

Algorithmique et complexité

François Pirot

10 novembre 2023

Table des matières

À propos du cours	5
1 Objectifs	5
2 Organisation	5
3 Évaluation	5
1 Introduction	7
1 Rappels d'algorithmique	7
1.1 Définition	7
1.2 Langage descriptif d'un algorithme	7
1.3 Variables et types élémentaires	8
2 Instructions conditionnelles et itératives	8
2.1 Instruction Si-Alors-Sinon	8
2.2 Boucle Pour	10
2.3 Boucles Tant-que et Répéter	11
2.4 Des conditions plus complexes	12
3 Terminaison et correction	12
3.1 Terminaison	12
3.2 Correction	13
2 Structures de données et complexité	15
1 Structurer les données	15
1.1 Les types de base	15
1.2 Les types construits	15
2 Estimation de la complexité d'un algorithme	16
2.1 Présentation du concept de complexité	16
2.2 La notation $O(\cdot)$	16
2.3 Les opérations élémentaires	17
3 Calcul théorique de la complexité	17
3.1 Pire cas, meilleur cas, moyenne	17
3.2 Décomposition du code par blocs	17
3.3 Exemples d'analyse de complexité	18
3.4 Amélioration de la complexité d'un algorithme	19
3.5 Exemple de calcul complexe : Crible d'Eratosthène	20
3 Récursivité	23
1 Fonctions récursives	23
1.1 Définition et usages	23
1.2 Récursion terminale	23
2 Structures récursives	24
2.1 Définition récursive des listes	24

2.2	Arbres	25
3	Analyse d'algorithmes récursifs	25
3.1	Arbre d'appels récursifs	25
3.2	Complexité	25
4	Tris	27
1	Quelques exemples de tri	27
1.1	Tris quadratiques	27
1.2	Tris pseudo-linéaires	28
2	Complexité générale d'un tri	30

À propos du cours

1 Objectifs

- Écrire une séquence d'instructions de façon structurée en pseudo-code.
- Savoir prouver la terminaison et la correction d'un algorithme (introduction de la notion d'invariant de boucle, preuves par induction).
- Estimation de la complexité d'un algorithme.

2 Organisation

- Pendant les TD, on écrit les algorithmes en pseudo-code. On se focalise sur la structure de l'algorithme plutôt que sur son implémentation.
- Dans les TP, on utilise le langage Python. On pourra utiliser l'interpréteur de son choix. L'interpréteur Thonny est très bien, car il permet de visualiser l'exécution des algorithmes et accompagne très bien la phase de débogage.

3 Évaluation

- **Projet** : à faire par binôme ; le sujet sera donné lors de la dernière séance de TP (semaine du 4 décembre 2023). Il y aura un rapport et un fichier de code à rendre par binôme pour le dimanche 17 décembre 23 :59.
- **Évaluation finale** : DS le vendredi 8 décembre, 9h-11h (11h45 pour les tiers-temps).

Chapitre 1

Introduction

1 Rappels d'algorithmique

1.1 Définition

Un *algorithme* est l'organisation structurée d'*opérations élémentaires* afin de réaliser une tâche complexe. On en retrouve dans la vie de tous les jours (exemple : recette de cuisine, notice de montage, itinéraire routier). Les opérations élémentaires dépendent du contexte de l'algorithme ; il s'agit d'opérations simples et rapides à effectuer (exemple en cuisine : éplucher une pomme de terre, casser un oeuf, mélanger une préparation ; exemple en informatique : opérations arithmétiques, affichage/lecture d'une chaîne de caractères, comparaison de deux valeurs). Des opérations de complexité intermédiaire intervenant très souvent peuvent également être définies sous forme de *macros*, et être utilisées comme des opérations élémentaires, en restant bien conscient de leur coût (exemple : préparer une béchamel, trier une liste d'entiers).

1.2 Langage descriptif d'un algorithme

Un algorithme s'écrit en *pseudo-code*. Il s'agit d'un langage descriptif générique pouvant être implémenté dans n'importe quel langage de programmation adapté (une telle implémentation s'appelle un *programme*). À la compilation, le programme est traduit en langage machine ; il s'agit alors d'un *processus* pouvant être exécuté sur une entrée donnée. Un algorithme s'organise généralement selon la structure suivante.

Algorithme 1 : Nom de l'algorithme

Entrées : Paramètres de l'algorithme

début

 Déclaration et initialisation de variables

 Séquence d'instructions

fin

retourner résultat

Par exemple, nous pouvons avoir la séquence d'instructions suivante.

Algorithme 2 : Salutation

début

 Afficher("Comment vous appelez-vous ?")

$x \leftarrow$ SaisieUtilisateur()

 Afficher("Bonjour ", x , " !")

fin

Le corps d'un algorithme s'écrit en pseudo-code, et fait appel à des instructions élémentaires (ici `Afficher`, `SaisieUtilisateur`). Dans le cadre de l'informatique, celles-ci font référence à des opérations machine décrites de façon informelle, et dont l'implémentation dépendra du langage machine utilisé.

Un algorithme qui a une valeur de retour est aussi appelé une *fonction*. S'il n'y a pas de valeur retour, on parle de *procédure*.

1.3 Variables et types élémentaires

Une *variable* est constituée d'un nom et d'un contenu, ce contenu étant d'un certain *type*. Parmi les types élémentaires, on peut citer : booléen, caractère, chaîne de caractères, nombre entier, nombre flottant, etc.

Pour la clarté de l'algorithme, il peut être intéressant de déclarer les variables utilisées et leur type au tout début. Quand l'algorithme sera traduit en programme cette déclaration aura d'autres avantages : réservation de l'espace mémoire correspondant au type, possibilité de vérifier le programme du point de vue de la cohérence des types, etc. Par tradition, on notera l'affectation d'une valeur à une variable en utilisant le symbole flèche vers la gauche (\leftarrow).

Deux notions différentes de variables apparaissent dans un algorithme : les *paramètres* dont dépendent le résultat, et les *variables internes* qui servent à la résolution et sont invisibles en dehors de l'exécution de l'algorithme.

2 Instructions conditionnelles et itératives

Un certain nombre de tâches nécessitent de répéter un groupe d'actions soit un nombre de fois donné, soit jusqu'à ce qu'une condition soit remplie. D'autres cas nécessitent de pouvoir choisir un comportement parmi plusieurs en fonction d'un certain nombre de conditions.

2.1 Instruction Si-Alors-Sinon

Démarrons par un exemple :

Algorithme 3 : QueFaireCeSoir

```
si Pluie alors
| MangerPlateauTélé()
| SeCoucherTôt()
sinon
| MangerAuRestaurant()
| AllerAuCinema()
fin
```

La structure `Si-Alors-Sinon` permet de tester une condition (ici "Pluie"). Si cette condition s'avère remplie (il pleut effectivement), alors les actions contenues dans le bloc `Alors` sont exécutées. Si elle s'avère fautive (pas de pluie à l'horizon), alors les actions du bloc `Sinon` sont exécutées.

Le bloc `Sinon` est optionnel. En son absence, si la condition n'est pas remplie, aucune action n'est exécutée.

Il est possible d'imbriquer les instructions `Si-Alors-Sinon` afin de tester des conditions complexes. Par exemple, si l'on cherche à afficher trois éléments par ordre croissant :

Algorithme 4 : AfficherDansLOrdre

```
Entrées :  $x, y, z$  : entiers
si  $x < y$  alors
  | si  $y < z$  alors
  | | Afficher( $x, y, z$ )
  | sinon
  | | si  $x < z$  alors
  | | | Afficher( $x, z, y$ )
  | | sinon
  | | | Afficher( $z, x, y$ )
  | | fin
  | fin
sinon
  | si  $x < z$  alors
  | | Afficher( $y, x, z$ )
  | sinon
  | | si  $y < z$  alors
  | | | Afficher( $y, z, x$ )
  | | sinon
  | | | Afficher( $z, y, x$ )
  | | fin
  | fin
fin
```

Dans ce cas, il peut être utile d'anoter le code avec des prédicats qui sont vérifiés au sein des blocs. Plutôt que d'imbriquer des conditions Si-Alors-Sinon de façon arborescente, il est parfois plus commode de composer les conditions afin de pouvoir immédiatement conclure à l'aide d'une disjonction de cas qui prendra une forme linéaire qui en simplifie la compréhension. Il y aura alors souvent plus que deux issues possible, ce qui s'exprime à l'aide d'un Sinon-Si. En reprenant le même exemple :

Algorithme 5 : AfficherDansLOrdre

```
Entrées :  $x, y, z$  : entiers
si  $x < y$  et  $y < z$  alors
  | Afficher( $x, y, z$ )
sinon si  $x < z$  et  $z < y$  alors
  | Afficher( $x, z, y$ )
sinon si  $y < x$  et  $x < z$  alors
  | Afficher( $y, x, z$ )
sinon si  $y < z$  et  $z < x$  alors
  | Afficher( $y, z, x$ )
sinon si  $z < x$  et  $x < y$  alors
  | Afficher( $z, x, y$ )
sinon
  | Afficher( $z, y, x$ )
fin
```

Dans ce cas, on peut être redondant dans les conditions sans impact sur l'exécution. Cela simplifie la démarche de conception de l'algorithme et rassure sur sa correction.

Il est parfois plus pertinent d'utiliser des variables internes qui devront prendre des valeurs précises afin de simplifier les instructions. En général, on a au moins une variable interne `res` dont la valeur à la fin de l'exécution

de l'algorithme est le résultat attendu. L'algorithme précédent peut ainsi être simplifié.

Algorithme 6 : AfficherDansLOrdre

```
Entrées :  $x, y, z$  : entiers  
 $a \leftarrow x$   
 $b \leftarrow y$   
 $c \leftarrow z$   
si  $b < a$  alors  
| Echanger( $a, b$ )  
fin  
si  $c < b$  alors  
| Echanger( $b, c$ )  
fin  
si  $b < a$  alors  
| Echanger( $a, b$ )  
fin  
Afficher( $a, b, c$ )
```

2.2 Boucle Pour

Une boucle `Pour` permet de répéter une même instruction un nombre donné de fois, tout en ayant accès à un compteur qui renseigne sur l'itération en cours. Par exemple, l'algorithme suivant dessine une ligne dont la longueur est donnée en argument.

Algorithme 7 : DessineLigne

```
Entrées :  $n$  : entier  
pour  $i$  de 1 à  $n$  faire  
| Afficher("-")  
fin
```

Il est également possible d'imbriquer les boucles `Pour`, ce qui est indispensable pour travailler en deux dimensions (dans les matrices, sur une image). Par exemple, pour dessiner un carré d'étoiles dont la longueur du côté est donnée en argument.

Algorithme 8 : DessineCarre

```
Entrées :  $n$  : entier  
pour  $i$  de 1 à  $n$  faire  
| pour  $j$  de 1 à  $n$  faire  
| | Afficher("*")  
| fin  
| RetourALaLigne()  
fin
```

Chaque boucle `Pour` introduit une variable entière, le compteur associé. Si l'on veut être formel, on devrait toutes les déclarer en tant que variables internes de la fonction. Cependant, chaque compteur n'a de sens qu'au sein du corps de sa boucle, et n'a aucune utilité en dehors. Ainsi, on préfère en général garder leur déclaration implicite par l'en-tête de la boucle.

On peut bien sûr se servir de cette variable au sein de l'instruction à répéter. Pour compter de 1 à 100, il suffit de faire l'instruction suivante.

```
pour  $i$  de 1 à 100 faire  
| Afficher( $i$ )  
fin
```

Enfin, on a la liberté d'initialiser le compteur à n'importe quelle valeur i_0 (en général $i_0 = 0$ ou $i_0 = 1$), et de l'incrémenter de n'importe quel pas r , même négatif (par défaut $r = 1$). Cela peut se simuler aisément en utilisant une variable $i' \leftarrow i_0 + i * r$.

2.3 Boucles Tant-que et Répéter

Parfois, on veut répéter une instruction un nombre indéterminé de fois, tant qu'une condition est remplie, ou bien avec une condition d'arrêt spécifique. Par exemple :

Algorithme 9 : Rouler

```
Entrées : vitesseLimite  
vitesse  $\leftarrow$  0  
tant que  $vitesse < vitesseLimite$  faire  
| Accélérer()  
fin  
MaintenirAllure()
```

Ou encore

Algorithme 10 : Rouler

```
Entrées : vitesseLimite  
vitesse  $\leftarrow$  0  
répéter  
| Accélérer()  
jusqu'à  $vitesse \geq vitesseLimite$   
MaintenirAllure()
```

À noter que, contrairement à la boucle tant que, on passe toujours au moins une fois dans une boucle répéter. Ainsi, dans l'exemple ci-dessus, on commence forcément par accélérer, ce qui veut dire qu'on ne doit l'utiliser que si l'on sait déjà que la vitesse limite est strictement positive.

Il est possible de simuler une boucle Pour à l'aide d'une boucle Tant-que. Dans ce cas, il faut déclarer la variable correspondant au compteur au préalable, et l'incrémenter à la fin du corps de la boucle. Pour compter de 1 à 100, cela donne :

```
 $i \leftarrow$  1  
tant que  $i \leq 100$  faire  
| Afficher( $i$ )  
|  $i \leftarrow i + 1$   
fin
```

2.4 Des conditions plus complexes

Il arrive que l'on ait besoin de tester des conditions trop complexes pour être décrites à l'aide d'une simple expression booléenne. En particulier, on peut avoir besoin de valider un test un grand nombre de fois, ce qui peut nécessiter l'utilisation d'une boucle. En guise d'exemple, disons que l'on souhaite diminuer le budget d'une armée à condition qu'il y ait eu une période de paix d'au moins 100 ans. On pourra au préalable écrire un algorithme permettant de vérifier cette condition, puis s'en resserrer dans l'algorithme de gestion du budget.

Algorithme 11 : PaixDurable

```
annee ← 2021
paix ← Vrai
tant que paix et annee ≥ 1921 faire
  | paix ← Paix(annee)
  | annee ← annee - 1
fin
retourner paix
```

On peut maintenant décrire l'algorithme de gestion du budget de l'armée.

Algorithme 12 : BudgetArmée

```
si PaixDurable() alors
  | Diminuer(Budget)
sinon
  | Augmenter(Budget)
fin
```

3 Terminaison et correction

Un *prédicat* est une formule logique vraie sur l'ensemble de ses entrées possibles. L'analyse d'un algorithme nécessite d'établir un certain nombre de prédicats sur les valeurs des variables internes à des moments-clé de son exécution, avec comme objectif de prouver comme prédicat "*La valeur retournée correspond à la sortie attendue de l'algorithme*".

Un prédicat vrai au début de chaque itération d'une boucle s'appelle un *invariant de boucle*.

Au sein du bloc **Alors** d'un **Si-Alors-Sinon** nous avons le prédicat suivant : "*la condition du bloc Si est vraie*". Au sein du bloc **Sinon**, nous avons "*la condition du bloc Si est fausse*". À la sortie d'une boucle **Tant-que**, nous avons "*La condition du bloc Tant-que est fausse*".

3.1 Terminaison

Lors de l'utilisation d'un bloc **Tant-que**, la terminaison de l'algorithme n'est pas garantie. Prouver qu'un algorithme termine est une tâche ardue : en général, ce problème est *indécidable* (il n'existe pas d'algorithme capable de prendre en entrée un programme, et de retourner **Vrai** s'il termine, et **Faux** sinon). On peut citer la célèbre conjecture de Syracuse, qui prétend que le programme suivant termine, mais n'a toujours pas de preuve.

Nos utilisations de la boucle **Tant-que** seront des cas particuliers où il sera relativement simple de montrer qu'à chaque itération, on se rapproche de la condition de sortie de boucle. Cela peut se montrer typiquement en montrant qu'une certaine quantité à valeurs entières positives décroît strictement à chaque itération. Par exemple, pour le calcul du pgcd de deux nombres :

Algorithme 13 : Syracuse

Entrées : n : entier
 $x \leftarrow n$
tant que $x > 1$ **faire**
 si x est pair **alors**
 $x \leftarrow x/2$
 sinon
 $x \leftarrow 3x + 1$
 fin
fin

Algorithme 14 : PGCD

Entrées : n, m : entiers
 $x \leftarrow \max(n, m)$
 $y \leftarrow \min(n, m)$
tant que $y > 0$ **faire**
 $(x, y) \leftarrow (y, x \bmod y)$
fin
retourner x

Pour montrer que la condition de sortie est toujours atteinte, nous allons étudier l'évolution de la valeur de y . Dans le corps de la boucle `Tant-que`, nous avons le prédicat " $y > 0$ ". Pour toute paire d'entiers (x, y) , si $y > 0$ alors $x \bmod y < y$, et donc la valeur de y décroît strictement à chaque itération. Cela prouve bien que l'on atteint la condition de sortie de la boucle (à savoir " $y \leq 0$ ") en au plus $\min(n, m)$ (la valeur initiale de y) itérations.

3.2 Correction

Pour prouver la correction d'un algorithme, il faut prouver le prédicat "*La valeur retournée est celle que l'algorithme est censé calculer*". Dans l'exemple précédent, on utilise l'invariant de boucle " $\text{pgcd}(x, y) = \text{pgcd}(n, m)$ ". Cela se prouve en utilisant le fait que $\text{pgcd}(x, y) = \text{pgcd}(y, x \bmod y)$ pour toute paire d'entiers (x, y) . La condition de sortie de boucle nous indique qu'à la fin de l'algorithme, on a le prédicat " $y \leq 0$ ", et en fait $y = 0$ puisque la valeur de y est le résultat de la dernière opération $x \bmod y$, qui est nécessairement positif. L'invariant de boucle nous donne $\text{pgcd}(n, m) = \text{pgcd}(x, y) = \text{pgcd}(x, 0) = x$, ce qui correspond bien à la valeur retournée.

Chapitre 2

Structures de données et complexité

1 Structurer les données

1.1 Les types de base

Les types de bases sont présents dans la plupart des langages de programmation, et interviennent dans les opérations élémentaires au sein des instructions. Leur représentation a une taille fixe, ce qui permet d'assurer que les opérations élémentaires qui les utilisent ont une complexité bornée par une constante.

- *Booléens* : ils prennent la valeur `Vrai` ou `Faux`. Les variables booléennes se combinent ensemble sous la forme d'expressions logiques, servant à exprimer des conditions au sein des blocs `Si-Alors-Sinon`, ou bien de sortie de boucles `Tant-que` ou `Répéter`.
- *Entiers* : Ils prennent des valeurs comprises entre -2^{30} et $2^{30} - 1$ s'ils sont codés sur 32 bits, ou entre -2^{62} et $2^{62} - 1$ s'ils sont codés sur 64 bits. Ces bornes peuvent varier en fonction des langages de programmation ; en particulier on peut utiliser des *entiers non signés* si on n'est pas intéressé par les valeurs négatives.
- *Flottants* : Ils représentent des nombres réels avec une précision finie. Il faut avoir conscience de certaines valeurs dépendant de leur implémentation qui en définissent la limite : `float_epsilon` `float_min` `float_max` `infinity` `nan`
- *Caractères* : Il s'agit de l'ensemble des caractères de code ASCII compris entre 32 et 127, et servent à l'affichage. Ils sont donc codés sur 8 bits. Pour utiliser des caractères spéciaux de types unicode, il faut utiliser une implémentation particulière spécifique à chaque langage.

1.2 Les types construits

En partant des types de base, il est possible de construire de nouveaux types, de taille variable. Ces types construits sont indispensables pour travailler sur des structures de données complexes, dont le choix est primordial pour le bon fonctionnement d'algorithmes traitant de nombreuses données.

1.2.1 Les tableaux

Les *tableaux* sont des structures d'enregistrement de données d'un même type. Les tableaux ont une taille fixe, à laquelle on accède par la fonction élémentaire `taille`. Ils sont optimisés pour la lecture et l'écriture en une position donnée. Comme leur taille est fixe, ils faut toujours prendre garde à ne pas tenter de lire/écrire en une position qui dépasse cette taille !

Les *chaînes de caractères* (string) sont un type particulier de tableaux, dont les données sont des caractères. Elles se terminent systématiquement par un caractère spécial `EOF` qui permet de détecter aisément que l'on a atteint la fin de la chaîne.

Tableaux multidimensionnels, tableaux dynamiques

1.2.2 Les listes

Les *listes* sont des structures dynamiques de stockage de données d'un même type. Elles sont optimisées pour l'ajout d'un élément en tête, et l'extraction de l'élément de tête. Leur taille varie constamment, et y accéder est coûteux. En revanche, on peut tester si une liste est vide efficacement avec la fonction élémentaire `EstVide`. Elles sont à privilégier pour toute forme d'énumération exhaustive.

Elles peuvent prendre plusieurs formes : simplement chaînées, doublement chaînée, LIFO, FIFO. Chaque implémentation de liste permet de privilégier certains types d'opérations.

1.2.3 Les tuples

Les *tuples* permettent la réunion de plusieurs variables pouvant avoir des types différents en une seule. Ils fonctionnent de façon similaire au produit cartésien en mathématique. Ils sont en général utilisés pour retourner une solution constituée de plusieurs variables. Par exemple,

- les coordonnées d'un point en plusieurs dimensions,
- la position et la valeur d'un élément dans un tableau,
- une liste et sa taille,
- un polynôme peut être représenté à l'aide d'une liste de couples (`coefficient`, `degré`).

Le type *enregistrement* reprend le principe d'un type, en nommant et spécifiant le type de chacune de ses variables constitutives. On les utilise pour créer des structures de données complexes.

2 Estimation de la complexité d'un algorithme

2.1 Présentation du concept de complexité

Afin d'évaluer la qualité d'un algorithme, on souhaite connaître l'évolution de son temps d'exécution sur des entrées de taille croissante. Il est possible de le faire expérimentalement, mais afin d'avoir une connaissance absolue de la complexité d'un algorithme, cela doit passer par une analyse théorique.

Comme le temps d'exécution d'un algorithme dépend fortement de son implémentation (langage utilisé, optimisations du compilateur, puissance du processeur), le calcul de la complexité d'un algorithme se fait en comptant le nombre d'opérations élémentaires, plutôt que par une estimation temporelle intrinsèquement fluctuante.

2.2 La notation $O(\cdot)$

Étant donné la nature intrinsèquement imprécise de la notion de complexité temporelle, on cherche rarement à l'exprimer de façon exacte. On est davantage intéressé par son évolution sur des données de taille croissante. Pour cela, on utilise la notation $O(\cdot)$ qui permet de résumer une fonction arithmétique à son terme dominant, et à une constante multiplicative près. Plus formellement :

Definition 1. Soient deux fonctions arithmétiques $f, g: \mathbb{N} \rightarrow \mathbb{N}$. On écrit $f(n) \underset{n \rightarrow \infty}{=} O(g(n))$ s'il existe une constante $C > 0$ et un entier n_0 tels que $f(n) \leq C \cdot g(n)$ pour tout $n \geq n_0$.

En général, la précision $n \rightarrow \infty$ est implicite lorsque l'on travaille sur des fonctions arithmétiques.

Par exemple, pour tout polynôme f de degré d , on a $f(n) = O(n^d)$. En particulier, n'importe quelle constante s'écrit $O(1)$. Une fonction arithmétique en $O(n)$ est dite *linéaire*; une fonction arithmétique en $O(n^2)$ est dite *quadratique*; une fonction arithmétique en $O(n^3)$ est dite *cubique*. Une fonction arithmétique en $O(\log n)$ est dite *logarithmique* (la base du log n'importe pas puisque l'on considère la fonction à une constante multiplicative près).

On a $O(f(n)) + O(g(n)) = O(f(n) + g(n))$, et $O(f(n)) \times O(g(n)) = O(f(n) \times g(n))$.

Si on s'intéresse à des bornes inférieures, on utilise la notation $\Omega()$ qui se définit similairement.

Definition 2. Soient deux fonctions arithmétiques $f, g: \mathbb{N} \rightarrow \mathbb{N}$. On écrit $f(n) \underset{n \rightarrow \infty}{=} \Omega(g(n))$ s'il existe une constante $C > 0$ et un entier n_0 tels que $f(n) \geq C \cdot g(n)$ pour tout $n \geq n_0$.

Si $f(n) = O(g(n))$ et $f(n) = \Omega(g(n))$, on écrit $f(n) = \Theta(g(n))$.

2.3 Les opérations élémentaires

Les opérations élémentaires sont de complexité $O(1)$, c'est à dire que sur n'importe quelle entrée elles prennent un temps d'exécution constant. C'est le cas de la plupart des opérations sur les types de bases, en particulier les opérations arithmétiques sur les entiers de taille bornée (à 32 ou 64 bits).

Chaque type est optimisé pour un certain nombre d'opérations élémentaires. Pour les types construits, on a :

- Tableaux : Lecture/écriture en position i , accès à la taille du tableau.
- Liste LIFO (pile) : Ajouter un élément en tête, extraire l'élément de tête, tester si la liste est vide.
- Liste FIFO (file) : Ajouter un élément en tête, extraire l'élément en queue, tester si la liste est vide.
- Liste doublement chaînée : Accès à l'élément en tête ou en queue, trouver l'élément qui suit ou celui qui précède un élément donné.

Lorsque l'on manipule des opérations pré-implémentées pour un type donné, il faut donc savoir si elles sont élémentaires, ou avoir conscience de leur complexité dans le cas contraire.

3 Calcul théorique de la complexité

Une fois que l'on connaît la complexité de chacune des instructions qui constituent un algorithme, il convient d'assembler ces complexités de façon à calculer la complexité globale de l'algorithme, comme une fonction de la valeur de ses paramètres.

3.1 Pire cas, meilleur cas, moyenne

On distingue plusieurs types de complexité en fonction de ce qui nous intéresse. En général, on cherche la complexité dans le pire cas car elle caractérise la limitation de l'algorithme. Certains algorithmes ont cependant une bien meilleure complexité en moyenne (sur une entrée aléatoire) par rapport à leur complexité dans le pire cas, ce qui mérite d'être relevé. Enfin, deux algorithmes à la complexité similaire (en moyenne / dans le pire cas) peuvent se distinguer sur leur complexité dans le meilleur cas.

3.2 Décomposition du code par blocs

Pour calculer la complexité d'un algorithme, on calcule de façon inductive la complexité au sein de chacun de ses blocs (la complexité de chaque bloc dépend des valeurs des variables externes à ce bloc), et on combine les complexités entre les différents blocs en suivant les règles suivantes.

- Pour une instruction `Si-Alors-Sinon`, on calcule la complexité c_0 de l'évaluation de la condition du `Si`, puis les complexités c_1 et c_2 des blocs `Alors` et `Sinon`, respectivement. La complexité de l'instruction est alors $c_0 + \max(c_1, c_2)$.
- Pour une boucle `Pour i de 1 à n`, on calcule la complexité $f(i)$ du corps de la boucle à l'itération i . La complexité de l'instruction est alors $O(n) + \sum_{i=1}^n f(i)$.
- Pour une boucle `Tq` ou `Répéter`, il faut être capable de prouver que la boucle termine, et estimer le nombre d'itérations de la boucle et la complexité de chacune de ces itérations. Il n'y a pas de recette miracle, mais il faut en général utiliser des *invariants de boucle*, c'est à dire des propriétés qui restent vraies à chaque itération de la boucle.

Considérons par exemple cette fonction qui teste si un tableau est croissant.

Algorithme 15 : croissant

```
Entrées : tab : tableau  
n ← taille(tab)  
i ← 0  
res ← Vrai  
tant que i ≤ n - 2 et res faire  
|   res ← tab[i] ≤ tab[i+1]  
|   i ← i + 1  
fin  
retourner res
```

Ici l'invariant de boucle est : Si `res` est vrai au début de l'étape `i`, alors `tab` est croissant jusqu'à sa position `i + 1`. On sort de la boucle à la première position où `tab` n'est plus croissant, ou bien à la fin du parcours de `tab` si une telle position n'existe pas. Dans le pire cas, on aura parcouru `tab` en entier, et la complexité est alors $O(n)$. En revanche, la complexité en moyenne est bien moindre. Sur une instance aléatoire uniforme, la probabilité que les i premiers éléments soient croissants est $\frac{1}{i!}$, et donc le nombre moyen d'itérations de boucle est $\sum_{i=0}^{n-2} \frac{1}{(i+1)!} < e$. On en conclut que la complexité moyenne est $O(1)$.

3.3 Exemples d'analyse de complexité

Commençons par un exemple simple utilisant des boucles `POUR`. Nous pouvons par exemple décrire un algorithme de tri par sélection qui trie *en place* (en modifiant directement le tableau donné en entrée) un tableau dans l'ordre décroissant.

Algorithme 16 : MaxFrom

```
Entrées : tab : tableau de taille n, i0 : entier inférieur à n  
res ← i0  
pour i de i0 + 1 à n - 1 faire  
|   si tab[i] > tab[res] alors  
|   |   res ← i  
|   fin  
fin  
retourner res
```

La complexité de l'initialisation de l'algorithme est $O(1)$. La complexité de chaque itération de la boucle `POUR` est $O(1)$, et le nombre d'itérations est $n - 1 - i_0$, donc la complexité totale de la boucle est $O(n - i_0)$.

Algorithme 17 : TriSélection

```
Entrées : tab : tableau de taille n  
pour i de 0 à n - 2 faire  
|   i0 ← MaxFrom(t, i)  
|   Echanger(t, i, i0)  
fin
```

La complexité de la i -ème itération de la boucle `POUR` est $O(n - i)$, donc la complexité totale est

$$\sum_{i=0}^{n-2} O(n - i) = O\left(\sum_{i=2}^n i\right) = O(n(n + 1)/2) = O(n^2)$$

Ici, bien que la complexité des itérations successives soit décroissante, cela ne suffit pas à ce que la complexité totale soit sensiblement inférieure à n fois la complexité de la pire itération. On ne gagne qu'un facteur multiplicatif 2, qui est de toute façon "absorbé" par le $O(\cdot)$.

Remarque Si on ne sait pas calculer $\sum_i f(i)$, on peut utiliser le résultat suivant. Pour toute fonction f monotone sur un intervalle $[a, b]$ avec $a \leq b$ entiers, on a

$$\min\{f(a), f(b)\} + \int_a^b f(x) dx \leq \sum_{i=a}^b f(i) \leq \max\{f(a), f(b)\} + \int_a^b f(x) dx.$$

Par conséquent, si f est à (dé)croissance sous-exponentielle, alors remplacer la somme par une intégrale fournit une bonne approximation du résultat.

Les choses se compliquent s'il y a une boucle `Tant-que`. L'analyse de la complexité nécessite d'évaluer son nombre d'itérations, ce qui dépend de l'évolution des valeurs des variables internes. Une stratégie classique est d'étudier la suite (x_i) des valeurs successives de chaque variable x , et de déterminer la plus petite valeur de i telle que x_i satisfait la condition de sortie : il s'agit du nombre d'itérations de la boucle.

Reprenons l'exemple du calcul du pgcd. La condition de sortie ne dépend que de y , donc étudions la suite (y_i) de ses valeurs successives. Bien que l'on puisse avoir deux termes consécutifs très proches dans cette suite (typiquement si $x_i = 2y_i - 1$, alors $y_{i+1} = y_i - 1$), cela n'arrive pas à chaque itération. En effet, si $y_i \leq x_i/2$, alors $x_i \bmod y_i < x_i/2$, et si $y_i > x_i/2$, alors $x_i \bmod y_i \leq x_i - y_i < x_i/2$. Comme $x_i = y_{i-1}$, on a donc $y_{i+1} < y_{i-1}/2$, pour tout $i \geq 1$, et donc $y_{2i} < y_0/2^i$. Au bout de $i = \lceil 2 \log_2 y_0 \rceil$ étapes, on a donc $y_i < 1$, et donc la condition de sortie est satisfaite. Comme chaque itération de boucle a complexité $O(1)$, on conclut que l'algorithme a pour complexité $O(\log \min(n, m))$.

3.4 Amélioration de la complexité d'un algorithme

- Prévoir et éviter les opérations inutiles
- Supprimer les calculs redondants
- Découper le problème

Exemple On cherche à simuler physiquement le mouvement de N boules de billard. La simulation consiste, à chaque pas, à calculer successivement la somme des forces appliquées à chaque boule par (potentiellement) toutes les autres boules puis, à partir de cette force, de calculer le déplacement de chaque boule. Au pas suivant, on calculera de nouveau la somme des forces résultant des nouvelles positions calculées au pas précédent et ainsi de suite.

On notera que, pour le calcul des forces, la partie la plus coûteuse est le calcul de la distance entre les deux boules.

On propose deux variantes pour ce calcul :

- (Algorithme A_1) : pour le calcul des forces, on calcule la distance entre chaque boule et toutes les autres boules. Et en fonction de cette distance, on décidera s'il faut ou non calculer une force.
- (Algorithme A_2) : on subdivise l'espace de la simulation en cases carrées de la taille d'une boule. On associe à chaque case la liste des boules qu'elle contient. Cette liste est remise à jour à chaque pas de simulation. Ces cases permettent de ne pas calculer la distance entre une boule et toutes les autres, mais seulement entre les boules qui se trouvent dans une même case.

La taille de ce problème est N le nombre de boules. D'un point de vue espace mémoire, A_2 est clairement moins performant que A_1 : selon son implémentation, cette complexité peut même dépendre de la taille de la boîte (si on utilise un tableau pour stocker la liste des boules présentes dans chaque case). En revanche, la complexité temporelle de A_1 est moins bonne que celle de A_2 en général. En effet, A_1 calcule les $\binom{N}{2} = \Theta(N^2)$ distances entre chaque paire de boules, tandis que A_2 ne calcule que les distances entre une boule et les boules qui apparaissent dans une même case. Si la taille des cases correspond au diamètre des boules, alors chaque boule intersecte au plus 4 cases, et une case ne pas contenir plus que 7 boules, ce qui signifie que l'on calcule au plus 24 distances pour chaque boule. L'algorithme A_2 calcule donc $O(N)$ distances.

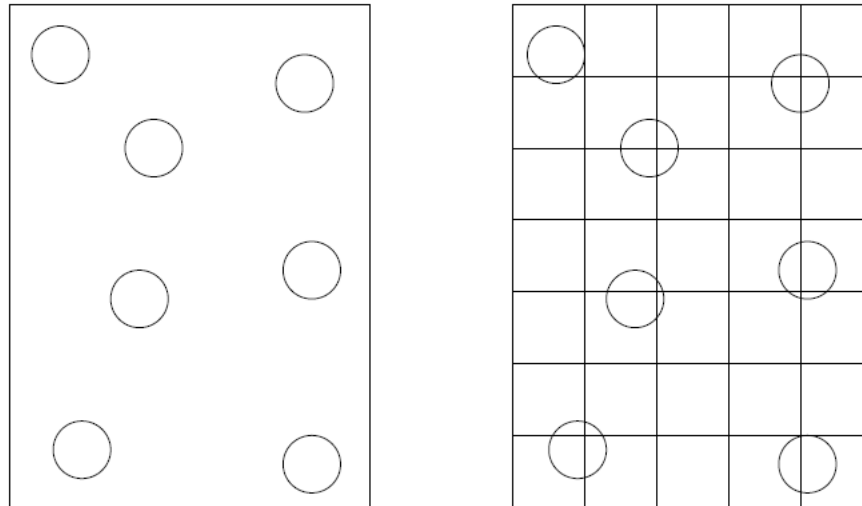


FIGURE 2.1 – Il s’agit de calculer, pour chaque boule, la somme des forces exercées par les autres boules et les parois. À gauche, l’algorithme A_1 et, à droite, l’algorithme A_2 .

3.5 Exemple de calcul complexe : Crible d’Eratosthène

On peut implémenter le crible d’Eratosthène naïvement de la sorte.

Algorithme 18 : Eratosthène

Entrées : n : entier
Sorties : res : liste des entiers premiers inférieurs à n

```

res ← []
T ← (n + 1) * [Vrai]
T[0] ← Faux
T[1] ← Faux
i ← 2
tant que i * i ≤ n faire
    si T[i] alors
        j ← 2 * i
        tant que j ≤ n faire
            T[j] ← Faux
            j ← j + i
        fin
    fin
    i ← i + 1
fin
pour j de 2 à n faire
    si T[j] alors
        res ← res + [j]
    fin
fin

```

Le nombre de fois où l’on fait l’opération *razer un nombre* (en pratique, $T[j] \leftarrow \text{Faux}$) dans l’exécution de Eratosthène (n) est

$$\sum_{p \in \mathbb{P}_n} \frac{n}{p} = n \sum_{p \in \mathbb{P}_n} \frac{1}{p},$$

où \mathbb{P}_n est l'ensemble des nombres premiers inférieurs ou égaux à n , de cardinalité $\pi(n)$. Pour évaluer cette complexité, il faut connaître le comportement de la série des inverses des nombres premiers. On peut utiliser pour cela le théorème des nombres premiers.

Theorem 1 (Théorème des nombres premiers). *Soit p_n le n -ième nombre premier. Alors $p_n \underset{n \rightarrow \infty}{\sim} n \ln n$. Similairement, le nombre $\pi(n)$ de nombres premiers plus petits que n satisfait $\pi(n) \underset{n \rightarrow \infty}{\sim} \frac{n}{\ln n}$.*

On a donc

$$\sum_{p \in \mathbb{P}_n} \frac{1}{p} = \sum_{i=1}^{\pi(n)} \frac{1}{p_i} \sim \frac{1}{2} + \frac{1}{3} + \sum_{i=3}^{\pi(n)} \frac{1}{i \ln i}$$

Soient $a < b \in \mathbb{N}$. Pour toute fonction f décroissante sur l'intervalle $[a-1, b+1]$, on a

$$\int_a^{b+1} f(x) dx \leq \sum_{i=a}^b f(i) \leq \int_{a-1}^b f(x) dx.$$

Or, pour tout $1 < a \leq b$, on a

$$\int_a^b \frac{1}{x \ln x} dx = [\ln \ln x]_a^b = \ln \ln b - \ln \ln a.$$

On conclut que

$$\ln \ln(\pi(n) + 1) - \ln \ln 3 \leq \sum_{i=3}^{\pi(n)} \frac{1}{i \ln i} \leq \ln \ln \pi(n) - \ln \ln 2,$$

et donc $\sum_{p \in \mathbb{P}_n} \frac{1}{p} \sim \ln \ln n$.

Avec l'implémentation naïve, on raye plusieurs fois le même nombre, en moyenne $\ln \ln n$ fois chacun. Pour éviter cela, plutôt que d'itérer sur tous les multiples de i , il faut n'itérer que sur les multiples de i avec un nombre $j \geq i$ pas encore rayé. Il faut ainsi changer la structure de données afin d'être capable d'effectuer cette itération efficacement. Pour cela, on renseigne dans chaque case i du tableau T non rayée la position des cases non rayées précédentes et suivantes, comme on le ferait dans une liste doublement chaînée. Initialement, ces positions sont respectivement $T[i].\text{prec} \leftarrow i-1$ et $T[i].\text{suiv} \leftarrow i+1$. Si l'on souhaite rayer la case i du tableau T pour la première (et unique !) fois, on doit alors faire

Algorithme 19 : Rayer

Entrées : T : tableau doublement chaîné, i : indice

$T[i].\text{rayé} \leftarrow \text{True}$

$i_0 \leftarrow T[i].\text{prec}$

$i_1 \leftarrow T[i].\text{suiv}$

$T[i_0].\text{suiv} \leftarrow i_1$

$T[i_1].\text{prec} \leftarrow i_0$

On verra en TP comment finir cette implémentation optimisée du crible d'Eratosthène. En s'y prenant bien, la complexité est linéaire en le nombre de fois que l'on applique la fonction `Rayer`, et comme on ne raye plus jamais deux fois le même nombre, on a donc une complexité totale en $O(n)$.

Chapitre 3

Réversivité

1 Fonctions récursives

1.1 Définition et usages

Une *fonction récursive* est une fonction qui fait appel à elle-même au sein de son corps, ce que l'on nomme un *appel récursif*. Cet appel doit se faire sur une entrée différente de celle donnée en paramètre à la fonction, autrement l'exécution serait infinie. Une fonction récursive contient toujours un cas de base qui ne fait pas d'appel récursif, mais renvoie immédiatement un résultat. Ainsi, une fonction récursive commence en général toujours par un test permettant de distinguer le cas de base des autres.

Les fonction récursives sont très utiles pour implémenter des opérations mathématiques, qui sont souvent définie de façon récursive. C'est par exemple le cas du calcul du pgcd.

Algorithme 20 : pgcd

Entrées : n, m : entiers
Sorties : plus grand diviseur commun de n et m
si $m = 0$ **alors**
| retourner n
sinon
| retourner $\text{pgcd}(m, n \bmod m)$
fin

1.2 Récursion terminale

Une récursion est terminale si l'appel récursif est la dernière instruction de l'algorithme. Cet appel récursif doit donc se faire dans l'instruction `Retourner` qui doit être forcément *pure*, c'est à dire qu'elle ne fait intervenir aucune opération autre que l'appel récursif.

Algorithme 21 : Taille

(non terminale)

Entrées : lst : liste
Sorties : Taille de la liste
si $\text{EstVide}(lst)$ **alors**
| retourner 0
sinon
| Extraire(lst)
| retourner $1 + \text{Taille}(lst)$
fin

La fonction récursive `taille` n'est pas terminale à cause de l'opération arithmétique effectuée au moment

de l'instruction `Retour`. Pour éviter cela, on peut utiliser une fonction auxiliaire qui prend une variable supplémentaire dont le rôle est stocker le résultat intermédiaire, à la manière d'une variable qui serait mise à jour au sein d'une boucle `Pour`.

Algorithme 22 : TailleAuxiliaire

Entrées : `lst` : liste, `cpt` : entier
Sorties : Taille de la liste
si `EstVide(lst)` **alors**
| **retourner** `cpt`
sinon
| `Extraire(lst)`
| **retourner** `TailleAuxiliaire(lst, cpt+1)`
fin

Invariant : `cpt` est le nombre d'appels récursifs à `TailleAuxiliaire`, c'est à dire le nombre d'élément que l'on a extraits de `lst`.

Algorithme 23 : Taille

(terminale)

Entrées : `lst` : liste
retourner `TailleAuxiliaire(lst, 0)`

Lorsque c'est possible, il faut privilégier l'implémentation d'un algorithme récursif avec une récursion terminale. En effet, lors de l'exécution d'une fonction récursive non-terminale, il faut garder en mémoire l'ensemble des appels récursifs, qui sont ensuite dépilés afin de calculer la solution. Pour une fonction récursive terminale, ce n'est pas nécessaire puisque le dernier appel récursif donne directement le résultat de l'exécution.

2 Structures récursives

La récursivité peut également servir à définir des structures de données. Un type récursif se définit de constructeurs faisant intervenir un ou plusieurs éléments plus petits de même type.

2.1 Définition récursive des listes

On définit le type `t liste`, qui représente des listes dont les éléments constitutifs sont de type `t`. Par exemple, pour une liste d'entiers, on a `t=entier`.

```
type t liste =  
| []           Le cas de base est la liste vide  
| x :: lst : (t * (t liste))
```

Le constructeur `::` s'applique sur un couple (x, lst) ; le type de `x` est `t`, et celui de `lst` est `t liste`. Alors, `x` est l'élément en tête de la liste, et `lst` est le reste de la liste.

Algorithme 24 : EstVide

Entrées : `lst` : liste
retourner `lst=[]`

Algorithme 25 : Extraire

```
Entrées : lst : liste
si lst = x :: r alors
  | lst ← r
  | retourner x
sinon
  | Erreur("Liste vide")
fin
```

2.2 Arbres

On définit tout d'abord les arbres binaires, qui sont des arbres d'arité 2 :

```
type t arbreBinaire =
| Feuille f : t
| Noeud(x, fils_gauche, fils_droit) : (t * (t arbre) * (t arbre))
```

Par exemple, les arbres de décision sont des arbres binaires où la valeur de chaque noeud est un test, et la valeur des feuilles est le résultat en fonction de la séquence des tests depuis la racine.

On peut généraliser à la structure des arbres d'arité k , pour tout $k \geq 1$. L'arité indique le nombre d'enfants de chaque noeud. On note que les arbres d'arité 1 sont structurellement équivalents aux listes.

Puis plus généralement on a les arbres d'arité variable :

```
type t arbre = Noeud(x, enfants) : (t * (t arbre) liste)
```

3 Analyse d'algorithmes récursifs

3.1 Arbre d'appels récursifs

Les appels récursifs lors de l'exécution d'un algorithme récursif ont une structure d'arbre, où chaque noeud a autant d'enfants qu'il y a d'appels récursifs dans le corps de la fonction avec les paramètres correspondants. Un appel récursif qui rentre dans le cas de base correspond à une feuille de l'arbre.

3.2 Complexité

Comme pour les boucles T_q , le calcul de complexité des algorithmes récursifs peut être complexe car il nécessite de garantir que l'exécution de l'algorithme termine. Cependant, dans la plupart des applications, il existe une quantité qui décroît au fil des appels récursifs de la fonction, et qui garantit que l'on atteigne un cas de base en temps borné. Le calcul de complexité se ramène alors à la résolution d'une formule de récurrence qui compte le nombre d'appels récursifs de la fonction.

Considérons par exemple l'algorithme récursif suivant.

Algorithme 26 : Algo

```
Entrées : n : entier
OperationElementaire()
pour i de 0 à n - 1 faire
  | Algo(i)
fin
```

En notant $c(n)$ le nombre d'appels à `OperationElementaire()` par cet algorithme sur une entrée n , on

a donc $c(0) = 1$, et pour tout $n \geq 1$

$$c(n) = 1 + \sum_{i=0}^{n-1} c(i) = c(n-1) + 1 + \underbrace{\sum_{i_0}^{n-2} c(i)}_{c(n-1)} = 2c(n-1).$$

En résolvant cette récurrence, on obtient $c(n) = 2^n$ pour tout $n \geq 0$.

Chapitre 4

Tris

Il est souvent intéressant en informatique de trier une entrée avant de travailler dessus. Par exemple, dans un tableau trié de taille n , la recherche d'un élément a une complexité logarithmique car on peut utiliser une recherche dichotomique. Il est donc important d'être capable de trier une structure de données (tableau, liste) de la manière la plus efficace possible. Pour ce faire, il existe de nombreux algorithmes de tris ayant divers avantages et inconvénients, et leur étude est un très bon cas d'école pour mieux comprendre les enjeux de l'algorithmique.

1 Quelques exemples de tri

1.1 Tris quadratiques

1.1.1 Le tri bulle

Algorithme 27 : TriBulle

```
Entrées : tab : tableau
n ← taille(tab)
répéter
| EstTrié ← Vrai
| pour  $i$  de 0 à  $n - 2$  faire
| | si  $tab[i] > tab[i+1]$  alors
| | | EstTrié ← Faux
| | | Echanger(tab, i, j)
| | fin
| fin
jusqu'à EstTrié
```

Après une itération de la boucle principale, l'élément maximal du tableau est positionné à la dernière case. En effet, dès qu'il est rencontré, il est échangé avec tous les éléments qui suivent, jusqu'à ce qu'il atteigne la dernière case.

Une première conséquence est que le tableau est entièrement trié après au plus n itérations de la boucle principale. En effet, par récurrence, au bout de i itérations, les i éléments maximaux sont correctement positionnés dans le tableau.

Une seconde conséquence est que la boucle `POUR` interne pourrait être stoppée quand $i = n - 1 - k$ à la k -ème itération de la boucle principale, puisqu'alors les k derniers éléments du tableau sont déjà correctement positionnés.

Même avec cette amélioration, la complexité totale de l'algorithme est $\Theta(n^2)$ dans le pire cas, à savoir quand le tableau en entrée est trié dans l'ordre décroissant.

Le tri bulle a cependant un avantage : il est en place, ce qui signifie que sa complexité en espace est $O(1)$: il ne fait appel à aucune structure de données auxiliaire complexe, mais ne fait que des modifications internes au tableau d'entrée.

1.1.2 Tri par insertion

Algorithme 28 : Insérer

```
Entrées :  $x$  : element,  $lst$  : liste triée
si  $EstVide(lst)$  alors
|   retourner  $[x]$ 
sinon
|    $y \leftarrow Extraire(lst)$ 
|   si  $x < y$  alors
|   |   retourner  $x :: (y :: lst)$ 
|   sinon
|   |   retourner  $y :: Insérer(x, lst)$ 
|   fin
fin
```

La complexité de `Insérer` est $\Theta(n)$ sur une liste de taille n , dans le pire cas et en moyenne. Dans le meilleur cas, elle est $O(1)$, si on insère l'élément minimum.

Algorithme 29 : TriInsertion

```
Entrées :  $lst$  : liste
 $res \leftarrow []$ 
pour  $x$  dans  $lst$  faire
|    $Insérer(x, res)$ 
fin
retourner  $lst$ 
```

Sur une entrée de taille n , l'algorithme `TriInsertion` fait n appels à `Insérer`. La complexité dans le pire cas est $\Theta(n^2)$, et est atteinte si la liste en entrée est déjà triée. Cependant, l'algorithme `TriInsertion` est plutôt rapide sur des entrées petites, et est donc souvent utilisé en pratique pour trier des listes sur un petit nombre de valeurs.

1.2 Tris pseudo-linéaires

Il est possible d'écrire des algorithmes de tri donc la complexité est pseudo-linéaire, c'est-à-dire $O(n \ln n)$ sur une entrée de taille n . Ces algorithmes sont à privilégier sur des entrées de grande taille.

1.2.1 Le tri fusion

Le tri fusion est une illustration du principe *Diviser pour Régner*. Il consiste à couper l'entrée en deux, appliquer la fonction récursive sur chacune des deux moitiés, puis recoller la solution. Ce recollement a une complexité linéaire, et cela permet d'atteindre une complexité pseudo-linéaire pour `TriFusion`. En effet, la complexité du corps de la fonction `TriFusion` est $O(n)$. En notant $c(n)$ la complexité totale dans le pire cas de `TriFusion`

Algorithme 30 : TriFusion

Entrées : lst : liste de taille n
si $lst=[]$ ou $lst=[x]$ **alors**
| **retourner** lst
fin
 $(lst1, lst2) \leftarrow \text{CouperEnDeux}(lst)$ //Complexité $O(n)$
retourner $Fusion(TriFusion(lst1), TriFusion(lst2))$

Algorithme 31 : Fusion

Entrées : $lst1, lst2$: listes triées
Sorties : res : union triée de $lst1$ et $lst2$
si $EstVide(lst1)$ **alors**
| **retourner** $lst2$
fin
si $EstVide(lst2)$ **alors**
| **retourner** $lst1$
fin
 $x \leftarrow \text{Extraire}(lst1)$
 $y \leftarrow \text{Extraire}(lst2)$
si $x < y$ **alors**
| **retourner** $x :: Fusion(lst1, y :: lst2)$
sinon
| **retourner** $y :: Fusion(x :: lst1, lst2)$
fin

sur une entrée de taille n , on a donc

$$\begin{aligned}c(n) &= O(n) + 2c(n/2) = O(n) + 2(O(n/2) + 2c(n/4)) \\ &= O(n) + 2O(n/2) + \dots + 2^i O(n/2^i) + \dots + 2^{\log_2 n} c(1) \\ &= O(n \ln n).\end{aligned}$$

Le tri fusion est un des meilleurs algorithmes de tri connus, et est très souvent utilisé en pratique. Par exemple, l'algorithme de tri utilisé par Python est un hybride entre le tri par insertion et le tri fusion.

1.2.2 Le tri rapide

Le tri rapide fonctionne de façon similaire au tri fusion, et utilise lui aussi le principe de Diviser pour Régner. Sa principale différence repose dans le fait que l'opération de recollement ne coûte rien, contrairement à la fonction `Fusion` du tri fusion qui a une complexité linéaire.

On commence par choisir un élément *pivot* x dans le tableau à trier tab , puis on déplace tous les éléments inférieurs à x dans la première partie du tableau, et tous les éléments supérieurs à x dans la seconde partie du tableau. On trie ensuite indépendamment chacune des deux parties du tableau obtenues via deux appels récursifs au tri fusion. Le choix optimal du pivot est l'élément médian du tableau, ce qui permet d'avoir deux parties de même taille sur lesquelles on applique la récursion. Mais un très bon choix consiste à prendre comme pivot un élément aléatoire de tab , ce qui garantit une complexité en moyenne $O(n \ln n)$.

Dans le pire cas, la complexité du tri fusion est $\Theta(n^2)$, si le pivot est systématiquement l'élément minimal de tab . Mais le choix d'un pivot aléatoire fonctionne très bien en pratique, si bien que le tri rapide est souvent plus efficace en pratique que le tri fusion (on note qu'il peut se faire en place, donc avec une complexité en espace $O(1)$).

2 Complexité générale d'un tri

On peut prouver que $\Theta(n \ln n)$ est la meilleure complexité possible dans le pire cas pour un algorithme de tri par comparaison. On suppose pour cela que l'on ne peut connaître la position relative de deux éléments x, y du tableau à trier qu'en faisant le test $x < y$. La complexité du tri sera alors bornée inférieurement par le nombre de tests $x < y$ réalisés.

On procède à une preuve par adversaire. Le principe est le suivant : au cours de l'exécution d'un algorithme de tri arbitraire sur un tableau t_{ab} de taille n , un adversaire choisit le résultat de chacun des tests $x < y$ de sorte à ralentir autant que possible l'algorithme, tout en restant consistant dans ses réponses (il doit toujours exister un ordre des éléments pour lequel les réponses aux tests données par l'adversaire sont cohérentes). Soit \mathcal{P}_i l'ensemble des ordres possibles sur les éléments de t_{ab} compte-tenu des i premiers résultats de tests donnés par l'adversaire. Alors \mathcal{P}_0 est l'ensemble des permutations des n éléments de t_{ab} , de taille $n!$. À chaque instant i , le i -ème test partitionne l'ensemble \mathcal{P}_{i-1} en deux : les ordres pour lesquels le résultat est *Vrai*, et ceux pour lesquels le résultat est *Faux*. L'adversaire choisit systématiquement le résultat qui laisse le plus grand nombre de possibilités. Avec ce choix, on a

$$|\mathcal{P}_{i+1}| \geq \frac{|\mathcal{P}_i|}{2}, \quad (4.1)$$

pour tout $i \geq 0$.

Ainsi, l'algorithme se termine uniquement quand $|\mathcal{P}_i| = 1$, en retournant le seul ordre possible restant. Par l'équation (4.1), on a alors $i \geq \log_2(n!) \sim n \log_2 n$. En effet, $\log_2(n!) = \sum_{i=2}^n \log_2 i$, et on a

$$\begin{aligned} \int_1^n \log_2 x \, dx &\leq \sum_{i=2}^n \log_2 i \leq \int_2^{n+1} \log_2 x \, dx \\ \frac{1}{\ln 2} \left[x \ln x - x \right]_1^n &\leq \sum_{i=2}^n \log_2 i \leq \frac{1}{\ln 2} \left[x \ln x - x \right]_2^{n+1} \\ n \log_2 n - \frac{n}{\ln 2} + 1 &\leq \sum_{i=2}^n \log_2 i \leq (n+1) \log_2(n+1) - \frac{n+1}{\ln 2} + 2 \ln 2 - 2. \end{aligned}$$

Si on la connaît, on peut aussi directement appliquer la formule de Stierling : $n! \sim \left(\frac{n}{e}\right)^n \sqrt{2\pi n}$.