` ; `[4]` ; `[5]` ; `[6]` ; `[7]` ; `[8]` et `[9]`.

Donc *l'arrondi à l'entier* ***supérieur*** de $\log_2(9)$ vaudra **4**. Et il faudra toujours *exactement* ***4*** *étapes* pour obtenir des listes d'un seul élément, à partir de *n'importe quelle liste* comptant ***au plus*** **16 éléments**. Mais dès le 17e élément, et jusqu'au 32e inclus, il faudra *exactement* ***5*** *étapes*. Et ainsi de suite.

2.4. Exemple simple et lien entre nombre de bits et $\log_2$

```
n = 1024
k = 0
while n > 1:
    n = n // 2
    k += 1
```

Ici, `k` compte le nombre de *divisions par 2* nécessaires pour « descendre jusqu'à 1 ». Pour `1024`, on obtient `10`, ce qui correspond à $\log_2(1024)$.

Si à chaque étape, en plus d'effectuer la division entière par 2, on garde la trace du reste de cette division entière (par exemple dans une liste), on obtient l'écriture binaire de l'entier `n` initial (il faut cette fois atteindre `0`).

```
n = 1024
bits = []
k = 0
while n > 0:            # Attention ici : on s'arrête à 0, pas à 1 !
    bits.append(n % 2)
    n = n // 2
    k += 1
bits.reverse()          # Mettre les bits dans « le bon ordre » (gros-boutisme)
n_bin = "".join([str(b) for b in bits])     # Création d'une chaîne
len(n_bin)              # Renvoie 11
```

> ✅ **Lien entre nombre de bits d'un entier et logarithme en base 2**
>
> L'écriture binaire (en base 2) d'un entier n (non nul) comporte $\log_2(n) + 1$ bits.

2.5. Exemple plus avancé

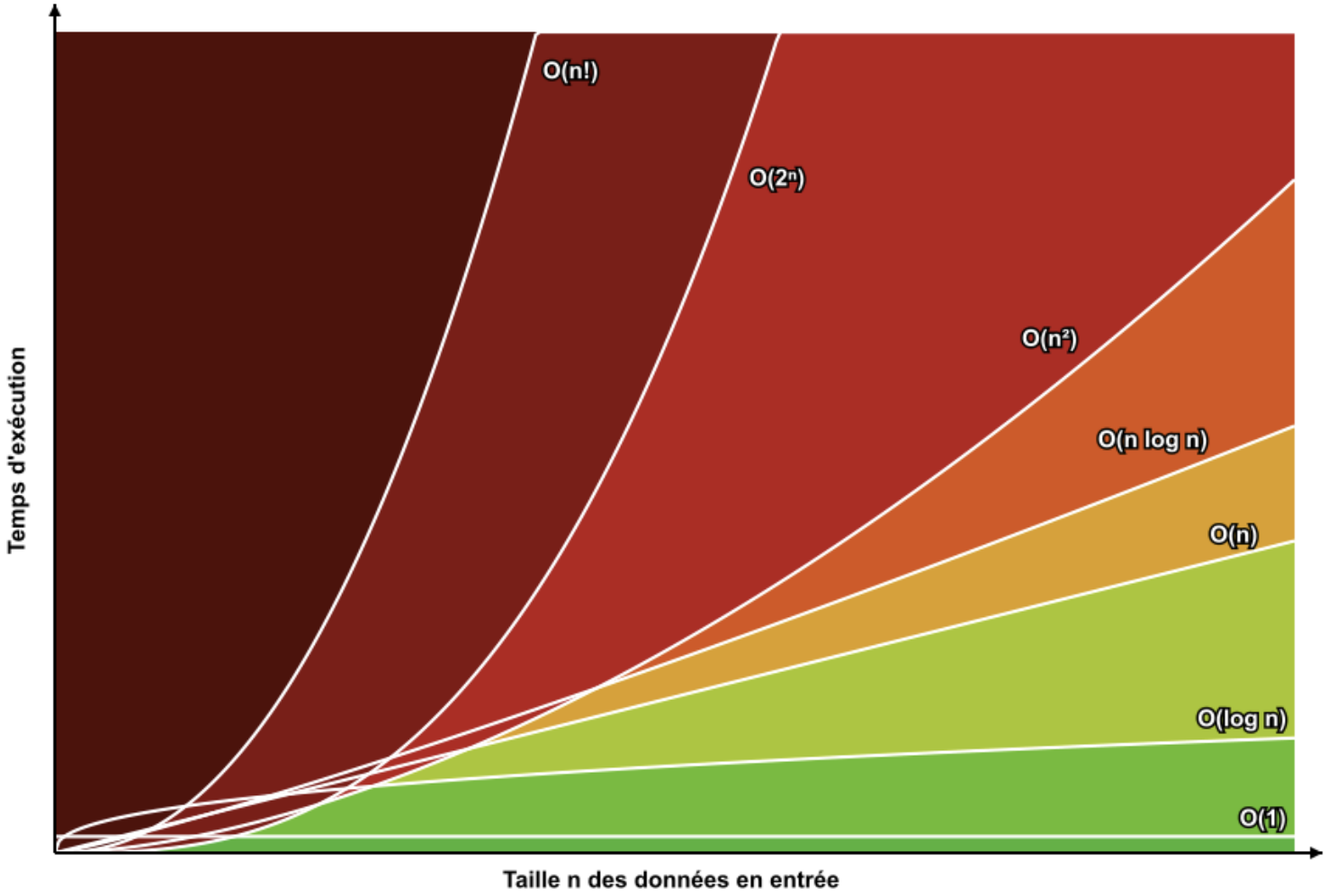
Le **tri-fusion** (https://fr.wikipedia.org/wiki/Tri_fusion) découpe la liste en deux à chaque étape, puis fusionne les sous-listes triées. Cela conduit à une complexité dite « en $O(n \log n)$ » — sous réserve de *bien* implémenter l'algorithme en utilisant une approche adaptée (évitant des copies inutiles ou des manipulations coûteuses).

3. Catégorisation selon la classe de complexité

Il existe différentes *classes de complexité*, selon l'expression entre les parenthèses de la notation $O(\dots)$.

Classe de complexité	Appréciation	Remarque	Exemple
$O(1)$	🤩 Idéal	Temps (à peu près) constant	Accès à une valeur dans un dictionnaire Python
$O(\log n)$	🥰 Excellent	Évolution logarithmique	Recherche dichotomique dans une liste triée
$O(n)$	😊 Bon	Évolution *linéaire*	Recherche d'élément dans une liste
$O(n \log n)$	😐 Pas mal	Évolution *quasi-linéaire*	Tri-fusion
$O(n^2)$	😔 Pas bon	Évolution *quadratique*	Tri par insertion ou sélection
$O(2^n)$	😱 Mauvais	Évolution exponentielle	Calcul récursif naïf de Fibonacci
$O(n!)$	🤮 Exécrable	Évolution combinatoire	Énumération de toutes les permutations d'une liste

- En informatique, il est important de comparer les ordres de grandeur : $O(1)$ est infiniment meilleur que $O(\log n)$, qui est *bien meilleur* que $O(n)$, qui est préférable à $O(n \log n)$, qui **enterre** $O(n^2)$, etc.



- La complexité est directement liée à la structure de l'algorithme.

4. Liens avec d’autres notions

- Le **tri fusion**.
- La **recherche dichotomique**.
- La **programmation dynamique**.
- La **récursivité**, qui rend souvent visible la structure en division par 2.